Otimização de Banco de Dados

SUMÁRIO

1. Visão Geral da Otimização

A parte mais importante para obter um sistema rápido é com certeza o projeto básico. Você também precisa saber quais tipos de coisas seus sistema estará fazendo, e quais são gargalos existentes.

Os gargalos mais comuns são:

- Pesquisa em disco É necessário tempo para o disco encontrar uma quantidade de dados. Com discos modernos em 1999, o tempo médio para isto era normalmente menor que 10ms, portanto em teoria poderíamos fazer 100 buscas por segundo. Este tempo melhora moderadamente com discos novos e isso é muito difícil otimizar para uma única tabela. A maneira de otimizar isto é colocando os dados em mais de um disco.
- Leitura de disco/Escrita (I/O) Quando o disco estiver na posição correta precisaremos que os dados sejam lidos. Com discos mais modernos em 1999, um disco retorna algo em torno de 10-20Mb/s. Isto é mais fácil de otimizar que as buscas porque você pode ler vários discos em paralelo.
- Ciclos de CPU. Quando tivermos os dados na memória principal (ou se eles já estiverem lá) precisaremos processá-los para conseguir nosso resultado. O fator de limitação mais comum é ter ppequenas tabelas, comparadas com a memória. Mas, com pequenas tabelas, normalmente não teremos problemas com velocidade.
- Largura de banda da memória. Quando a CPU precisa de mais dados que podem caber no cache da CPU a largura da banda da memória principal se torna um gargalo. Isto é um gargalo muito incomum para a maioria dos sistema, mas é bom estarmos ciente dele.

1.1. Limitações do Projeto MySQL/Trocas

Quando usamos o mecanismos de armazenamento MyISAM, o MySQL utiliza travamento de tabela extremamente rápidos (múltiplas leituras / única escrita). O maior problema com este tipo de tabela ocorre quando você tem uma mistura do fluxo fixo de atualizações e seleções lentas na mesma tabela. Se isto for um problema com algumas tabelas, você pode usa outro tipo de tabela.

O MySQL pode trabalhar com tabelas transacionais e não transacionais. Para trabalhar sem problemas com tabelas não transacionais (nas quais não se pode fazer um rollback se alguma coisa der errada), o MySQL tem as seguintes regras:

- Todas as colunas possuem valor padrão.
- Se você inserir um valor 'errado' em uma coluna, como um **NULL** em uma coluna **NOT NULL** ou um valor numérico muito grande em uma coluna numérica, o MySQL definir a coluna com o 'melhor valor possível' em vez de dar um erro. Para valores numéricos isto é 0, o menor valor possível ou o maior valor possível. Para strings into é tanto uma string vazia quanto a maior string possível que possa estar na coluna.
- Todas as expressões calculadas retornam um valor que pode ser usado em vez de apresentar uma condição de erro. Por exemplo, 1/0 retorna **NULL**

O mostrado acima quer dizer que não se deve usar o MySQL para verificar o conteúdo dos campos, mas deve se fazer isto no aplicativo.

1.2. Portabilidade

Como todos os servidores SQL implementam diferentes partes de SQL, é trabalhoso escrever aplicativos SQL portáveis. Para selects/inserts muito simples é muito fácil, mas quanto mais recursos você precisa, mais difícil se torna. Se você quiser uma aplicação quue é rápida com muitos bancos de dados ela se torna ainda mais difícil.

Para fazer um aplicativo portável complexo você precisa escolher um número de servidores SQL com o qual ele deve trabalhar.

Você pode utilizar o MySQL programa/web-page **crash-me** - http://www.mysql.com/information/crash-me.php - para encontrar funções, tipos e limites que você pode utilizar com uma seleção de servidores de bancos de dados. O Crash-me agora testa quase tudo possível, mas continua compreensível com aproximadamente 450 itens testados.

Por exemplo, você não deve ter nomes de colunas maior do que 18 caracteres se desejar utilizar o Informix ou DB2.

Os programas de benchmarks e **crash-me** do MySQL são bastante independentes do bancos de dados. Dando uma olhada em como nós os tratamos, você pode sentir o que é necessário para escrever sua aplicação independente do banco de dados. Os benchmarks podem ser encontrados no diretório sql-bench na distribuição fonte do MySQL. Eles são escritos em Perl com a interface de banco de dados DBI (que resolve a parte do problema de acesso).

Veja http://www.mysql.com/information/benchmarks.html para os resultados deste benchmark.

Como pode ser visto nestes resultados, todos os bancos de dados tem alguns pontos fracos. Isto é, eles possuem diferentes compromissos de projeto que levam a comportamentos diferentes.

Se você procura por independencia de banco de dados, precisará ter uma boa idéia dos gargalos de cada servidor SQL. O MySQL é muito rápido para recuperação e atualização de dados, mas terá problemas em misturar leituras/escritas lentas na mesma tabela. O Oracle, por outro lado, possui um grande problema quando você tentar acessar registros que foram recentemente atualizados (até eles serem atualizados no disco). Bancos de dados transacionais geralmente não são muito bons gerando tabelas de resumo das tabelas log, nestes casos o travamento de registros é praticamente inútil.

Para fazer sua aplicação *realmente* independente de banco de dados, você precisará definir uma interface que possa ser expandida, por meio da qual você fará a manipulação dos dados. Como o C++ está disponível na maioria dos sistemas, faz sentido utilizar classes C++ para fazer a interface ao banco de dados.

Se você utilizar algum recurso específico para algum banco de dados (como o comando **REPLACE** no MySQL), você deve codificar um método para os outros serviodores SQL para implementar o mesmo recurso (mas mais lento). Com o MySQL você pode utilizar a sintaxe **/*! */** para adicionar palavras chave específicas do MySQL para uma query. O código dentro de **/**/** será tratado como um comentário (ignorado) pela maioria dos servidores SQL.

Se alta performance REAL é mais importante que exatidão, como em algumas aplicações WEB, uma possibilidade é criar uma camada de aplicação que armazena todos os resultados para lhe fornecer uma performance ainda mais alta. Deixando resultados antigos 'expirar' depois de um tempo, você pode manter o cache razoavelmente atual. Isto é muito bom no caso de uma carga extremamente pesada, pois neste caso você pode aumentar o cache dinamicamente e configurar o tempo de expiração maior até que as coisas voltem ao normal.

Neste caso a informação de criação de tabelas devem conter informações do tamanho inicial do cache e com qual frequência a tabela, normalmente, deve ser renovada.

1.3. Para que Utilizamos o MySQL?

Durante o desenvolvimento inicial do MySQL, os recursos do MySQL foram desenvolvidos para atender nosso maior cliente. Eles lidam com data warehousing para alguns dos maiores varejistas na Suécia.

De todas as lojas, obtemos resumos semanais de todas as transações de cartões de bonus e esperamos fornecer informações úteis para ajudar os donos das lojas a descobrir como suas campanhas publicitárias estão afetando seus clientes.

Os dados são bem grandes (cerca de 7 milhões de transações por mês), e armazenamos dados por cerca de 4-10 anos que precisamos apresentar para os usuários. Recebemos requisições semanais dos clientes que desejam ter acesso 'instantâneo' aos novos relatórios contendo estes dados.

Resolvemos este problema armazenando todas informações mensalmente em tabelas com transações compactadas. Temos um conjunto de macros (script) que geram tabelas resumidas agrupadas por diferentes critérios (grupo de produto, id do cliente, loja...) das tabelas com transações. Os relatórios são páginas Web que são geradas dinamicamente por um pequeno shell script que analisa uma página Web, executa as instruções SQL na mesma e insere os resultados. Nós usariamos PHP ou mod_perl mas eles não estavam disponíveis na época.

Para dados graficos escrevemos um ferramenta simples em **C** que pode produzir GIFs baseados no resultado de uma consulta SQL (com alguns processamentos do resultado). Isto também é executado dinamicamente a partir do script Perl que analisa os arquivos HTML.

Na maioria dos casos um novo relatório pode simplesmente ser feito copiando um script existente e modificando a consulta SQL no mesmo. Em alguns casos, precisamos adicionar mais campos a uma tabela de resumo existente ou gerar uma nova, mas isto também é bem simples, pois mantemos todas as tabelas com as transaçõs no disco. (Atualmente possuimos pelo menos 50G de tabelas com transações e 200G de outos dados do cliente.)

Nós também deixamos nossos clientes acessarem as tabelas sumárias diretamente com ODBC para que os usuários avançados possam também fazer experimentar com os dados.

Nós não tivemos nenhum problema lidando com isso em um servidor Sun Ultra SPARCstation (2x200 Mhz) bem modesto. Atualmente atualizamos um de nossos servidores para um UltraSPARC com 2 CPUs de 400 Mhz, e planejamos lidar com transações no nível de produto, o que pode significar um aumento de pelo menos dez vezes nosso volume de dados. Acreditamos que podemos lidar com isto apenas adicionando mais disco aos nossos sistemas.

Também estamos experimentando com Intel-Linux para obter mais poder de CPU por um melhor preço. Agora que possuimos o formato binários do bancos de dados portáveis (a partir da versão 3.23), começaremos a utilizá-lo para partes da aplicação.

Nossa sensação inicial é que o Linux irá atuar muito melhor em cargas baixas e médias e o Solaris irá atuar melhor quando você começar a ter uma carga alta pelo uso extremo de IO de disco, mas ainda não temos nada conclusivo sobre isto. Depois de algumas discussões com um desenvolvedor do kernel do Linux, concluímos que isto pode ser um efeito colateral do Linux; alocar muitos recursos para uma tarefa batch que a performance interativa se torna muito baixa. Isto deixa a máquina muito lenta e sem resposta enquanto grandes batches estiverem em execução. Esperamos que isto tenha um tratamento melhor em futuras versões do kernel Linux.

1.4. O Pacote de Benchmark do MySQL

Esta seção deve conter uma descrição técnica do pacote de benchmarks do MySQL (e **crash-me**), mas a descrição ainda não está pronta. Atualmente, você pode ter uma boa idéia do benchmark verificando os códigos e resultados no diretório sql-bench em qualquer distribuição fonte do MySQL.

Este conjunto de benchmark pretende ser um benchmark que irá dizer a qualquer usuário que operações uma determinada implementação SQL irá realizar bem ou mal.

Note que este benchmark utiliza uma única thead, portanto ele mede o tempo mínimo para as operações realizadas. Planejamos adicionar vários testes multi-threaded no conjunto de benchmark no futuro.

A seguinte tabela mostra alguns resultados comparativos de benchmark para diversos servidores de bancos de dados quando acessados por meio do ODBC em uma máquina Windows NT 4.0.

Lendo 2000000 linhas por índice	Segundos	Segundos
mysql	367	249
mysql_odbc	464	
db2_odbc	1206	
informix_odbc	121126	
ms-sql_odbc	1634	
oracle_odbc	20800	
solid_odbc	877	
sybase_odbc	17614	

Inserindo 350768 linhas	Segundos	Segundos
mysql	381	206
mysql_odbc	619	
db2_odbc	3460	
informix_odbc	2692	
ms-sql_odbc	4012	
oracle_odbc	11291	
solid_odbc	1801	
sybase_odbc	4802	

Para os testes anteriores, o MySQL foi executado com um cache de índices de 8M.

Perceba que a Oracle não está incluída porque eles solicitaram a remoção. Todos benchmarks Oracle devem ser aprovados pela Oracle! Acreditamos que os benchmarks da Oracle são ***MUITO*** tendecioso pois os benchmarks acima devem ser executados supostamente para uma instalação padrão para um único cliente.

Para executar a suite de benchmarks, as seguintes exigências devem ser satisfeitas:

- O pacote de benchamark é fornecido com a distribuição fonte do MySQL, assim você deve ter uma distribuição fonte. Você também pode fazer um download de uma distribuição em http://www.mysql.com/downloads/, ou usar a árvore fonte de desenvolvimento atual.
- Os scripts do benchmark são escritos em Perl e usam o módulo Perl DBI para acessar o servidor de banco de dados, assim o DBI deve estar instalado. Você também precisará do driver DBD espercífico do servidor para cada um dos servidores que você quer testar. Por exemplo, para testar o MySQL, PostgreSQL, e DB2, os módulos DBD::mysql, DBD::Pg e DBD::DB2 devem estar instalados.

O pacote de benchmark está localizado no diretório sql-bench da distribição fonte do MySQL. Para executar o teste de benchmark, altera a localização dentro daquele diretório e execute o script **run-all-tests**:

```
shell> cd sql-bench
shell> perl run-all-tests --server=server_name
```

server_name é um dos servidores suportados. Você pode obter uma lista de todos parâmetros e servidores suportados executando **run-all-tests --help**.

crash-me tenta determinar quais recursos um banco de dados suporta e quais suas capacidades e limitações atuais para a execução de consultas. Por exemplo, ele determina:

- Quais tipos de colunas são suportados
- Quantos índices são suportados
- Quais funções são suportadas
- Qual o tamanho máximo de uma query
- Qual o tamanho máximo de um registro do tipo **VARCHAR**

1.5. Utilizando seus Próprios Benchmarks

Definitivamente você deve fazer benchmarks de sua aplicação e banco de dados para saber quais são os gargalos. Corrigindo (ou substituindo o gargalho com um ``módulo burro'') você pode facilmente identificar o próximo gargalo (e continuar). Mesmo se a performance geral para sua aplicação atualmente é aceitável, você deve pelo menos criar um plano para cada gargalo e decidir como resolvê-lo se algum dia você precisar de performance extra.

Para um exemplo de programas de benchmarks portáveis, consulte o conjunto de benchmarks do MySQL. Você pode pegar qualquer programa deste conjunto e modificá-lo para suas necessidades. Fazendo isto você pode tentar soluções diferentes para seu problema e testar qual é a mais rápida para você.

Outro pacote de benchmark grátis é o **Open Source Database Benchmark** disponível em http://osdb.sourceforge.net/.

É muito comum que um problemas ocorram apenas quando o sistema estiver muito carregado. Nós tivemos alguns clientes que nos contactaram quando eles testaram um sistema em produção e encontraram problemas de carga. Na maioria dos casos, problemas de desempenho ocorrem devido a assuntos relacionados ao projeto básico do banco de dados (busca em tabelas **não são bons** com alta carga) ou problemas com o sistema operacional e de bibliotecaa. A maioria das vezes, estes problemas seriam ***MUITO*** mais fáceis de resolver se os sistemas já não estivessem em uso.

Para evitar problemas deste tipo, você deve colocar algum esforço em testar a performance de toda sua aplicação sobre a pior carga possível! Você pode utilizar o Super Smack para isto. Ele está disponível em: http://www.mysql.com/Downloads/super-smack/super-smack-1.0.tar.gz. Como o nome sugere, ele pode derrubar seu sistema se você solicitar, portanto, utilize-o somente em sistemas de desenvolvimento.

2. Otimizando SELECTs e Outras Consultas

Primeiramente, uma coisa que afeta todas as consultas: Quanto mais complexo seu sistema de permissões, maior a sobrecarga.

Se você não tiver nenhuma instrução **GRANT** realizada, MySQL otmizará a verificação de permissões de alguma forma. Dessa forma, se você possui um volume muito alto, o tempo pode piorar tentando permitir o acesso. Por outro lado, maior verificação de permissões resulta em uma sobrecarga maior.

Se o seu problema é com alguma função explícita do MySQL, você pode sempre consultar o tempo da mesma com o cliente MySQL:

```
mysql> SELECT BENCHMARK(1000000,1+1);
+------------------------+
| BENCHMARK(1000000,1+1) |
+------------------------+
|                      0 |
+------------------------+
1 row in set (0.32 sec)
```

O exemplo acima demonstra que o MySQL pode excutar 1.000.000 expressões + ou - em 0.32 segundos em um **PentiumII 400MHz**.

Todas funções MySQL devem ser bem otimizadas, mas existem algumas excessões e o **benchmark(loop_count,expression)** é uma ótima ferramenta para saber se existe um problema com sua query.

2.1. Sintaxe de EXPLAIN (Obter informações sobre uma SELECT)

```
    EXPLAIN nome_tabela
ou  EXPLAIN SELECT opções_select
```

EXPLAIN nome_tabela é um sinônimo para **DESCRIBE nome_tabela** ou **SHOW COLUMNS FROM nome_tabela**.

Quando uma instrução **SELECT** for precedida da palavra chave **EXPLAIN**, o MySQL explicará como ele deve processar a **SELECT**, fornecendo informação sobre como as tabelas estão sendo unidas e em qual ordem.

Com a ajuda de **EXPLAIN**, você pode ver quando devem ser adicionados índices à tabelas para obter uma **SELECT** mais rápida que utiliza índices para encontrar os registros.

Voce deve executar frequentemente **ANALYZE TABLE** para atualizar estatísticas de tabela tais como a cardinalidade das chaves que podem afetar a escolha que o otimizador faz.

Você também pode ver se o otimizador une as tabelas em uma melhor ordem. Para forçar o otimizador a utilizar uma ordem específica de join para uma instrução **SELECT**, adicione uma cláusula **STRAIGHT_JOIN**.

Para ligações mais complexas, **EXPLAIN** retorna uma linha de informação para cada tabela utilizada na instrução **SELECT**. As tabelas são listadas na ordem que seriam lidas. O MySQL soluciona todas as joins utilizando um método multi-join de varedura simples. Isto significa que o MySQL lê uma linha da primeira tabela, depois encontra uma linha que combina na segunda tabela, depois na terceira tabela e continua. Quando todas tabelas são processadas, ele exibe as colunas selecionadas e recua através da lista de tabelas até uma tabela na qual existem registros coincidentes for encontrada. O próximo registro é lido desta tabela e o processo continua com a próxima tabela.

No MySQL versão 4.1 a saída do **EXPLAIN** foi alterada para funcionar melhor com construções como **UNION**s, subqueries e tabelas derivadas. A mais notável é a adição de duas novas colunas: **id** e **select_type**.

A saída de **EXPLAIN** inclui as seguintes colunas:

- **id**

  Identificador **SELECT**, o número sequêncial desta **SELECT** dentro da consulta.

- **select_type**

  Tipo de cláusula **SELECT**, que pode ser uma das seguintes:

  - **SIMPLE**

    SELECT simples (sem **UNION**s ou subqueries).

  - **PRIMARY**

    SELECT mais externa.

- **UNION**

  Segunda **SELECT** e as **SELECT**s posteriores do **UNION**

- **DEPENDENT UNION**

  Segunda **SELECT** e **SELECT**s posteriores do **UNION**, dependente da subquery exterior.

- **SUBQUERY**

  Primeiro **SELECT** na subquery.

- **DEPENDENT SUBQUERY**

  Primeiro **SELECT**, dependente da subquery exterior.

- **DERIVED**

  SELECT de tabela derivada (subquery na cláusula **FROM**).

- **table**

  A tabela para a qual a linha de saída se refere.

- **type**

O tipo de join. Os diferentes tipos de joins são listados aqui, ordenados do melhor para o pior tipo:

- **system**

  A tabela só tem uma linha (= tabela de sistema). Este é um caso especial do tipo de join **const**.

- **const**

A tabela têm no máximo um registro coincidente, o qual será lido na inicialização da consulta. Como só há um registro, os valores da coluna neste registro podem ser considerados constantes pelo resto do otimizador. Tabelas **const** são muito rápidas e são lidas apenas uma vez!

const é usado quando você compara todas as partes de uma chave **PRIMARY**/**UNIQUE** com restrições:

```
SELECT * FROM const_table WHERE primary_key=1;

SELECT * FROM const_table
WHERE primary_key_part1=1 AND primary_key_part2=2;
```

- **eq_ref**

Uma linha será lida desta tabela para cada combinação de linhas da tabela anterior. Este é o melhor tipo de join depois dos tipos **const**. É usado quando todas as partes do índice são usados pela join e o índice é é único (**UNIQUE**) ou uma chave primária (**PRIMARY KEY**).

eq_ref pode ser usado para coluna indexadas que é comparada com o\ operador =. O item comparado pode ser uma constante ou uma expressão que usa colunas de tabelas que são lidas antes desta tabela.

Nos seguintes examplos, **ref_table** poderá usar **eq_ref**

```
SELECT * FROM ref_table,other_table
WHERE ref_table.key_column=other_table.column;

SELECT * FROM ref_table,other_table
WHERE ref_table.key_column_part1=other_table.column
AND ref_table.key_column_part2=1;
```

- **ref**

Todas as colunas com valores de índices correspondentes serão lidos desta tabela para cada combinação de registros da tabela anterior. **ref** é usado se o join usa apenas o prefixo mais a esquerda da chave, ou se a chave não é única (**UNIQUE**) ou uma chave primária (**PRIMARY KEY**) (em outras palavras, se a join não puder selecionar um único registro baseado no valor da chave). Se a chave que é usada coincide apenas em alguns registros, este tipo de join é bom.

ref pode ser usado para colunas indexadas que são comparadas com o operador =.

Nos seguintes exemplos, **ref_table** poderá usar **ref**

```
SELECT * FROM ref_table WHERE key_column=expr;

SELECT * FROM ref_table,other_table
WHERE ref_table.key_column=other_table.column;

SELECT * FROM ref_table,other_table
WHERE ref_table.key_column_part1=other_table.column
AND ref_table.key_column_part2=1;
```

- **ref_or_null**

Como **ref**, mas com o adicional que faremos uma busca extra para linhas com **NULL**.

```
SELECT * FROM ref_table WHERE key_column=expr OR key_column IS NULL;
```

Esta otimização do tipo join é nova para o MySQL 4.1.1 e é mais usada na resolução de sub queries.

- **range**

Apenas registros que estão numa dada faixa serão retornados, usando um índice para selecionar os registros. A coluna **key** indica qual índice é usado. **key_len** contém a maior parte da chave que foi usada. A coluna **ref** será **NULL** para este tipo.

range pode ser usado para quando uma coluna de chave é comparada a uma constante com =, <>, >, >=, <, <=, **IS NULL**, <=>, **BETWEEN** e **IN**.

```
SELECT * FROM range_table WHERE key_column = 10;

SELECT * FROM range_table WHERE key_column BETWEEN 10 and 20;

SELECT * FROM range_table WHERE key_column IN (10, 20, 30);

SELECT * FROM range_table WHERE key_part1= 10 and key_part2 IN (10,
20, 30);
```

- **index**

Isto é o mesmo que **ALL**, exceto que apenas a árvore de índice é varrida. Isto é normalmente mais rápido que **ALL**, já que o arquivo de índice normalmente é menor que o arquivo de dados.

Ele pode ser usado quando a consulta só usa colunas que são parte de um índice.

- **ALL**

Será feita uma varredura completa da tabela para cada combinação de registros da tabela anterior. Isto normalmente não é bom se a tabela é a primeiro tabela não marcada como **const**, e normalmente ***muito*** ruim em todos os casos ordenados. Você normalmente pode ebitar **ALL** adicionando mais índices, assim o registro pode ser retornado baseado em valores constantes ou valores de colunas de tabelas anteriores.

- **possible_keys**

A coluna **possible_keys** indica quais índices o MySQL pode utilizar para encontrar os registros nesta tabela. Note que esta coluna é totalmente independente da ordem das tabelas. Isto significa que algumas das chaves em **possible_keys** podem não ser usadas na prática com a ordem de tabela gerada.

Se esta coluna for **NULL**, não existem índices relevantes. Neste caso, você poderá melhora a performance de sua query examinando a cláusula **WHERE** para ver se ela refere a alguma coluna ou colunas que podem ser indexadas. Se for verdade, crie um índice apropriado e confira a consulta com **EXPLAIN** novamente.

Para ver os índices existentes em uma tabela, utilize **SHOW INDEX FROM nome_tabela**.

- **key**

A coluna **key** indica a chave (índice) que o MySQL decidiu usar. A chave será **NULL** se nenhum índice for escolhido. Para forçar o MySQL a usar um índice listado na coluna **possible_keys**, use **USE INDEX/IGNORE INDEX** em sua consulta.

Executando **myisamchk --** ou **ANALYSE TABLE** na tabela também ajudará o otimizador a escolher índices melhores.

- **key_len**

  A coluna **key_len** indica o tamanho da chave que o MySQL decidiu utilizar. O tamanho será **NULL** se **key** for **NULL**. Note que isto nos diz quantas partes de uma chave multi-partes o MySQL realmente está utilizando.

- **ref**

  A coluna **ref** exibe quais colunas ou contantes são usadas com a **key** para selecionar registros da tabela.

- **rows**

  A coluna **rows** informa o número de linhas que o MySQL deve examinar para executar a consulta.

- **Extra**

Esta coluna contem informações adicionais de como o MySQL irá resolver a consulta. A seguir uma explicação das diferentes strings de texto que podem ser encontradas nesta coluna:

  - **Distinct**

O MySQL não continuará a procurar por mais registros para a combinação de registro atual depois de ter encontrado o primeiro registro coincidente.

  - **Not exists**

O MySQL estava apto a fazer uma otimização **LEFT JOIN** na consulta e não examinará mais registros nesta tabela para a combinação do registro anterior depois que encontrar um registro que satisfaça o critério do **LEFT JOIN**.

Exemplo:

```
SELECT * FROM t1 LEFT JOIN t2 ON t1.id=t2.id
WHERE t2.id IS NULL;
```

Assume que **t2.id** é definido com **NOT NULL**. Neste caso o MySQL irá percorrer **t1** e procurar pelos registros em **t2** através de **t1.id**. Se o MySQL encontrar um registro combinando em **t2**, ele sabe que **t2.id** nunca poderá ser **NULL** e não ir percorrer até o resto dos registros em **t2** que possuirem o mesmo **id**. Em outras

palavras, para cada registro em **t1** o MySQL só precisa fazer uma única pesquisa em **t2**, independente de quantos registros coincidentes existirem em **t2**.

- **range checked for each record (index map: #)**

O MySQL não encontrou um bom índice para usar. No lugar, ele irá fazer uma verificação sobre qual índice usar (se existir) para cada combinação das tabelas precedentes, e usará este índice para recuperar os registros da tabela. Isto não é muito rápido mas é mais rápido que fazer um join sem um índice.

- **Using filesort**

O MySQL precisará fazer uma passada extra para descobrir como recuperar os registros na ordem de classificação. A classificação é feita indo através de todos os registros de acordo com **join type** e armazenar a chave de ordenação mais o ponteiro para o registro para todos os registros que combinarem com o **WHERE**. Então as chaves são classificadas. Finalmente os registros são recuperados na ordem de classificação.

- **Using index**

A informação da coluna é recuperada da tabela utilizando somente informações na árvore de índices sem ter que fazer uma pesquisa adicional para ler o registro atual. Isto pode ser feito quando todas as colunas usadas para a tabela fizerem parte do mesmo índice.

- **Using temporary**

Para resolver a consulta, o MySQL precisará criar uma tabela temporária para armazenar o resultado. Isto acontece normalmente se você fizer um **ORDER BY** em um conjunto de colunas diferentes das quais você fez um **GROUP BY**.

- **Using where**

Uma cláusula **WHERE** será utilizada para restringir quais registros serão combinados com a próxima tabela ou enviar para o cliente. se você não possui esta informação e a tabela é do tipo **ALL** ou **index**, pode existir alguma coisa errada na sua query (Se você não pretender examinar todos os registros da tabela).

Se você desejar deixar suas consultas o mais rápido possível, você deve dar uma olhada em **Using filesort** e **Using temporary**.

Você pode ter uma boa indicação de quão boa é sua join multiplicando todos os valores na coluna **rows** na saída de **EXPLAIN**. Isto deve dizer a grosso modo quantos registros o MySQL deve examinar para executar a consulta. Este número é também usado quando você restringe consultas com a variável **max_join_size**.

O exemplo a seguir mostra como um **JOIN** pode ser otimizado progressivamente utilizando a informação fornecida por **EXPLAIN**.

Suponha que você tem a instrução **SELECT** exibida abaixo, que você está examinando utilizando **EXPLAIN**:

```
EXPLAIN SELECT tt.TicketNumber, tt.TimeIn,
            tt.ProjectReference, tt.EstimatedShipDate,
            tt.ActualShipDate, tt.ClientID,
            tt.ServiceCodes, tt.RepetitiveID,
            tt.CurrentProcess, tt.CurrentDPPerson,
            tt.RecordVolume, tt.DPPrinted, et.COUNTRY,
            et_1.COUNTRY, do.CUSTNAME
        FROM tt, et, et AS et_1, do
        WHERE tt.SubmitTime IS NULL
            AND tt.ActualPC = et.EMPLOYID
            AND tt.AssignedPC = et_1.EMPLOYID
            AND tt.ClientID = do.CUSTNMBR;
```

Para este exemplo, assuma que:

- As colunas comparadas foram declaradas como a seguir:

Tabela	Coluna	Tipo da coluna
tt	ActualPC	CHAR(10)
tt	AssignedPC	CHAR(10)
tt	ClientID	CHAR(10)
et	EMPLOYID	CHAR(15)
do	CUSTNMBR	CHAR(15)

- As tabelas possuem os índices mostrados abaixo:

Tabela	Índice
tt	ActualPC
tt	AssignedPC
tt	ClientID
et	EMPLOYID (chave primária)
do	CUSTNMBR (chave primária)

- The **tt.ActualPC** values aren't evenly distributed.

Initially, before any optimizations have been performed, the **EXPLAIN** statement produces the following information:

```
table type possible_keys                key  key_len ref  rows  Extra
et    ALL  PRIMARY                      NULL NULL    NULL 74
do    ALL  PRIMARY                      NULL NULL    NULL 2135
et_1  ALL  PRIMARY                      NULL NULL    NULL 74
tt    ALL  AssignedPC,ClientID,ActualPC NULL NULL    NULL 3872
      range checked for each record (key map: 35)
```

Como o **tipo** é **ALL** em todas tabelas, esta saída indica que o MySQL está gerando um produto Cartesiano de todas as tabelas! Isto levará muito tempo para ser executado, pois o produto do número de registros em cada tabela deve ser examinado ! Neste caso, existem **74 * 2135 * 74 * 3872** registros. Se as tabelas forem maiores, imagine quanto tempo este tipo de consulta pode demorar.

Um dos problemas aqui é que o MySQL não pode (ainda) utilizar índices em colunas de maneira eficiente se elas foram declaras ide forma diferente. Neste contexto, **VARCHAR** e **CHAR** são o mesmo a menos que tenham sido declarados com tamanhos diferentes. Como **tt.ActualPC** é declarado como **CHAR(10)** e **et.EMPLOYID** é declarado como **CHAR(15)**, existe aqui uma diferença de tamanho.

Para corrigir esta diferença entre tamanhos de registros, utilize **ALTER TABLE** para alterar o tamanho de **ActualPC** de 10 para 15 caracteres:

```
mysql> ALTER TABLE tt MODIFY ActualPC VARCHAR(15);
```

Agora ambos campos **tt.ActualPC** e **et.EMPLOYID** são **VARCHAR(15)**. Executando a instrução **EXPLAIN** novamente produzirá este resultado:

```
table type   possible_keys   key     key_len ref         rows    Extra
tt    ALL    AssignedPC,ClientID,ActualPC NULL NULL NULL 3872    Using
where
do    ALL    PRIMARY         NULL    NULL    NULL        2135
      range checked for each record (key map: 1)
et_1  ALL    PRIMARY         NULL    NULL    NULL        74
      range checked for each record (key map: 1)
et    eq_ref PRIMARY         PRIMARY 15      tt.ActualPC 1
```

Isto não está perfeito, mas está bem melhor (o produto dos valores de **rows** agora menor por um fator de 74). Esta versão é executada em vários segundos.

Uma segunda alteração pode ser feita para eliminar as diferenças de tamanho das colunas para as comparações **tt.AssignedPC = et_1.EMPLOYID** e **tt.ClientID = do.CUSTNMBR**

```
mysql> ALTER TABLE tt MODIFY AssignedPC VARCHAR(15),
    ->                MODIFY ClientID   VARCHAR(15);
```

Agora **EXPLAIN** produz a saída mostrada abaixo:

```
table type   possible_keys   key      key_len ref          rows Extra
et    ALL    PRIMARY         NULL     NULL    NULL         74
tt    ref    AssignedPC,     ActualPC 15      et.EMPLOYID  52   Using
where
             ClientID,
             ActualPC
et_1  eq_ref PRIMARY         PRIMARY  15      tt.AssignedPC 1
do    eq_ref PRIMARY         PRIMARY  15      tt.ClientID   1
```

Este resultado é quase o melhor que se pode obter.

O problema restante é que, por padrão, o MySQL assume que valores na coluna **tt.ActualPC** estão distribuídos igualmente, e este não é o caso para a tabela **tt**. Felizmente, é fácil informar ao MySQL sobre isto:

```
shell> myisamchk --analyze PATH_TO_MYSQL_DATABASE/tt
shell> mysqladmin refresh
```

Agora a join está perfeita, e **EXPLAIN** produz esta saída:

```
table type   possible_keys key      key_len ref           rows Extra
tt    ALL    AssignedPC    NULL     NULL    NULL          3872 Using
where
             ClientID,
             ActualPC
et    eq_ref PRIMARY       PRIMARY 15      tt.ActualPC   1
et_1  eq_ref PRIMARY       PRIMARY 15      tt.AssignedPC 1
do    eq_ref PRIMARY       PRIMARY 15      tt.ClientID   1
```

Perceba que a coluna **rows** na saída de **EXPLAIN** é uma boa ajuda para otimizador de joins do MySQL. Para otimizar uma consulta, você deve conferir se os números estão perto da realidade. Se não, você pode obter melhor desempenho utilizando **STRAIGHT_JOIN** em sua instrução **SELECT** e tentar listar as tabelas em uma ordem diferente na cláusula **FROM**.

2.2. Estimando o Desempenho de uma Consulta

Na maioria dos casos você pode estimar a performance contando buscas em disco. Para tabelas pequenas, normalmente você pode encontrar o registro com 1 pesquisa em disco (uma vez que o índice provavelmente está no cache). Par tabelas maiores, você pode estimar (usando indíces de arvores B++) que você precisará de: **log(row_count) / log(index_block_length / 3 * 2 / (index_length + data_pointer_length)) + 1** buscas em disco para encontrar um registro.

No MySQL um bloco de índice tem geralmente 1024 bytes e o ponteiro de dados 4 bytes. Uma tabela de 500.000 registros com um índice com tamanho de 3 (inteiro médio) lhe dá: **log(500,000)/log(1024/3*2/(3+4)) + 1** = 4 pesquisas.

Como o índice acima necessita cerca de 500,000 * 7 * 3/2 = 5.2M, (assumindo que os buffers de índices são carregados até 2/3, que é o normal) você provavelmente terá grande parte dos índices em memória e provavelmente precisará somente de 1 ou 2 chamadas para ler dados do SO para encontrar o registro.

Entretanto, para escritas, você precisará utilizar 4 requisições para encontrar onde posicionar o novo índice e normalmente 2 buscas para atualizar o índice e escrever o registro.

Perceba que o que foi dito acima não significa que sua aplicação perderá performance por N log N! Como tudo é armazenado no cache de seu SO ou do servidor SQL as coisas começarão a ficar um pouco mais lentas quando as tabelas começarem a

crescer. Quando os dados se tornam muito grandes para o cache, as coisas começarão a ficar bem mais lentas até que suas aplicações estejam limitadas a buscas em disco (o que aumenta em N log N). Para evitar isto, aumente o cache de índice quando os dados crescerem.

2.3. Velocidade das Consultas que Utilizam SELECT

Em geral, quando você desejar tornar uma consulta lenta **SELECT ... WHERE** mais rápida, a primeira coisa que deve ser conferida é se você pode ou não adicionar um índice. Todas as referências entre diferentes tabelas devem ser feitas normalmente com índices. Você pode utilizar o comando **EXPLAIN** para determinas quais índices são usados para uma **SELECT.**

Algumas dicas gerais:

- Para ajudar o MySQL a otimizar melhor as consultas, execute **myisamchk --analyze** em uma tabela depois dela ter sido carregada com dados relevantes. Isto atualiza um valor para cada parte do índice que indica o número médio de registros que tem o mesmo valor. (Para índices únicos, isto é sempre 1, é claro). O MySQL usará isto para decidir qual índice escolher quando você conectar duas tabelas utilizando uma 'expressão não constante'. Os resultados de **analyze** podem ser conferidos utilizando **SHOW INDEX FROM nome_tabela** e examindo a coluna **Cardinality**.
- Para ordenar um índice e dados de acordo com um índice, utilize **myisamchk --sort-index --sort-records=1** (se você deseja ordenar pelo índice 1). Se você possui um índice unico no qual deseja ler todos registros na ordem do índice, esta é uma boa forma para torná-lo mais rápido. Perceba entretanto, que esta ordenação não foi escrita de maneira otimizada e levará muito tempo em tabelas grandes!

2.4. Como o MySQL Otimiza Cláusulas WHERE

As otimizações **WHERE** são colocadas aqui na parte da **SELECT** porque normalmente elas são usadas com **SELECT**, mas as mesmas otimizações aplicam-se para **WHERE** em instruções **DELETE** e **UPDATE**.

Note também que esta seção está incompleta. O MySQL faz várias otimizações e ainda não tivemos tempo para documentarmos todas elas.

Algumas das otimizações feitas pelo MySQL são são listadas abaixo:

Remoção de parênteses desnecessários:

```
•          ((a AND b) AND c OR (((a AND b) AND (c AND d))))
•       -> (a AND b AND c) OR (a AND b AND c AND d)
```

Enlaços de constantes:

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

Remoção de condições contantes (necessário por causa dos enlaços de contantes):

```
          (B>=5 AND B=5) OR (B=6 AND 5=5) OR (B=7 AND 5=6)
      -> B=5 OR B=6
```

Expressões constantes utilizadas por índices são avaliadas somente uma vez.

- **COUNT(*)** em uma única tabela sem um **WHERE** é recuperado diretamente da informação da tabela dos tipos **MyISAM** e **HEAP**. Isto também é feito para qualquer expressão **NOT NULL** quando usada somente com uma tabela.
- Pré detecção de expressões contantes inválidas. O MySQL detecta rapidamente que algumas instruções **SELECT** são impossíveis e não retornará registros.
- **HAVING** é fundido com **WHERE** se não for utilizado **GROUP BY** ou funções de agrupamento (**COUNT()**, **MIN()**...).
- Para cada sub-join, um **WHERE** mais simples é construído para obter uma avaliação mais rápida de **WHERE** para cada sub-join e também para saltar registros da maneira mais rápida possível.
- Todas tabelas constantes são lidas primeiro, antes de qualquer tabelas na consulta. Uma tabela constante é:
  - Uma tabela vazia ou uma tabela com 1 registro.
  - Uma tabela que é usada com uma cláusula **WHERE** em um índice **UNIQUE**, ou uma **PRIMARY KEY**, onde todas as partes do índice são usadas com expressões constantes e as partes do índice são definidas como **NOT NULL**.

Todas as tabelas seguintes são usadas como tabelas constantes:

```
mysql> SELECT * FROM t WHERE primary_key=1;
mysql> SELECT * FROM t1,t2
    -> WHERE t1.primary_key=1 AND t2.primary_key=t1.id;
```

- A melhor combinação de join para unir as tabelas é encontrada tentando todas as possibilidades. Se todas colunas em **ORDER BY** e em **GROUP BY** vierem da mesma tabela, então esta tabela será preferencialmente a primeira na união.
- Se existerem uma cláusula **ORDER BY** e uma **GROUP BY** diferente, ou se a **ORDER BY** ou **GROUP BY** conterem colunas de tabelas diferentes da primeira tabela na fila de join, uma tabela temporária será criada.
- Se você utilizar **SQL_SMALL_RESULT**, o MySQL usará a tabela temporária em memória.
- Cada índice de tabela é consultado e o melhor índice que cobrir menos de 30% dos registros é usado. Se nenhum índice for encontrado, uma varredura rápida é feita pela tabela.
- Em alguns casos, o MySQL pode ler registros do índice mesmo sem consultar o arquivo de dados. Se todas colunas usadas do índice são numéricas, então somente a árvore de índice é usada para resolver a consulta.

- Antes de dar saída em cada registro, aqueles que não combinam com a cláusula **HAVING** são ignorados.

Alguns exemplos de querrys que são mais rápidas:

```
mysql> SELECT COUNT(*) FROM tbl_name;
mysql> SELECT MIN(key_part1),MAX(key_part1) FROM tbl_name;
mysql> SELECT MAX(key_part2) FROM tbl_name
    ->        WHERE key_part_1=constant;
mysql> SELECT ... FROM tbl_name
    ->        ORDER BY key_part1,key_part2,... LIMIT 10;
mysql> SELECT ... FROM tbl_name
    ->        ORDER BY key_part1 DESC,key_part2 DESC,... LIMIT 10;
```

As seguintes consultas são resolvidas utilizando somente a árvore de índices (assumindo que as colunas indexadas são numéricas):

```
mysql> SELECT key_part1,key_part2 FROM tbl_name WHERE key_part1=val;
mysql> SELECT COUNT(*) FROM tbl_name
    ->        WHERE key_part1=val1 AND key_part2=val2;
mysql> SELECT key_part2 FROM tbl_name GROUP BY key_part1;
```

As consultas a seguir utilizam indexação para recuperar os registros na ordem de classificação sem um passo de ordenação separado:

```
mysql> SELECT ... FROM tbl_name
    ->            ORDER BY key_part1,key_part2,... ;
mysql> SELECT ... FROM tbl_name
    ->            ORDER BY key_part1 DESC,key_part2 DESC,... ;
```

2.5. Como o MySQL Otimiza IS NULL

O MySQL pode fazer a mesma otimização em **column IS NULL** que ele pode com **column = constant_value**. Por exemplos, o MySQL pode usar índices e faixas para buscar por **NULL** com **IS NULL**.

```
SELECT * FROM table_name WHERE key_col IS NULL;

SELECT * FROM table_name WHERE key_col <=> NULL;

SELECT * FROM table_name WHERE key_col=# OR key_col=# OR key_col IS
NULL
```

Se você usa **column_name IS NULL** em um **NOT NULL** em uma cláusula WHERE na tabela que não é usada no **OUTER JOIN**, esta espressão será otimizada de qualquer forma.

O MySQL 4.1. pode adicionalmente otimizar a combinação **column = expr AND column IS NULL**, uma forma que é comum em sub queries resolvidas. **EXPLAIN** mostrará **ref_or_null** quando esta otimização é usada.

Esta otimização pode tratar um **IS NULL** para qualquer parte da chave.

Alguns exemplos de consultas que são otimizadas (assumindo chave em t2 (a,b)):

```
SELECT * FROM t1 WHERE t1.a=expr OR t1.a IS NULL;

SELECT * FROM t1,t2 WHERE t1.a=t2.a OR t2.a IS NULL;

SELECT * FROM t1,t2 WHERE (t1.a=t2.a OR t2.a IS NULL) AND t2.b=t1.b;

SELECT * FROM t1,t2 WHERE t1.a=t2.a AND (t2.b=t1.b OR t2.b IS NULL);

SELECT * FROM t1,t2 WHERE (t1.a=t2.a AND t2.a IS NULL AND ...) OR
(t1.a=t2.a AND t2.a IS NULL AND ...);
```

ref_or_null funciona fazendo primeiro uma leitura na chave indicada e depois disto uma busca separada por linhas com chave NULL.

Note que a otimização só pode tratar um nível **IS NULL**.

```
SELECT * FROM t1,t2 where (t1.a=t2.a AND t2.a IS NULL) OR (t1.b=t2.b
AND t2.b IS NULL);
```

No caso acima o MySQL só usará busca de chave na parte **(t1.a=t2.a AND t2.a IS NULL)** e não poderá usar a parte da chave em **b**.

2.6. Como o MySQL Otimiza Cláusulas DISTINCT

DISTINCT combinado com **ORDER BY** também irá em vários casos criar uma tabela temporária.

Note que como **DISTINCT** pode usar **GROUP BY**, você deve estar ciente de como o MySQL funciona com campos na parte **ORDER BY** ou **HAVING** que não são parte dos campos selecionados. See Secção 6.3.7.3, “**GROUP BY** com Campos Escondidos”.

Quando combinando **LIMIT row_count** com **DISTINCT**, o MySQL irá parar logo que encontrar **row_count** registros únicos.

Se você não utiliza colunas de todas tabelas usadas, o MySQL irá parar a varredura das tabelas não usadas logo que encontrar a primeira coincidência.

```
SELECT DISTINCT t1.a FROM t1,t2 where t1.a=t2.a;
```

Neste caso, assumindo que **t1** é usando antes de **t2** (confira com **EXPLAIN**), MySQL irá parar de ler de **t2** (para aquele registro particular em **t1**) quandoo primeiro registro em **t2** for encontrado.

2.7. Como o MySQL Otimiza LEFT JOIN e RIGHT JOIN

A LEFT JOIN B join_condition no MySQL está implementada como a seguir:

- A tabela **B** é configurada para ser dependente da tabela **A** e de todas as tabelas das quais **A** depende.
- A tabela **A** é configurada para ser dependente de todas as tabelas (exceto **B**) que são usadas na condição **LEFT JOIN**.
- A condição **LEFT JOIN** é usada para decidir como devemos recuperar registros a partir da tabela B. (Em outras palavras, qualquer condição na claúsula **WHERE** não é usada).
- Todas as otimizações padrões de join são feitas, com a excessão que uma tabela é sempre lida depois de todas as tabelas das quais é dependente. Se existir uma dependência circular o MySQL irá emitir um erro.
- Todas as otimizações padrões de **WHERE** são realizadas.
- Se existir um registro em **A** que coincida com a cláusula **WHERE**, mas não existir nenhum registro em **B** que coincida com a condição **ON** então um registro extra em **B** é gerado com todas as colunas com valor **NULL**.
- Se você utiliza **LEFT JOIN** para encontrar registros que não existem em alguma tabela e está usando o seguinte teste: **nome_coluna IS NULL** na parte **WHERE**, onde nome_colun é um campo que é declarado como **NOT NULL**, então o MySQL para de pesquisar por mais registros (para uma combinação particular de chaves) depois de ter encontrado um registro que combinar com a condição **LEFT JOIN**.

RIGHT JOIN é implementado de forma análoga à **LEFT JOIN**.

A ordem de leitura das tabelas forçada por **LEFT JOIN** e **STRAIGHT JOIN** irá ajudar o otimizador de joins (que calcula em qual ordem as tabelas devem ser unidas) a fazer seu trabalho mais rapidamente, já que haverão poucas permutações de tabelas a serem conferidas.

Perceba que o texto acima significa que se você fizer uma consulta do tipo:

```
SELECT * FROM b,a LEFT JOIN c ON (c.key=a.key) LEFT JOIN d
(d.key=a.key)
        WHERE b.key=d.key
```

A partir do MySQL 4.0.14, o MySQL faz a seguinte otimização **LEFT JOIN**:

Se a condição **WHERE** é sempre falsa para a linha **NULL** gerada, o **LEFT JOIN** é alterado para um join normal.

Por exemplo, na seguinte consulta a cláusula **WHERE** seria falso se t2.coluna fosse **NULL**, asssim é seguro converter para uma join normal.

```
SELECT * FROM t1 LEFT t2 ON (column) WHERE t2.column2 =5;
->
SELECT * FROM t1,t2 WHERE t2.column2=5 AND t1.column=t2.column;
```

Isto pode ser feito mais rápido já que o MySQL pode agora usar a tabela **t2** antes da tabela **t1** se resultasse consulta melhor. Para forçar uma ordem de tabela específica, use **STRAIGHT JOIN**.

O MySQL irá fazer uma pesquisa completa em **b** já que o **LEFT JOIN** irá força-lo a ser lido antes de **d**.

A correção neste caso é alterar a consulta para:

```
SELECT * FROM b,a LEFT JOIN c ON (c.key=a.key) LEFT JOIN d
(d.key=a.key)
         WHERE b.key=d.key
```

2.8. Como o MySQL Otimiza Cláusulas ORDER BY

Em alguns casos o MySQL pode utilizar índices para satisfazer uma requisição de **ORDER BY** ou **GROUP BY** sem fazer uma ordenação extra.

O índice também pode ser usado mesmo se o **ORDER BY** não coincidir exatamente com o índice, uma vez que todas as partes de índices não usadas e todos os extras na coluna **ORDER BY** são constantes na cláusula **WHERE**. A seguinte consulta usará o índice para resolver a parte **ORDER BY** / **GROUP BY**:

```
SELECT * FROM t1 ORDER BY key_part1,key_part2,...
SELECT * FROM t1 WHERE key_part1=constante ORDER BY key_part2
SELECT * FROM t1 WHERE key_part1=constante GROUP BY key_part2
SELECT * FROM t1 ORDER BY key_part1 DESC,key_part2 DESC
SELECT * FROM t1 WHERE key_part1=1 ORDER BY key_part1 DESC,key_part2
DESC
```

Alguns casos onde o MySQL ***não*** pode usar índices para resolver o **ORDER BY**: (Note que o MySQL ainda usará índices para encontrar o registro que coincide com a cláusula **WHERE**):

- Você está fazendo um **ORDER BY** em diferentes chaves:

  SELECT * FROM t1 ORDER BY key1,key2

- Você está fazendo um **ORDER BY** usando partes de chaves não consecutivas.

  SELECT * FROM t1 WHERE key2=constant ORDER BY key_part2

- Você está misturando **ASC** e **DESC**.

  SELECT * FROM t1 ORDER BY key_part1 DESC,key_part2 ASC

- As chaves usadas para buscar os registros são as mesmas usadas para fazer o **ORDER BY**:

SELECT * FROM t1 WHERE key2=constant ORDER BY key1

- Você está unindo muitas tabelas e as colunas nas quais você está fazendo um **ORDER BY** não são todas da primeira tabela que não é **const** e que é usada para retornar registros. (Esta é a primeira tabela na saída do **EXPLAIN** que não usa um método de busca de registro **const**).
- Você tem diferentes expressões **ORDER BY** e **GROUP BY**.
- O índice da tabela usada é um tipo de índice que não armazena registros em ordem. (Como o índice **HASH** em tabelsn **HEAP**).

Nestes casos onde o MySQL tem que ordenar o resultado, ele usa o seguinte algoritmo:

- Lê todos os registros de acordo com a chave ou por uma varredura da tabela. Registros que não coincidem com a cláusula **WHERE** são saltados.
- Armazena a chave ordenada em um buffer (de tamanho **sort_buffer**).
- Quando o buffer ficar cheio, execute ordeno-o e armazene o resultado em um arquivo temposrário. Salve um ponteiro para o bloco ordenado. (No caso de todos os regitros caberem no buffer ordenado, nenhum arquivo temporário é criado).
- Repete o armazenamento acima até todas as linhas tenham sido lidos.
- Faz um multi-merge até **MERGEBUFF** (7) regiões para um bloco em outro arquivo temporário. Repete até que todos os blocos do primeiro arquivo estejam no segundo arquivo.
- Repete o seguinte até que restem menos que **MERGEBUFF2** (15) blocos.
- No último multi-merge, só o ponteiro para o registro (última parte de chave ordenada) é escrito em um arquivo de resultado.
- Agora o código em sql/records.cc será usado para ler através deles ordenadamente usando os ponteiros de registro no arquivo resultante. Para otimização , lemos em um grande bloco de ponteiros de registros, ordena-os então lemos o registros ordenadamente de de um buffer de registro. (**read_rnd_buffer_size**) .

Você pode verificar com **EXPLAIN SELECT ... ORDER BY** se o MySQL pode usar índices para resolver a consulta. Se você obtiver **Using filesort** na coluna **extra**, então o MySQL não pode usar índices para resolver o **ORDER BY**.

Se você quiser ter uma velocidade **ORDER BY** maior, primeiro você deve ver se você pode fazer que o MySQL use índices em vez de fazer um fase de ordenação extra. Se não for possível, então você pode fazer:

- Aumente o tamanho da variável **sort_buffer_size**.
- Aumente o temenho da variável **read_rnd_buffer_size**.
- Altere **tmpdir** para apontar para um disco dedicado com muito espaço vazio. Se você usa o MySQL 4.1 ou posterior você pode distribuir a carga entre diversos discos físicos definindo **tmpdir** com uma lista de caminhos separados por dois pontos **:** (ponto e vírgula **;** no Windows). Eles serão usados de acordo com o método round-robin. ***Nota:*** Estes caminho devem estar em diferentes discos ***físicos***, e não em diferentes partições do mesmo disco.

Por padrão, o MySQL ordena todas as consultas **GROUP BY x,y[,...]** como se você tivesse especificado **ORDER BY x,y[,...]**. Se você incluir a cláusula **ORDER BY** explicitamente, o MySQL a otimizará sem qualquer penalidade na velocidade, embora a ordenacao ainda ocorra. Se a consulta inclui um **GROUP BY** mas você deseja evitar a sobrecarga da ordenar o resultado, você pode suprimir a ordenacao especificando **ORDER BY NULL**:

```
INSERT INTO foo SELECT a,COUNT(*) FROM bar GROUP BY a ORDER BY NULL;
```

2.9. Como o MySQL Otimiza Cláusulas LIMIT

Em alguns casos o MySQL irá tratar a consulta de maneira diferente quando você estiver utilizando **LIMIT row_count** e não estiver utilizando **HAVING**:

- Se você estiver selecionando apenas alguns registros com **LIMIT**, o MySQL usará índices em alguns casos quando ele normalmente preferiria fazer uma varredura completa na tabela.
- Se você utilizar **LIMIT row_count** com **ORDER BY**, O MySQL irá terminar a ordenação logo que ele encontrar os primeiros **row_count** registros em vez de ordenar a tabela inteira.
- Ao combinar **LIMIT row_count** com **DISTINCT**, o MySQL irá parar logo que ele encontrar **row_count** registros únicos.
- Em alguns casos um **GROUP BY** pode ser resolvido lendo a chave em ordem (ou fazer uma classificação na chave) e então calcular resumos até o valor da chave alterar. Neste caso, **LIMIT row_count** não irá calcular nenhum **GROUP BY** desnecessário.
- Logo que o MySQL enviar os primeiros **#** registros para o cliente, ele irá abortar a consulta.
- **LIMIT 0** irá sempre retornar rapidamente um conjunto vazio. Isto é util para conferir a consulta e obter os tipos de campos do resultado.
- Quando o servidor utiliza tabelas temporárias para resolver a consulta, o **LIMIT row_count** é usado para calcular a quantidade de espaço necessário.

2.10. Performance das Consultas que Utilizam INSERT

O tempo para inserir um registro consiste aproximadamente de:

- Conexão: (3)
- Enviar a consulta para o servidor: (2)
- Analisar a consulta (2)
- Inserir o registro: (1 x tamanho do registro)
- Inserir os índices: (1 x número de índices)
- Fechar: (1)

onde os números são de certa forma proporcionais ao tempo total. Isto não leva em consideracão o sobrecarga inicial para abrir tabelas (que é feita uma vez para cada consulta concorrente em execução).

O tamanho da tabela diminuem a velocidade da inserção de índices em N log N (Arvores B).

Algumas maneiras de acelerar as inserções:

- Se você estiver inserindo vários registros do mesmo cliente ao mesmo tempo, utilize instruções **INSERT** com listas de múltiplos valores. Isto é muito mais rápido (muitas vezes em alguns casos) do que utilizar instruções **INSERT** separadas. Se você esta adicionando dados a uma tabela que não está vazia, você pode ajustar a variável **bulk_insert_buffer_size** para tornár isto mais rápido.
- Se você inserir vários registros de diferentes clientes, você pode obter velocidades mais altas utilizando a instrução **INSERT DELAYED**.
- Perceba que com **MyISAM** você pode inserir registros ao mesmo tempo que **SELECT**s estejam executando se não existirem registros apagados nas tabelas.
- Ao carregar uma tabela de um arquivo texto, utilize **LOAD DATA INFILE**. Isto é normalmente 20 vezes mais rápido do que utilizar várias instruções **INSERT**.
- É possível com algum trabalho extra fazer o **LOAD DATA INFILE** executar ainda mais rápido quando a tabela tiver vários índices. Utilize o seguinte procedimento:
    a. Opcionalmente crie a tabela com **CREATE TABLE**. Por exemplo, utilizando **mysql** ou Perl-DBI.
    b. Execute a instrução **FLUSH TABLES** ou o comando shell **mysqladmin flush-tables**.
    c. Utilize **myisamchk --keys-used=0 -rq /path/to/db/nome_tabela**. Isto removerá o uso de todos os índices da tabela.
    d. Insira dados na tabela com **LOAD DATA INFILE**. Isto não atualizará índices e será muito mais rápido.
    e. Se no futuro você precisar da tabela somente para leitura, execute **myisampack** na mesma para torná-la menor.
    f. Recrie os índices com **myisamchk -r -q /caminho/para/bd/nome_tabela**. Isto criará a árvore de índices em memória antes de escrevê-la para o disco, que é muito mais rápido porque evita que seja feita muita busca disco. A árvore de índices resultante é também balanceada perfeitamente.
    g. Execute uma instrução **FLUSH TABLES** ou o comando shell **mysqladmin flush-tables**.

Note que **LOAD DATA INFILE** també faz a otimização acima se você a inserção for em uma tabela vazia; a principal diferença com o procedimento acima é qeu você pode deixar o **myisamchk** alocar muita mais memória temporária para a criação do índice que você deseje que o MySQL alocasse para todas as recriações de índice.

Desde o MySQL 4.0 você também pode usar **ALTER TABLE nome_tbl DISABLE KEYS** em vez de **myisamchk --keys-used=0 -rq /caminho/para/bd/nome_tbl** e **ALTER TABLE nome_tbl ENABLE KEYS** em vez

de **myisamchk -r -q /caminho/para/bd/nome_tbl**. Deste modo você também pode saltar os passos **FLUSH TABLES**.

- Você pode acelerar inserções feitas usando várias instruções bloqueando suas tabelas:

```
mysql> LOCK TABLES a WRITE;
mysql> INSERT INTO a VALUES (1,23),(2,34),(4,33);
mysql> INSERT INTO a VALUES (8,26),(6,29);
mysql> UNLOCK TABLES;
```

A principal diferença na velocidade é que o buffer de índices é descarregado no disco somente uma vez, depois de todas instruções **INSERT** term sido completadas. Normalmente existiria tantas descargas do buffer de índices quanto instruções **INSERT** diferentes. O bloqueio não é necessário se você pode inserir todos registros com uma simples instrução.

Para tabelas transacionais, você deve usar **BEGIN/COMMIT** em vez de **LOCK TABLES** para conseguir um aumento na velocidade.

O bloqueio irá também diminuir o tempo total de testes de multi-conexões, mas o tempo máximo de espera para algumas threads irá aumentar (porque eles esperam pelos bloqueios). Por exemplo:

```
thread 1 faz 1000 inserções
thread 2, 3 e 4 faz 1 inserção
thread 5 faz 1000 inserções
```

Se você não estiver usando travas, 2, 3 e 4 irão terminar antes de 1 e 5, Se estiver utilizando travas, 2, 3 e 4 provavelmente não irão terminar antes de 1 ou 5, mas o tempo total deve ser cerca de 40% mais rápido.

Como as operações **INSERT**, **UPDATE** e **DELETE** são muito rápidas no MySQL, você obterá melhor perfomance geral adicionando travas em tudo que fizer mais que cerca de 5 inserções ou atualizações em um registro. Se você fizer várias inserções em um registro, você pode utilizar **LOCK TABLES** seguido de um **UNLOCK TABLES** de vez em quando (em torno de 1000 registro) para permitr que outras threads acessem a tabela. Isto também continua mostrando um bom ganho de performance.

Com certeza, **LOAD DATA INFILE** é muito mais rápido para carregar dados.

Para obter mais velocidade para **LOAD DATA INFILE** e **INSERT**, aumente o tamanho do buffer de chaves.

2.11. Performance das Consultas que Utilizam UPDATE

Consultas de atualização são otimizadas como uma consulta que usa **SELECT** com a sobrecarga adicional de escrita. A velocida da escrita depende do tamanho dos dados e do número de índices que serão atualizados. Índices que não forem alterados não serão atualizados.

Outra forma para obter atualizações rápidas é atrasar as atualizações e então fazer várias atualizações em um registro posteriormente. Fazer várias atualizações em um registro é muito mais rápido do que fazer uma por vez se você travar a tabela.

Perceba que, com formato de registros dinâmicos, atualizar um registro para um valor maior que o tamanho total pode dividir o registro. Portanto, se você faz isso frequentemente, é muito importante usar **OPTIMZE TABLE** de vez em quando.

2.12. Performance das Consultas que Utilizam DELETE

Se você deseja apagar todos os registros em uma tabela, deve usar **TRUNCATE TABLE nome_tabela**.

O tempo para apagar um registro é exatamente proporcional ao número de índices. Para apagar registros mais rapidamente, você pode aumentar o tamanho do cache de índices.

2.13. Mais Dicas sobre Otimizações

Dicas não ordenadas para sistemas rápidos:

- Utilize conexões persistentes aos banco de dados para evitar a sobrecarga da conexão. Se você não poder utilizar conexões persistentes e for fazer várias novas conexões para o banco de dados, você pode desejar alterar o valor da variável **thread_cache_size**.
- Sempre verifique se todas as suas consultas realmente utilizam os índices que foram criados nas tabelas. No MySQL você pode fazer isto com o comando **EXPLAIN**. See Explain: (manual) Explain.
- Tente evitar consultas **SELECT** complexas em tabelas que são muito atualizadas. Isto evita problemas com travamento de tabelas.
- Com tabelas **MyISAM** que não tenham linhas deletadas, você pode inserir registros ao mesmo tempo que outra tabela a estiver lendo. Se este recurso é importante para você, deve considerar métodos onde você não tem que apagar registrou ou executar **OPTIMIZE TABLE** depois de ter apagado vários registros.

- Utilize **ALTER TABLE ... ORDER BY expr1,expr2...** se você na maioria das vezes recupera registros na ordem expr1,expr2... Utilizando esta opção depois de grandes alterações para a tabela, pode lhe dar um ganho de performance.
- Em alguns casos pode fazer sentido introduzir uma coluna 'hash' baseada nas informações das outras colunas. Se esta coluna for curta e razoavelmente única pode ser muito mais rápido do que ter um grande índice em várias colunas. No MySQL é muito fácil usar esta coluna extra: **SELECT * FROM nome_tabela WHERE hash=MD5(concat(col1,col2)) AND col_1='constante' AND col_2='constante'**
- Para tabelas que alteram muito você deve tentar evitar todas colunas **VARCHAR** ou **BLOB**. Você terá tamanho de registro dinâmico assim que usar um simples campo **VARCHAR** ou **BLOB**.
- Normalmente não é muito útil cortar uma tabela em diferentes tabelas apenas porque os registros estão 'grandes'. Para acessar um registro, o maior problema para a performance é a busca em disco para encontra o primeiro byte do registro. Depois de encontrar os dados a maioria dos novos discos podem ler o registro inteiro rápido o bastante para a maioria das aplicações. Os únicos caos onde realmente faz sentido dividir uma tabela é se ela é uma tabela de registros com tamanho dinâmico (veja acima) que você pode alterar para um tamanho fixo, ou se você frequentemente precisa examinar a tabela e não precisa da maioria das colunas.
- Se frequentemente você precisar calcular alguma coisa baseada em informação de vários registros (ex: contagem de registros), provavlmente é melhor introduzir uma nova tabela e atualizar o contador em tempo real. Uma atualização do tipo **UPDATE table set count=count+1 where index_column=constante** é muito rapida!

  Isto é realmente importante quando você usa bancos de dados como o MySQL que só tem travamento de tabelas (multiplos leituras/escrita única). Isto também dará melhor performance com a maioria dos banco de dados, já que o gerenciador de bloqueio de registro terá menos a fazer neste caso.

- Se você precisar colerar estatisicas de tabelas maiores, utilize tabelas resumo em vez de buscar em toda a tabela. Manter os resumos deve ser mais rápido que tentar criar estatitíscas instantaneamente. É muito mais rápido criar novas tabelas através dos logs quando as coisas mudam (dependendo das descisões de negócio) que ter que alterar a aplicação em execução.
- Se possível, deve-se classificar relatórios como 'instantâneo' ou 'estatísticos' onde os dados necessários para relatórios estaiísticos são gerados apenas com base nas tabelas resumo que são geradas a partir dos dados atuais.
- Tire vantagem do fato de que a coluna tem valores padrões. Insira valores explicitamente apenas quando os valores a serem inseridos diferem do padrão. Isto reduz a analise que o MySQL precisa fazer e aumenta a velocidade de inserção.
- Em alguns casos é conveniente empacotar e armazenar os dados em um campo blob. Neste caso você deve adicionar algum código em sua aplicação para empacotar/desempacotar as coisas no campo blob, mas isto pode poupar vários acessos a algum estágio. Isto é prático quando você possui dados que não conformam com uma estrutura estática de tabela.

- Normalmente, você deve tentar manter todos dados não-redundantes (o que é chamado de 3a forma normal na teoria de bancos de dados), mas você não deve ter medo de duplicar alguns itens ou criar tabelas de resumo se você precisar delas para ganhar mais velocidade.
- Stored Procedures ou UDF (funções definidas pelo usuários) pode ser uma boa forma para obter mais performance. Neste caso você deve, entretanto, sempre ter uma maneira de fazer isso de outra maneira (mais lenta) se você utilizar algum banco de dados que não suporta isto.
- Você sempr pode ganhar velocidade fazendo cache de perguntas/respostas na sua aplicação e tentando fazer várias inserções/atualizações ao mesmo tempo. Se seu banco de dados suporta travamento de tabelas (como o MySQL e Oracle), isto deve ajudar a garantir que o cache de índices é descarregado somente uma vez depois de todas atualizações.
- Use **INSERT /*! DELAYED */** quando não precisar saber quando os dados são gravados. Isto melhora a velocidade porque vários registros podem ser gravados com uma simples escrita em disco.
- Use **INSERT /*! LOW_PRIORITY */** quando você desejar que suas consultas sejam mais importantes.
- Use **SELECT /*! HIGH_PRIORITY */** para obter consultas que ignoram a fila. Isto é, a consulta é feita mesmo se alguem estiver esperando para fazer uma escrita.
- Use a instrução **INSERT** multi-linhas para armazenar vários registros com um comando SQL (vários servidores SQL suportam isto).
- Use **LOAD DATA INFILE** para carregar volumes maiores de dados. Isto é mais rápido que as inserções normais e mais rápido até quando o **myisamchk** for integrado no **mysqld**.
- Use colunas **AUTO_INCREMENT** para garantir valores únicos.
- Use **OPTIMIZE TABLE** de vez em quando para evitar fragmentação quando estiver usando formatos de tabela dinâmica.
- Use tabelas **HEAP** para obter mais velocidade sempre que possível.
- Quando estiver usando uma configuração de servidor Web normal, imagens devem ser armazenadas como arquivos. Isto é, armazene apenas uma referência para o arquivo no banco de dados. A principal razão para isto é que um servidor Web normal é muito melhor trabalhando com cache de arquivos do que com conteúdo de banco de dados. Portanto será muito mais fácil obter um sistema rápido se você utilizar arquivos.
- Use tabelas em memória para dados não-críticos que são acessados frequentemente (como informações sobre o último banner visto para usuários que não possuem cookies).
- Colunas com informações identicas em diferentes tabelas devem ser declaradas idênticas e ter nomes idênticos. No entanto, antes da versão 3.23, você pode obter ligações mais lentas.

  Tente manter os nomes mais simples (use **nome** em vez de **nome_cliente** na tabela cliente). Para deixar seus nomes portáveis para outros servidores SQL você deve mantê-los menores que 18 caracteres.

- Se você realmente precisa de alta velocidade, você deve verificar as interfaces de baixo nível para armazenagem de dados que os diferentes servidores SQL suportam! Por exemplo, para acessar tabelas MySQL **MyISAM** diretamente,

você pode obter um aumento de velocidade de 2-5 vezes comparado ao uso da interface SQL. Para conseguir essa façanha, os dados devem estar no mesmo servidor que sua aplicação, e normalmente devem ser acessados por apenas um processo (porque travamento de arquivos externo são muito lentos). Os problemas acima podem ser eliminados introduzindo comandos **MyISAM** de baixo nível no servidor MySQL (isto pode ser a maneira mais fácil para aumentar a performance). Tenha cuidado em projetar a interface com o banco de dados, ela deve ser bem facil para suportar estes tipos de otimizações.

- Em vários casos é mais rápido acessar dados de um banco de dados (utilizando uma conexão ativa) do que acessar um arquivo texto, apenas pelo fato do banco de dados ser mais compacto do que o arquivo texto (se você estiver utilizando dados numéricos), e isto irá envolver menos acessos à disco. Você também irá poupar código porque não será necessário analisar seus arquivos texto para encontrar limites de registros e campos.
- Você pode também usar replicação para conseguir ainda mais performance nas suas aplicações.
- Declarando uma tabela com **DELAY_KEY_WRITE=1** irá tornar a atualização de índices mais rápida, pois as mesmas não serão escritas em disco até o arquivo ser fechado. O lado ruim é que você deve executar **myisamchk** nestas tabelas antes de iniciar o **mysqld** para garantir que os dados estão corretos se o **mysqld** for finalizado no meio da execução. Como a informação de chave pode sempre ser gerada a partir dos dados, você não deve perder nada usando **DELAY_KEY_WRITE**.

3. Detalhes sobre Locks

3.1. Como o MySQL Trava as Tabelas

Você pode encontrar uma discussão sobre diferentes métodos de bloqueios no apêndice.

Todos os bloqueios no MySQL são livres de deadlock, exceto para tipos de tabela **InnoDB** e **BDB**. Isto é gerenciado sempre requisitando todos os bloqueios necessários de uma vez no começo de uma consulta e sempre bloqueando as tabelas na mesma ordem.

Tipos de tabela **InnoDB** automaticamente adquire seus locks de registro e os tipos de tabela **BDB** seus locks de páginas, durante o processamento das instruções SQL, e não no início da transação.

O método de bloqueio que o MySQL utiliza para **ESCRITA** funciona da seguinte forma:

- Se não existirem travas na tabela, coloque uma bloqueio de escrita na mesma.
- Caso contrário, coloca a requisição de trava na fila de bloqueios para escrita.

O método de bloqueio que o MySQL utilizado para **LEITURA** funciona da seguinte maneira:

- Se não existirem tarvas na tabela, coloca um bloqueio de leitura na mesma.
- Caso contrário, coloca a requisição de trava na fila de bloqueios para leitura.

Quando um bloqueio é liberado, a trava fica disponível para as threads na fila de bloqueios de escrita, e então para as threads na fila de bloqueios de leitura.

Isto significa que se você possui várias atualizações em uma tabela, instruções **SELECT** irão esperar até que não existam mais atualizações.

Para contornar este problema no caso onde você precisa fazer várias operações de **INSERT** e **SELECT** em uma tabela, você pode inserir registros em uma tabela temporária e atualizar a tabela real com os registros da tabela temporária de uma só vez.

Isto pode ser feito usando o código a seguir:

```
mysql> LOCK TABLES real_table WRITE, insert_table WRITE;
mysql> INSERT INTO real_table SELECT * FROM insert_table;
mysql> TRUNCATE TABLE insert_table;
mysql> UNLOCK TABLES;
```

Você pode utilizar as opções **LOW_PRIORITY** com **INSERT**, **UPDATE** ou **DELETE** ou **HIGH_PRIORITY** com **SELECT** se você desejar priorizar a recuperação em alguns casos específicos. Também podei-se iniciar o **mysqld** com --low-priority-updates para obter o mesmo comportamento.

Utilizar **SQL_BUFFER_RESULT** pode também tornar a criação de locks de tabelas mais curtos.

Você também pode alterar o código de bloqueioss no mysys/thr_lock.c para usar uma fila simples. Neste caso, bloqueios de escrita e leitura devem ter a mesma prioridade, o que pode ajudar em algumas aplicações.

3.2. Detalhes sobre Lock de Tabelas

O código de bloqueio de tabelas no MySQL é livre de deadlock.

O MySQL utiliza bloqueio de tabelas (no lugar de bloqueio de registros ou colnas) em todos os tipos de tabelas, exceto tabelas **BDB**, para obter uma alta velocidade nos bloqueios. Para grandes tabelas, bloqueio de tabelas é MUITO melhor que bloqueio de registros para a maioria das aplicações, mas existem, é claro, algumas desvantagens.

Para tabelas **BDB** e **InnoDB**, O MySQL só utiliza bloqueio de tabelas se você bloquear explicitamente a tabela com **LOCK TABLES** ou executar um comando quer irá modificar todos os registros na tabela, como **ALTER TABLE**. Para estes tipos de tabelas nós recomendamos a você não utilizar **LOCK TABLES**.

No MySQL versão 3.23.7 ou superior , você pode inserir registros em tabelas **MyISAM** ao mesmo tempo que outras threads estão lendo da mesma tabela. Perceba que atualmente isto funciona somente se não existirem buracos depois de registros apagados na tabela no momento que a inserção é feita. Quando todos os buracos forem preenchidos com novos dados, inserções concorrentes irão automaticamente ser habilitadas novamente.

O bloqueio de tabelas habilita várias threads para lerem de uma tabela ao mesmo tempo, mas se uma thread desejar escrever a uma tabela, ela primeiramente deve obter acesso exclusivo. Durante a atualização, todas outras threads que desejarem acessar esta tabela em particular irão esperar até que a atualização acabe.

Como atualizações em tabelas normalmente são consideradas mais importantes que **SELECT**, todas as instruções que atualizam uma tabela tem maior prioridade que instruções que simplesmente recuperam informações. Isto deve garantir que atualizações não fiquem na fila por terem sido passadas várias consultas pesadas em uma tabela específica. (Você pode alterar isto utilizando LOW_PRIORITY com a instrução que faz a atualização ou **HIGH_PRIORITY** com a instrução **SELECT**.)

A partir do MySQL versão 3.23.7 pode-se utilizadar a variável **max_write_lock_count** para forçar o MySQL a fornecer temporariamente a todas as instruções **SELECT**, que esperam por uma tabela, uma prioridade mais alta depois de um número específico de inserções em uma tabela.

O bloqueio de tabela não é, no entanto, muito bom sobre os seguintes cenários:

- Um cliente emite uma **SELECT** que exige muito tempo para ser executada.
- Outro cliente então executa um **UPDATE** na tabela usada. Este cliente terá que esperar até que a **SELECT** seja terminada.
- Outro cliente executa outra instrução **SELECT** na mesma tabela. Como **UPDATE** tem maior prioridade que **SELECT**, esta **SELECT** irá esperar pelo término da **UPDATE**. Ela também irá esperar pelo término da primeira **SELECT**!
- Uma thread está esperando por algo do tipo **disco cheio**, caso em que todas as threads que desejam acessar a tabela com problema irão ser colocadas em estado de espera até que mais espaço em disco seja disponível.

Algumas soluções possíveis para este problema são:

- Tente deixar suas instruções **SELECT** sempre rápidas. Você pode ter que criar algumas tabelas de resumo para fazer isto.
- Inicie o **mysqld** com --low-priority-updates. Isto irá fornecer a todas instruções que atualizam (modificam) uma tabela prioridade menor que uma instrução **SELECT**. Neste caso a última instrução **SELECT** no cenário anterior deveria executar antes da instrução **INSERT**.

  Você pode fornecer a uma instrução **INSERT**, **UPDATE** ou **DELETE** específica menor prioridade com o atributo **LOW_PRIORITY**.

- Inicie o **mysqld** com um valor baixo para ***max_write_lock_count*** para fornecer bloqueios de **LEITURA** depois de um certo número de bloqueios de **ESCRITA**.
- Você pode especificar que todas as atualizações de uma thread específica deve ser feita utilizando prioridade baixa com o comando SQL: **SET SQL_LOW_PRIORITY_UPDATES=1**.
- Você pode especificar que uma **SELECT** específica é muito importante com o atributo **HIGH_PRIORITY**.
- Se você tiver problemas com **INSERT** combinado com **SELECT**, utilize as novas tabelas **MyISAM**, pois elas suportam **SELECT**s e **INSERT**s concorrentes.
- Se você utiliza principalmente instruções **INSERT** e **SELECT** misturadas, o atributo **DELAYED** no **INSERT** provavelmente irá resolver seus problemas.
- Se você tiver problemas com **SELECT** e **DELETE**, a opção **LIMIT** para **DELETE** pode ajudar.

4. Otimizando a Estrutura de Banco de Dados

4.1. Opções do Projeto

O MySQL mantem dados de registros e índices em arquivos separados. Vários (quase todos) bancos de dados misturam dados de registros e índice no mesmo arquivo. Nós acreditamos que a escolha do MySQL é melhor para uma ampla escala de sistemas modernos.

Outra forma de armazenar os dados de registros é manter a informação para cada coluna em uma área separada (exemplos são o SDBM e o Focus). Isto irá causar um ponto de performance para toda consulta que acessar mais de uma coluna. Como isto degrada rapidamente quando mais de uma coluna é acessada, acreditamos que este modelo não é bom para propósitos gerais de bancos de dados.

O caso mais comum é aquele em que o índice e dados são armazenados juntos (como no Oracle/Sybase). Neste caso você irá encontrar a informação do registro na folha da página de índice. A coisa boa com este layout é que ele, em vários casos, dependendo de como o índice é armazenado no cache, salva uma leitura de disco. As desvantagens deste layout são:

- A varredura da tabela é muito mais lenta porque você tem que ler os índices para encontrar os dados.
- Não podem ser usados apenas a tabela de índices para recuperar dados para uma consulta.
- Você perde muito espaço de armazenagem, já que que os índices devem ser duplicados nos nós (pois os registros não podem ser armazenados nos nós).
- Deleções irão degenerar a tabela depois de um tempo (já que os índices nos nós normalmente não são atualizados na deleção).
- É mais difícil fazer o cache somente dos dados de índices.

4.2. Deixando os Dados com o Menor Tamanho Possível

Uma das otimizações mais básicas é tentar manter seus dados (e índices) utilizando o menor espaço possível no disco (e em memória). Isto pode fornecer grandes

melhorias porque a leitura de disco é mais rápida e normalmente menos memória principal será usada. A indexação também exige menos recursos se for feita em colunas menores.

O MySQL suporta vários diferentes tipos de tabelas e formatos de registros. Você pode ter um ótimo ganho de performance escolhendo o formato certo de tabela a ser usada.

Pode-se obter melhor performance em uma tabela e minimizar espaço de armazenagem utilizando as técnicas listadas abaixo:

- Utilize os tipos mais eficientes (menores) sempre que possível. O MySQL tem vários tipos especializados que economizam espaço em disco e memória.
- Utilize tipos inteiros menores se possível para obter tabelas menores. Por exemplo, **MEDIUMINT** normalmente é melhor que **INT**.
- Declare colunas para serem **NOT NULL** se possível. Isto deixa tudo mais rápido e você economiza um bit por coluna. Perceba que se você realmente precisa de **NULL** nas suas aplicações, podem ser usados. Tente simplesmente não usá-la em todas as colunas por padrão.
- Se você não possui nenhuma coluna de tamanho variável (**VARCHAR**, **TEXT** ou **BLOB**), um formato de registro de tamanho fixo para é utilizado. Isto é mais rápido mas infelizmente pode ocupar mais espaço.
- O índice primário de uma tabela deve ser o mais curto possível. Isto torna a identificação de um registro fácil e eficiente.
- Para cada tabela, você deve decidir qual metódo de armazenamento/índice utilizar.
- Crie somente os índices necessários. Índices são bons para recuperação mas ruins quando você precisa armazenar os dados rapidamente. Se na maioria das vezes você acessa uma tabela pesquisando em uma combinação de colunas, crie um índice para elas. A primeira parte do índice deve ser a coluna mais utilizada. Se você SEMPRE utiliza várias colunas, deve usar a coluna com mais duplicações em primeiro lugar para obter melhor compactação do índice.
- Se for melhor que uma coluna tenha um prefixo único nos primeiros caracteres, é melhor indexar somente este prefixo. O MySQL suporta um índice em uma parte de uma coluna de caracteres. Índices menores são mais rápidos não somente porque eles exigem menos espaço em disco mas também porque eles irão fornecer a você mais acerto no cache de índice e isto diminui acessos a disco.
- Em algumas circunstâncias pode ser benéfico dividir uma tabela que é varrida frequentemente em duas. Isto é verdade especificamente se a tabela tiver um formato dinâmico e for possível utilizar um formato de tabela estático que possa ser usada para encontrar os registros relevantes quando se fizer uma varredura da tabela.

4.3. Como o MySQL Utiliza Índices

Os índices são utilizados para encontrar registros com um valor específico de uma coluna rapidamente. Sem um índice o MySQL tem de iniciar com o primeiro registro e depois ler através de toda a tabela até que ele encontre os registros relevantes. Quanto maior a tabela, maior será o custo. Se a tabela possui um índice para as colunas em questão, o MySQL pode rapidamente obter uma posição para procurar no meio do arquivo de dados sem ter que varrer todos os registros. Se uma tabela possui 1000 registros, isto é pelo menos 100 vezes mais rápido do que ler todos os registros sequencialmente. Note que se você precisar acessar quase todos os 1000 registros, seria mais rápido acessá-los sequencialmente porque evitaria acessos ao disco.

Todos os índices do MySQL (**PRIMARY**, **UNIQUE** e **INDEX**) são armazenados em árvores B. Strings são automaticamente compactadas nos espaços finais e prefixados.

Índices são utilizados nos seguintes modos:

- Para encontrar rapidamente os registros que coincidam com uma cláusula **WHERE**.
- Para recuperar registros de outras tabelas ao realizar joins.
- Para encontrar o valor **MAX()** ou **MIN()** para uma coluna indexada espeifica. Isto é otimizado por um preprocessador que confere se você está utilizando **WHERE** key_part_#=constante em todas as partes da chave < N. Neste caso o MySQL irá fazer uma simples procura na chave e trocar a expressão **MIN()** com uma constante. Se todas as expressões forem trocadas por constantes, a consulta retornará imediatamente:

```
SELECT MIN(key_part2),MAX(key_part2) FROM nome_tabela where
key_part1=10
```

- Para ordenar ou agrupar uma tabela se a ordenação ou agrupamento for feito em um prefixo mais à esquerda de uma chave util (por exemplo, **ORDER BY key_part_1, key_part_2**). A chave é lida na ordem invertida se todas as partes da chave forem seguidas por **DESC**.
- Em alguns casos uma consulta pode ser otimizada para recuperar valores sem consultar o arquivo de dados. Se todas colunas utilizadas para alguma tabela são numéricas e formam um prefixo mais à esquerda para alguma chave, os valores podem ser recuperados da árvore de índices para aumentar a velocidade:

```
SELECT key_part3 FROM nome_tabela WHERE key_part1=1
```

Suponha que você utilize a seguinte instrução **SELECT**:

```
mysql> SELECT * FROM nome_tabela WHERE col1=val1 AND col2=val2;
```

Se um índice de colunas múltiplas existir em **col1** e **col2**, os registros apropriados podem ser recuperados diretamente. Se índices separados de únicas colunas existirem em **col1** e **col2**, o otimizador tentará encontrar o índice mais restritivo

decidindo qual índice irá encontrar menos registros e usará este índice para recuperar os registros.

Se a tabela possuir um índice de múltiplas colunas, qualquer prefixo mais à esquerda do índice pode ser usado pelo otimizador para encontrar registros. Por exemplo, se você possui um índice de três colunas em **(col1, col2, col3)**, você tem capacidades de busca indexada em **(col1)**, **(col1, col2)** e **(col1, col2, col3)**.

O MySQL não pode utilizar um índice parcial se as colunas não formarem um prefixo mais à esquerda do índice. Suponha que você tenha as instruções **SELECT** mostradas abaixo:

```
mysql> SELECT * FROM nome_tabela WHERE col1=val1;
mysql> SELECT * FROM nome_tabela WHERE col2=val2;
mysql> SELECT * FROM nome_tabela WHERE col2=val2 AND col3=val3;
```

Se um índice existir em **(col1, col2, col3)**, somente a primeira consulta anteriores utiliza o índice. A segunda e terceira consultas involvem colunas indexadas, mas **(col2)** e **(col2, col3)** não são os prefixos mais à esquerda de **(col1, col2, col3)**.

O MySQL também utiliza índices para comparações do tipo **LIKE** se o argumento para **LIKE** for uma string constante que não inicie com um meta caracter Por exemplo as seguintes instruções **SELECT** utilizam índices:

```
mysql> SELECT * FROM nome_tbl WHERE key_col LIKE "Patrick%";
mysql> SELECT * FROM nome_tbl WHERE key_col LIKE "Pat%_ck%";
```

Na primeira instrução, somente os registros com **"Patrick" <= key_col < "Patricl"** são considerados. Na segunda instrução, somente registros com **"Pat" <= key_col < "Pau"** são considerados.

As seguintes instruções **SELECT** não usarão índices:

```
mysql> SELECT * FROM nome_tbl WHERE key_col LIKE "%Patrick%";
mysql> SELECT * FROM nome_tbl WHERE key_col LIKE other_col;
```

Na primeira instrução, o valor **LIKE** inicia com um meta caracter. Na segunda instrução, o valor **LIKE** não é uma constante.

O MySQL 4.0 faz outra otimização em **LIKE**. Se você usar **... LIKE "%string%"** e **string** tiver mais de 3 caracteres, o MySQL usará o algorítmo **Turbo Boyer-Moore** para inicializar o padrão para a string e então usar este padrão para realizar a pesquisa mais rápido.

Buscas usando **nome_coluna IS NULL** usa índices se **nome_coluna** é um índice.

O MySQL normalmente utiliza o índice que encontra o menor número de registros. Um índice é usado para colunas que você compara com os seguintes operadores: =, >, >=, <, <=, **BETWEEN** ou um **LIKE** com um padrão que começa com um prefixo sem meta caracteres como **'algo%'**.

Qualquer índice que não cobrem todos os níveis de **AND** na cláusula **WHERE** não é utilizado para otimizar a consulta. Em outras palavras: Para poder usar um índice, um prefixo do índice deve ser utilizado em todo agrupamento **AND**.

A seguinte cláusula **WHERE** utilizará índices:

```
... WHERE index_part1=1 AND index_part2=2 AND other_column=3
... WHERE index=1 OR A=10 AND index=2      /* index = 1 OR index = 2
*/
... WHERE index_part1='hello' AND index_part_3=5
          /* optimised like "index_part1='hello'" */
... WHERE index1=1 AND index2=2 OR index1=3 AND index3=3;
          /* Can use index on index1 but not on index2 or index 3 */
```

Estas cláusulas **WHERE** ***não*** utilizam índices:

```
... WHERE index_part2=1 AND index_part3=2  /* index_part_1 is not used
*/
... WHERE index=1 OR A=10                  /* Index is not used in
                                                        both AND parts
*/
... WHERE index_part1=1 OR index_part2=10  /* No index spans all rows
*/
```

Perceba que algumas vezes o MySQL não utilizará um índice, mesmo se algum estiver disponível. Um exemplo deste caso é quando o uso do índice necessita que o MySQL acesse mais de 30% dos registros na tabela. (Neste caso uma varredura da tabela é provavelmente mais rápido, já que ela necessitará de menos pesquisas em discos). No entanto, se uma consulta utiliza **LIMIT** para recuperar somente parte dos registros, o MySQL irá utilizar um índice de qualquer forma, pois assim pode encontrar os poucos registros mais rapidamente e retornar o resultado.

4.4. Índices de Colunas

Todos os tipos de colunas do MySQL podem ser indexadas. O uso de índices nas colunas relevantes é a melhor forma de melhorar a performance de operações **SELECT**.

O número máximo de índices por tabelas e o tamanho máximo de um índice é definido pelo mecanismo de armazenamento. Todos os mecanismos de armazenamentos suportam um mínimo de 16 chaves por tabela e um índice de tamanho total mínimo de 256 bytes.

Para colunas **CHAR** e **VARCHAR** você pode indexar um prefixo da coluna. Isto é muito mais rápido e necessita de menos espaço em disco do que indexar a coluna

inteira. A sintaxe para utilizar na instrução **CREATE TABLE** para indexar um prefixo de uma coluna se parece com o exemplo a seguir:

```
INDEX nome_indice (nome_campo(tamanho))
```

O exemplo abaixo cria um índice para os primeiros 10 caracteres da coluna **nome**:

```
mysql> CREATE TABLE teste (
           nome CHAR(200) NOT NULL,
           INDEX nome_indice (nome(10)));
```

Para colunas **BLOB** e **TEXT**, você deve indexar um prefixo da coluna. O índice pode ter até 255 bytes.

No MySQL Versão 3.23.23 ou posterior, você pode também criar índices ***FULLTEXT*** especiais. Eles são utilizados para pesquisas textuais. Somente o tipo de tabela **MyISAM** suporta índices **FULLTEXT** e apenas para colunas **CHAR**, **VARCHAR** e **TEXT**. Indexação sempre acontece sobre toda a coluna e indexação parcial (prefixo) não é suportada.

4.5. Índices de Múltiplas Colunas

O MySQL pode criar índices em múltiplas colunas. Um índice pode consistir de até 15 colunas. (Em colunas **CHAR** e **VARCHAR** você também pode utilizar um prefixo da coluna como parte de um índice).

Um índice de múltiplas colunas pode ser considerado um array ordenado contendo valores que são criados concatenando valores de colunas indexadas.

O MySQL utiliza índices de múltiplas colunas de forma que consultas são rápidas quando você especifica uma quantidade conhecida para a primeira coluna do índice em uma cláusula **WHERE**, mesmo se você não especificar valores para as outras colunas.

Suponha que uma tabela tenha a seguinte especificação:

```
mysql> CREATE TABLE teste (
           id INT NOT NULL,
           ultimo_nome CHAR(30) NOT NULL,
           primeiro_nome CHAR(30) NOT NULL,
           PRIMARY KEY (id),
           INDEX nome (ultimo_nome,primeiro_nome));
```

Então o índice **nome** é um índice com **ultimo_nome** e **primeiro_nome**. O índice será usado para consultas que especificarem valores em um limite conhecido para **ultimo_nome**, ou para ambos **ultimo_nome** e **primeiro_nome**. Desta forma, o índice **nome** será usado nas seguintes consultas:

```
mysql> SELECT * FROM teste WHERE ultimo_nome="Widenius";
```

```
mysql> SELECT * FROM teste WHERE ultimo_nome="Widenius"
                            AND primeiro_nome="Michael";

mysql> SELECT * FROM teste WHERE ultimo_nome="Widenius"
                           AND (primeiro_nome="Michael" OR
primeiro_nome="Monty");

mysql> SELECT * FROM teste WHERE ultimo_nome="Widenius"
                           AND primeiro_nome >="M" AND primeiro_nome <
"N";
```

Entretanto, o índice **nome *não*** será usado nas seguintes consultas:

```
mysql> SELECT * FROM teste WHERE primeiro_nome="Michael";

mysql> SELECT * FROM teste WHERE ultimo_nome="Widenius"
                           OR primeiro_nome="Michael";
```

4.6. Como o MySQL Conta as Tabelas Abertas

Ao executar o comando **mysqladmin status**, você verá algo deste tipo:

```
Uptime: 426 Running threads: 1 Questions: 11082 Reloads: 1 Open
tables: 12
```

O valor **Open tables** de 12 ode ser bastante estranho se você só possui 6 tabelas.

O MySQL é multithreaded, portanto ele pode haver clientes enviando consultas para uma determinada tabela simultaneamente. Para minimizar o problema com dois clientes tendo diferentes estados no mesmo arquivo, a tabela é aberta independentemente por cada thread concorrente. Isto exige mais memória mas normalmente aumentará o desempenho. Com tabelas **ISAM** e **MyISAM**, um descritor extra de arquivo é necessário para o arquivo de dados, para cada cliente que tem a tabela aberta. O descritor de arquivo de índice é compartilhado entre todas as threads.

4.7. Como o MySQL Abre e Fecha as Tabelas

As variáveis do servidor **table_cache**, **max_connections** e **max_tmp_tables** afetam o número máximo de arquivos que o servidor mantêm abertos. Se você aumentar um ou ambos destes valores, você pode ir contra um limite imposto pelo seu sistema operacional no número de arquivos abertos por processo. Você pode aumentar o limite de arquivos abertos em muitos sistemas operacionais, embora o método varia muito de um sistema para outro. Consulte a documentação de seu Sistema Operacional para saber como fazê-lo, porque o método para alterar o limite varia muito de um sistema para outro.

table_cache é relacionado a **max_connections**. Por exemplo, para 200 conexões concorrentes em execução, você deve ter um tamanho de cache de tabela de pelo menos **200 * n**, onde **n** é o número máximo de tabelas em um join. Você também precisa reservar alguns descritores de arquivos para tabelas e arquivos temporários.

Esteja certo de que o seu sistema operacional pode tratar o número de descritores de arquivos abertos definido pelo valor de **table_cache**. Se **table_cache** for muito alto, o MySQL pode esgotar os descritores de arquivo e recusar conexões, falhar na execução de consultas e ser muito instavel. Você também têm que levar em conta que o mecanismo de armazenamento **MyISAM** precisa de dois descritores de arquivos para cada tabela aberta. Você pode aumentar o número de descritores de arquivo disponíveis para o MySQL com a opção de inicialização --open-files-limit.

A cache de tabelas abertas será mantido em um nível de **table_cache** entradas. O valor padrão é 64; isto pode ser alterado com a opção **-O table_cache=#** do **mysqld**. Note que o MySQL pode temporariamente abrir mais tabelas para poder se executar consultas.

Um tabela não usada é fechada e removida da cache de tabelas sob as seguintes circuntâncias:

- Quando a cache está cheia e um thread tenta abrir uma tabela que não está na cache.
- Quando a cache contém mais que **table_cache** entradas e uma thread não está mais usando uma tabela.
- Quando alguém executa **mysqladmin refresh** ou **mysqladmin flush-tables**.
- Quando alguém executa uma instrução **FLUSH TABLES**.

Quando o cache de tabela encher, o servidor usa o seguinte procedimento para encontrar uma entrada de cache para usar:

- Tabelas que não estiverem em uso são liberadas, na ordem LRU (least-recently-used), ou seja, a tabela que foi usada menos rcentemente.
- Se o cache estiver cheio e nenhuma tabelas pode ser liberada, mas uma nova tabela precisar ser aberta, o cache é extendido temporariamente quando necessário.
- Se o cache estiver no estado temporariamente extendido e uma tabela vai do estado em-uso para o fora-de-uso, a tabela é fechada e liberada do cache.

A table is opened for each concurrent access. This means the table needs to be opened twice if two threads access the same table or if a thread accesses the table twice in the same query (for example, by joining the table to itself).

Uma tabela é aberta para cada acesso simultâneo. Isto significa a tabela precisa ser aberta duas vezes se duas threads acessam a mesma tabela ou se uma thread acessa a tabela duas vezes na mesma consulta (por exemplo, fazendo um join da tabela com ela mesma). A primeira abertura de qualquer tabela exige dois descritores de arquivos; cada uso adicional da tabela exige somente um descritor. O descritor extra para a primeira abertura é para o arquivo de índice: este descritor é compartilhado entre todas as threads.

Se você está abrindo uma tabela com a instrução **HANDLER nome_tabela OPEN**, uma tabela dedicada é alocada para a thread. Este objeto da tabela não é compartilhado por outras threads e não será fechado até que a thread chame **HANDLER nome_tabela CLOSE** ou seja finalizada. Quando isto acontece, a tabela é colocada de volta na cache de tabela (se a cache não estiver cheia).

Você pode conferir se o seu cache de tabela está muito pequeno conferindo a variável **opened_tables** do **mysqld**. Se este valor for muito grande, mesmo se você não fez vários **FLUSH TABLES**, você deve aumentar o tamanho da sua cache de tabelas.

4.8. Desvantagem em Criar um Número Grande de Tabelas no Mesmo Banco de Dados

Se você possui muitos arquivos em um diretório, operações de abrir, fechar e criação ficarão lentos. Se você executar instruções **SELECT** em diversas tabelas, existirá uma pequena sobrecarga quando o cache de tabela estiver cheio, porque para toda tabela que teve que ser aberta, outra deve ser fechada. Você pode reduzir esta sobrecarga tornando o cache de tabelas maior.

5. Otimizando o Servidor MySQL

5.1. Sintonia dos Parâmetros em Tempo de Sistema/Compilação e na Inicialização

Nós iniciamos com o fator do nível do sistema pois algumas destas decisões devem ser feitas bem cedo. Em outros casos uma rápida olhada para esta seção pode satisfazer porque ela não é tão importante para os grandes ganhos. Entretanto, é sempre bom ter ter noções de como você pode obter melhorias alterando coisas neste nível.

Qual sistema operacional a usar é realmente importante! Para obter o melhor uso de máquinas com múltiplas CPUs você deve utilizar Solaris (porque a sua implemetação das threads funcionam muito bem) ou Linux (porque o kernel 2.2 tem suporte SMP muito bom). Também, em Linux mais antigos temos o limite de tamanho de arquivo de 2G por padrão. Se você tem tal kernel e precisa desesperadamente de trabalhar com arquivos maiores que 2G em máquinas intel Linux, você deve obter o patch LFS para o sistema de arquivos ext2. Outros sistemas de arquivo como ReiserFS e XFS não possuem esta limitação de 2G.

Como ainda não temos o MySQL em produção em muitas outras plataformas, nós aconselhamos que você teste a plataforma pretendida antes de escolhe-la, se possível.

Outras dicas:

- Se você possui RAM suficiente, você pode remover todos os dispositivos de troca. Alguns sistemas operacionais irão utilizar um disposotico de troca em alguns contextos, mesmo se você possuir memória livre.
- Utilize a opção do MySQL --skip-external-locking para evitar locks externos. Perceba que isto não irá afetar a funcionalidade do MySQL se você estiver executando um único servidor. Apenas lembre-se de desligar o servidor (ou travar as partes relevantes) antes de executar **myisamchk**. Em alguns sistemas esta opção é obrigatório porque o lock externo não funcionam em nenhum caso.

  A opção --skip-external-locking está ligada por padrão a partir do MySQL 4.0. Antes disto, era ligada por padrão quando compilando com MIT-pthreads, porque **flock()** não é totalmente suportado pelas MIT-pthreads em todas plataformas. É também o padrão para Linux pois o bloqueio de arquivos no Linux não é muito seguro.

O único caso que você não pode utilizar --skip-external-locking é se você precisa de vários *servidores* MySQL (não clientes) acessando os mesmos dados, ou executar **myisamchk** na tabela sem dizer ao servidor para descarregar e travar as tabelas primeiro

Você pode continuar usando **LOCK TABLES**/**UNLOCK TABLES** mesmo se você estiver utilizando --skip-external-locking.

5.2. Parâmetros de Sintonia do Servidor

Você pode determinar tamanho padrão do buffer usados pelo servidor **mysqld** com este comando:

```
shell> mysqld --help
```

Este comando produz uma lista de todas as opções do **mysqld** e variáveis configuráveis. A saída inclui os valores padrão das variáveis e se parece com isto:

```
back_log                current value: 5
bdb_cache_size          current value: 1048540
binlog_cache_size       current value: 32768
connect_timeout         current value: 5
delayed_insert_timeout  current value: 300
delayed_insert_limit    current value: 100
delayed_queue_size      current value: 1000
flush_time              current value: 0
interactive_timeout     current value: 28800
join_buffer_size        current value: 131072
key_buffer_size         current value: 1048540
lower_case_nome_tabelas  current value: 0
long_query_time         current value: 10
max_allowed_packet      current value: 1048576
max_binlog_cache_size   current value: 4294967295
max_connections         current value: 100
max_connect_errors      current value: 10
max_delayed_threads     current value: 20
max_heap_table_size     current value: 16777216
max_join_size           current value: 4294967295
max_sort_length         current value: 1024
max_tmp_tables          current value: 32
max_write_lock_count    current value: 4294967295
myisam_sort_buffer_size current value: 8388608
net_buffer_length       current value: 16384
net_retry_count         current value: 10
net_read_timeout        current value: 30
net_write_timeout       current value: 60
read_buffer_size        current value: 131072
record_rnd_buffer_size  current value: 262144
slow_launch_time        current value: 2
sort_buffer             current value: 2097116
table_cache             current value: 64
thread_concurrency      current value: 10
tmp_table_size          current value: 1048576
thread_stack            current value: 131072
```

```
wait_timeout              current value: 28800
```

Se existir um servidor **mysqld** em execução, você pode ver quais valores ele está usando atualmente para as variáveis executando esta instrução:

```
mysql> SHOW VARIABLES;
```

Você também pode ver algumas estatísticas e indicadores de status para um servidor em execução executando este comando:

```
mysql> SHOW STATUS;
```

Para encontrar uma descrição completa de todas as variáveis na seção **SHOW VARIABLES** neste manual.

Variáveis de servidor e informação de status também pode ser obtido usando **mysqladmin**:

```
shell> mysqladmin variables
shell> mysqladmin extended-status
```

O MySQL utiliza algorítmos que são muito escaláveis, portanto, normalmente você pode trabalhar com pouca memória. Entretanto, se você fornecer ao MySQL mais memória, obterá um desempenho melhor.

Quando estiver ajustando um servidor MySQL, as duas variáveis mais importantes que devem ser usadas são **key_buffer_size** e **table_cache**. Você deve se sentir confiante que as duas estejam corretas antes de tentar alterar qualquer outra variável.

Os seguintes exemplos indicam alguns valores típicos de variáveis para diferentes configurações de tempo de execução. Os exemplos usam o script **mysqld_safe** e usam a sintaxe --name=value para definir a variável **name** com o valor **value**. Esta sintaxe está disponível a partir do MySQL 4.0. Para versões mais antigas do MySQL, tome as seguintes diferenças nas contas:

- Use **safe_mysqld** em vez de **mysqld_safe**.
- Configure as variáveis usando a sintaxe --set-variable=name=value ou **-O name=value**
- Para nomes de variáveis que finalizam em **_size**, você pode precisar especificá-las sem **_size**. Por exemplo, o nome antigo para **sort_buffer_size** é **sort_buffer**. O nome antigo para **read_buffer_size** é **record_buffer**. Para ver quais variáveis a versão do seu servidor reconhece, use **mysqld --help**.

Se você possui pelo menos 256M de memória e várias tabelas e deseja obter o melhor desempenho com um número moderado de clientes, deve utilizar algo como:

```
shell> mysqld_safe --key_buffer_size=64M --table_cache=256 \
           --sort_buffer_size=4M --read_buffer_size=1M &
```

Se possui apenas 128M de memória e apenas algumas poucas tabelas, mas ainda deseja realizar várias ordenações, você pode utilizar:

```
shell> mysqld_safe --key_buffer_size=16M --sort_buffer_size=1M
```

Se você possuir pouca memória e tiver muitas conexões, utilize algo como:

```
shell> mysqld_safe --key_buffer_size=512K --sort_buffer_size=100K \
           --read_buffer_size=100K &
```

ou mesmo isto:

```
shell> mysqld_safe --key_buffer_size=512K --sort_buffer_size=16K \
           --table_cache=32 --read_buffer_size=8K -O net_buffer_length=1K
&
```

Se você estiver executando um **GROUP BY** ou **ORDER BY** em tabelas que são muito maiores que sua memória disponível você deve aumentar o valor de **record_rnd_buffer_size** para acelerar a leitura de registros após a operação de ordenação.

Quando você tiver instalado o MySQL, o diretório support-files irá conter alguns arquivos exemplos do **my.cnf**, my-huge.cnf, my-large.cnf, my-medium.cnf e my-small.cnf, você pode usá-los como base para otimizar seu sistema.

Se você possui várias conexões simultâneas, ``problemas de trocas" podem ocorrer a menos que o **mysqld** tenha sido configurado para usar muito pouca memória para cada conexão. O **mysqld** tem melhor performance se você tiver memória suficiente para todas as conexões, é claro.

Perceba que se você especifica uma opção na linha de comando para o **mysqld**, ou **mysqld_safe** ele permanece em efeito somente para aquela chamada do servidor. Para usar a opção toda vez que o servidor executa, coloque-o em um arquivo de opção.

Para ver os efeitos de uma alteração de parâmetro, faça algo como:

```
shell> mysqld --key_buffer_size=32m --help
```

Tenha certeza que a opção --help seja a última do comando; de outra forma o efeito de qualquer opções listadas depois na linha de comando não serão refletidas na saída.

5.3. Como a Compilação e a Ligação Afetam a Velocidade do MySQL

A maioria dos testes seguintes são feitos no Linux com os benchmarks do MySQL, mas eles devem fornecer alguma indicação para outros sistemas operacionais e workloads.

Você obtêm um executável mais veloz quando ligado com **-static**.

No Linux, você irá obter o código mais rápido quando compilando com **pgcc** e **-03**. Para compilar sql_yacc.cc com estas opções, você precisa de cerca de 200M de memória porque o **gcc/pgcc** precisa de muita memória para criar todas as funções em linha. Também deve ser configurado o parâmetro **CXX=gcc** para evitar que a biblioteca **libstdc++** seja incluida (não é necessária). Perceba que com algumas versões do **pgcc**, o código resultante irá executar somente em verdadeiros processadores Pentium, mesmo que você utilize a opção do compilador para o código resultante que você quer, funcionando em todos os processadores do tipo x586 (como AMD).

Só pelo fato de utilizar um melhor compilador e/ou melhores opções do compilador você pode obter um aumento de desempenho de 10-30% na sua aplicação. Isto é particularmente importante se você mesmo compila o servidor SQL!

Nós testamos ambos os compiladores Cygnus Codefusion e o Fujitsu, mas quando os testamos, nenhum dos dois era suficientemente livre de erros para que o MySQL compilasse com as otimizações.

Quando você compila o MySQL deve incluir suporte somente para os conjuntos de caracteres que deseja usar. (Opção --with-charset=xxx). As distribuições binárias padrão do MySQL são compiladas com suporte para todos os conjuntos de caracteres.

Segue uma lista de algumas medidas que temos feito:

- Se você utiliza o **pgcc** e compila tudo com **-O6**, o servidor **mysqld** é 1% mais rápido do que com o **gcc** 2.95.2.
- Se você liga dinamicamente (sem **-static**), o resultado é 13% mais lento no Linux. Note que você ainda pode utilizar uma biblioteca do MySQL dinamicamente ligada à sua aplicação cliente. É só o servidor que é crítico para performance.
- Se você corta seu binário **mysqld** com **strip libexec/mysqld**, o binário gerado pode ficar até 4% mais rápido.
- Para uma conexão de um cliente para um servidor em execução na mesma máquina, se você conecta utilizando TCP/IP em vez de utilizar um arquivo socket Unix, o rendimento é 7.5% mais lento no mesmo computador. (Se você fizer conexão à **localhost**, o MySQL irá, por padrão, utilizar sockets).
- Para conexões TCP/IP de um cliente para um servidor, conectando a um servidor remoto em outra máquina será 8-11% mais lento que conectando ao servidor local na mesma máquina, mesmo para conexões Ethernet de 100M.
- Quando executar o nosso teste de benchamrk usando conexões seguras (todos os dados crptografados com suporte interno SSL) ele se torna 55% mais lento.
- Se você compilar com --with-debug=full, a maioria das consultas será 20% mais lentas. Algumas consultas podem demorar muito mais tempo (por exemplo, os benchmarks do MySQL demonstram 35% de perda). Se utilizar --with-debug, a queda será de apenas 15%. Para uma versão do **mysqld** compilada com --with-debug=full, você pode desabilitar a verificação de memória em tempo de execução iniciando-o com a opção --skip-safemalloc. O resultado final neste caso deve estar próximo de quando compilado com --with-debug.
- Em um Sun UltraSPARC-IIe, Forte 5.0 é 4% mais rápido que **gcc** 3.2.

- Em um Sun UltraSPARC-IIe, Forte 5.0 é 4% mais rápido em modo de 32 bits que em modo de 64 bits.
- Compilando com **gcc** 2.95.2 para o ultrasparc com a opção **-mcpu=v8 -Wa,-xarch=v8plusa** melhora a performance em 4%.
- No Solaris 2.5.1, a MIT-pthreads é 8-12% mais lenta do que as threads nativas do Solaris em um único processador. Com mais carga/CPUs a diferença deve aumentar.
- Executar com --log-bin deixa o **mysqld** 1 % mais lento.
- Compilando no Linux-x86 com gcc sem frame pointers **-fomit-frame-pointer** ou **-fomit-frame-pointer -ffixed-ebp** deixa o **mysqld** 1-4% mais rápido.

A distribuição MySQL-Linux fornecida pela MySQL AB é normalmente compilada com **pgcc**, mas vamos retornar ao uso do **gcc** pelo fato de um bug no **pgcc** que gera o código que não executa no AMD. Continuaremos a usar o **gcc** até que o bug seja resolvido. Neste meio tempo, se você possui uma máquina que não seja AMD, você pode ter um binário mais rápido compilando com o **pgcc**. O binário padrão do MySQL para Linux é ligado estaticamente para conseguir mais desempenho e ser mais portável.

5.4. Como o MySQL Utiliza a Memória

A lista abaixo indica algumas das maneiras inas quais o servidor **mysqld** utiliza a memória. Onde aplicável, o nome da variável do servidor relevante ao uso de memória é fornecido:

- O buffer de chave (variável **key_buffer_size**) é compartilhado por todas as threads; Outros buffers usados pelo servido são alocados quando necessários.
- Cada conexão utiliza algum espaço específico da thread: Uma de pilha (padrão de 64K, variável **thread_stack**), um buffer de conexão (variável **net_buffer_lenght**), e um buffer de resultados (variável **net_buffer_lenght**). Os buffers de conexões e resultados são aumentados dinamicamente para **max_allowed_packet** quando necessário. Quando uma consulta está sendo executada, uma cópia da string da consulta atual é também alocada.
- Todas as threads compartilhas a mesma memória base.
- Somente as tabelas **ISAM** e **MyISAM** compactadas são mapeadas em memória. Isto é porque o espaço de memória de 32-bits de 4GB não é grande o bastante para a maioria das grandes tabelas. Quando sistemas com endereçamento de 64-bits se tornarem comuns poderemos adicionar um suporte gieral para o mapeamento de memória.
- Cada requisição fazendo uma varredura sequencial em uma tabela aloca um buffer de leitura (variável **read_buffer_size**).
- Ao ler registros na ordem ``randômica" (por exemplo, depois de uma ordenação) um buffer de leitura randômico é alocado para evitar pesquisas em disco. (variável **read_rnd_buffer_size**).
- Todas as joins são feitas em um único passo, e a maioria delas podem ser feitas mesmo sem usar uma tabela temporária. A maioria das tabelas temporárias são tabelas baseadas em memória (HEAP). Tabelas temporárias com uma grande extensão de registros (calculada como a soma do tamanho de todas as colunas) ou que contenham colunas **BLOB** são armazenadas em disco.

Um problema nas versões do MySQL anteriores a 3.23.2 é que se uma tabela HEAP excede o tamanho de **tmp_table_size**, você recebe o erro **The table nome_tabela is full**. A partir da versão 3.23.2, isto é tratado alterando automaticamente a tabela em memória **HEAP** para uma tabela baseada em disco **MyISAM** quando necessário. Para contornar este problema, você pode aumentar o tamanho da tabela temporária configurando a opção **tmp_table_size** do **mysqld**, ou configurando a opção do SQL **SQL_BIG_TABLES** no progrma cliente. Na versão 3.20 do MySQL, o número máximo da tabela temporária é **record_buffer*16**; se você estiver utilizando esta versão, você terá que aumentar o valor **record_buffer**. Você também pode iniciar o **mysqld** com a opção --big-tables para sempre armazenar as tabelas temporárias em disco. Entretanto isto afetará a velocidade de várias consultas complicadas.

- A maioria das requisições que realizam ordenação alocam um bufer de ordenação e 0-2 arquivos temporários dependendo do tamanho do resultado.
- Quase todas as análises e cálculos são feitos em um armazenamento de memória local. Nenhuma sobrecarga de memória é necessário para ítens pequenos e a alocação e liberação normal de memória lenta é evitada. A memória é alocada somente para grandes strings inesperadas; isto é feito com **malloc()** e **free()**.
- Cada arquivo de índice é aberto uma vez e o arquivo de dados é aberto uma vez para cada thread concorrente. Uma estrutura de tabela, estrutura de coluna para cada coluna e um buffer de tamanho **3 * n** é alocado para cada thread concorrente. (onde **n** é o maior tamanho do registro, sem levar em consideração colunas **BLOB**. Uma coluna **BLOB** utiliza de 5 a 8 bytes mais o tamanho dos dados contidos na mesma. O manipulador de tabelas **ISAM**/**MyISAM** irão usar um registro extra no buffer para uso interno.
- Para cada tabela com colunas **BLOB**, um buffer é aumentado dinamicamente para ler grandes valores **BLOB**. Se você ler uma tabela, um buffer do tamanho do maior registro **BLOB** é alocado.
- Estruturas de manipulacão para todas tabelas em uso são salvos em um cache e gerenciado como FIFO. Normalmente o cache possui 64 entradas. Se uma tabela foi usada por duas threads ao mesmo tempo, o cache terá duas entredas para a tabela.
- Um comando **mysqladmin flush-tables** fecha (ou instruções **FLUSH TABLES**) todas tabelas que não estão em uso e marca todas tabelas em uso para serem fechadas quando a thread atualmente em execução terminar. Isto irá liberar efetivamente a maioria da memória em uso.

ps e outros programas de informações do sistema podem relatar que o **mysqld** usa muita memória. Isto pode ser causado pelas pilhas de threads em diferentes endereços de memória. Por exemplo, a versão do **ps** do Solaris conta a memória não usada entre as pilhas como memória usada. Você pode verificar isto conferindo a memória disponível com **swap -s**. Temos testado o **mysqld** com detectores comerciais de perda de memória, portanto tais perdas não devem existir.

5.5. Como o MySQL Utiliza o DNS

Quando um novo cliente conecta ao **mysqld**, o **mysqld** extende uma nova thread para lidar com o pedido. Esta thread primeiro confere se o nome da máquina está no cache de nomes de máquinas. Se não, a thread tenta resolver o nome da máquina.

- Se o sistema operacional suporta as chamadas seguras com thread **gethostbyaddr_r()** e **gethostbyname_r()**, a thread as utiliza para fazer a resolução do nome máquina.
- Se o sistema operacional não suporta as chamadas de threads seguras, a thread trava um mutex e chama **gethostbyaddr()** e **gethostbyname()**. Perceba que neste caso nenhuma outra thread pode resolver outros nomes de máquinas que não existam no cache de nomes de máquina até que a primeira thread esteja destrave o mutex.

Você pode desabilitar a procura de nomes de máquinas no DNS iniciando o **mysqld** com a opção --skip-name-resolve. No entanto, neste caso você só pode usar números IP nas tabelas de privilégio do MySQL.

Se você possuir um DNS muito lento e várias máquinas, pode obter mais desempenho desligando a procura de nomes de máquinas usando a opção --skip-name-resolve ou aumentando **HOST_CACHE_SIZE** (valor padrão: 128) e recompilar **mysqld**.

Você pode desabilitar o cache de nomes de máquinas iniciando o servidor com a opção --skip-host-cache. Para limpar a cache do nome de máquinas, envie uma instru;ção **FLUSH HOSTS** ou execute o comando **mysqladmin flush-hosts**.

Se você deseja disabilitar as conexões **TCP/IP** totalmente, inicie o **mysqld** com a opção --skip-networking.

5.6. Sintaxe de SET

```
SET [GLOBAL | SESSION] sql_variable=expression,
    [[GLOBAL | SESSION] sql_variable=expression] ...
```

SET configura várias opções que afetam a operação do servidor ou seu cliente.

Os seguintes exemplos mostram as diferentes sintaxes que se pode usar para configurar variáveis:

Em versões antigas do MySQL permitiamos o uso da sintaxe **SET OPTION**, mas esta sintaxe agora está obsoleta.

No MySQL 4.0.3 adicionamos as opções **GLOBAL** e **SESSION** e acessamos as variáveis de inicialização mais importantes.

LOCAL pode ser usado como sinôniumo de **SESSION**.

Se você define diversas variáveis na mesma linha de comando, o último modo **GLOBAL | SESSION** é utilizado

```
SET sort_buffer_size=10000;
SET @@local.sort_buffer_size=10000;
SET GLOBAL sort_buffer_size=1000000, SESSION sort_buffer_size=1000000;
SET @@sort_buffer_size=1000000;
SET @@global.sort_buffer_size=1000000,
@@local.sort_buffer_size=1000000;
```

A sintaxe **@@nome_variável** é suoprtada para tornar a sintaxe do MySQL compatível com outros bancos de dados.

As diferentes variáveis de sistema que podem ser configuradas estão descritas na seção de variáveis de sistema deste manual.

Se você estiver usando **SESSION** (o padrão) a opção que você definir terá efeito até que o sessão atual finalize ou até que vecê atribua um valor diferente a esta opção. Se você estiver usando **GLOBAL**, que exige o privilégio **SUPER**, a opção é lembrada e usada pelas novas conexões até que o servidor reinicie. Se você quiser tornar uma opção permanente, você deve definí-la em um arquivo de opção.

Para evitar o uso incorreto, o MySQL exibirá um erro se você usar **SET GLOBAL** com uma variável que só pode ser usada com **SET SESSION** ou se você não estiver usando **SET GLOBAL** com uma variável global.

Se você quiser definir uma variável **SESSION** com um valor **GLOBAL** ou um valor **GLOBAL** ao valor padrão do MySQL, você pode configurá-lo com **DEFAULT**.

```
SET max_join_size=DEFAULT;
```

Isto é idêntico a:

```
SET @@session.max_join_size=@@global.max_join_size;
```

Se você quiser restringir o valor máximo com o qual uma variável de servidor pode ser configurado com o comando **SET**, você pode especificaá-lo usando a opção de linha de comando --maximum-variable-name.

Você pode obter uma lista da maioria das variáveis com **SHOW VARIABLES**. Você pode obter o valor de uma variável específica com a sintaxe **@@[global.|local.]variable_name**:

```
SHOW VARIABLES like "max_join_size";
SHOW GLOBAL VARIABLES like "max_join_size";
SELECT @@max_join_size, @@global.max_join_size;
```

Segue aqui a descrição das variáveis que usam uma sintaxe **SET** não padrão e algumas das outras variáveis. A definição das outras variáveis podem ser encontrados na seção variáveis de sistema, entre as opções de inicialização ou na descrição de **SHOW VARIABLES**.

- **AUTOCOMMIT= 0 | 1**

  Se configurado com **1** todas alterações em uma tabela será feita de uma vez. Para iniciar uma transação de vários comandos, deve ser usada a instrução **BEGIN**. Se configurado com **0** deve ser usado **COMMIT**/**ROLLBACK** para aceitar/recusar aquela transação. Note que quando você altera do modo não-**AUTOCOMMIT** para **AUTOCOMMIT**, o MySQL irá fazer um **COMMIT** automático em quaisquer transações abertas.

- **BIG_TABLES = 0 | 1**

  Se definido com **1**, todas as tabelas temporárias são armazenadas no disco em vez de o ser na meória. Isto será um pouco mais lento, mas você não terá o erro **The table tbl_name is full** para grandes operações **SELECT** que exigem uma tabela temporária maior. O valor padrão para uma nova conexão é **0** (isto é, usa tabelas temporárias em memória) Esta opção era chamada **SQL_BIG_TABLES**. No MySQL 4.0 você normalmente nunca deve precisar deste parâmetro já que o MySQL converterá automaticamente tabelas em memória para tabelas em disco se isto for necessário.

- **CHARACTER SET nome_conjunto_caracteres | DEFAULT**

  Mapeia todas as strings do e para o cliente com o mapa especificado. Atualmente a única opção para **character_set_name** é **cp1251_koi8**, mas você pode adicionar novos mapas editando o arquivo sql/convert.cc na distribuição fonte do MySQL. O mapeamento padrão pode ser restaurado utilizando o valor **DEFAULT** para **character_set_name**.

  Perceba que a sintaxe para configurar a opção **CHARACTER SET** é diferente da sintaxe para configurar as outras opções.

- **DATE_FORMAT = format_str**

  Determina como o servidor converte valores DATE para strings. Esta variável está disponível como uma opção global, local ou de linha de comando. **format_str** pode ser especificado convenientemente usando a função **GET_FORMAT()**.

- **DATETIME_FORMAT = format_str**

  Determina como o servidor converte valores DATETIME para string. Esta variável está disponível como uma opção global, local ou de linha de comando. **format_str** pode ser especificada convenientemente usando a função **GET_FORMAT()**.

- **INSERT_ID = #**

  Configura o valor que será usado pelo comando **INSERT** ou **ALTER TABLE** seguinte ao inserir um valor **AUTO_INCREMENT**. Isto é usado principalmente com o log de atualizações.

- **LAST_INSERT_ID = #**

  Configura o valor a ser retornado de **LAST_INSERT_ID()**. Ele é armazenado no log de atualizações quando você utiliza **LAST_INSERT_ID()** em um comando que atualiza uma tabela.

- **LOW_PRIORITY_UPDATES = 0 | 1**

  Se configurado com **1**, todas instruções **INSERT**, **UPDATE**, **DELETE** e **LOCK TABLE WRITE** irão esperar até que não existam **SELECT** ou **LOCK TABLE READ** pendentes na tabela afetada. Esta opção era chamada **SQL_LOW_PRIORITY_UPDATES**.

- **MAX_JOIN_SIZE = value | DEFAULT**

  Não permite que **SELECT**s que provavelmente necessitem examinar mais que **valor** combinações de registros. Configurando este valor, você pode obter **SELECT**s onde chaves não são usadas corretamente e que provavelmente gastarão um bom tempo. Configurando-o para um valor diferente do **DEFAULT** irá definir o atributo **SQL_BIG_SELECTS** com o padrão. Se você configurar o atributo **SQL_BIG_SELECTS** novamente, a variável **SQL_MAX_JOIN_SIZE** será ignorada. Você pode configurar um valor padrão para esta variável iniciando o **mysqld** com **-O max_join_size=#**. Esta opção era chamada **SQL_MAX_JOIN_SIZE**

  Note que se o resultado da consulta ja estiver na cache de consultas, o verificaição acima não será feita. O MySQL irá enviar o resultado ao cliente. Uma vez que o resultado da consulta já foi consultado e não será responsabilidade do servidor enviar o resultado ao cliente.

- **PASSWORD = PASSWORD('alguma senha')**

  Configura a senha para o usuário atual. Qualquer usuário que não seja anônimo pode alterar sua própria senha!

- **PASSWORD FOR user = PASSWORD('alguma senha')**

  Configura a senha para um usuário específico no servidor atual. Somente um usuário com acesso ao banco de dados **mysql** pode fazer isto. O usuário deve ser fornecido no formato **usuário@home_maquina**, onde **usuário** e **nome_máquina** são exatamente o que estão listados nas colunas **User** e **Host** da tabela **mysql.user**. Por exemplo, se você possui uma entrada com os campos **User** e **Host** com **'bob'** e **'%.loc.gov'**, você escreveria:

```
mysql> SET PASSWORD FOR 'bob'@'%.loc.gov' = PASSWORD('newpass');
```

  Que é equivalente a:

```
mysql> UPDATE mysql.user SET Password=PASSWORD('newpass')
    ->                   WHERE User='bob' AND Host='%.loc.gov';
```

```
mysql> FLUSH PRIVILEGES;
```

- **QUERY_CACHE_TYPE = OFF | ON | DEMAND**, **QUERY_CACHE_TYPE = 0 | 1 | 2**

  Define a configuração da cache de consultas para esta thread. Set query cache setting for this thread.

Opção	Descrição
0 ou OFF	Não armazena ou recupera resultados.
1 ou ON	Armazena todos os resultados, exceto consultas **SELECT SQL_NO_CACHE ...**.
2 ou DEMAND	Armazena apenas consultas **SELECT SQL_CACHE ...**.

- **SQL_AUTO_IS_NULL = 0 | 1**

  Se configurado com **1** (padrão) o último registro inserido em uma tabela com um regitro auto_incremnto pode ser encontrado com a seguinte construção: **WHERE auto_increment_column IS NULL**. Isto é usado por alguns programas ODBC como o Access.

- **SQL_BIG_SELECTS = 0 | 1**

  Se configurado com **0**, o MySQL aborta as instruções **SELECT**s que provavelmente levam muito tempo (isto é, instruções para as quais o otimizador estima que o número de registros examinados provavelmente irá exceder o valor de **MAX_JOIN_SIZE**. Isto é útil quando uma instrução **WHERE** não aconselhada for utilizado. O valor padrão para uma nova conexão é **1** (que permitirá qualquer instrução **SELECT**).

  Se você definir **MAX_JOIN_SIZE** com um valor diferente de **DEFAULT**, **SQL_BIG_SELECTS** será definida com **0**.

- **SQL_BUFFER_RESULT = 0 | 1**

  SQL_BUFFER_RESULT força para que o resultado das **SELECT**'s seja colocado em tabelas temporárias. Isto irá ajudar o MySQL a liberar mais cedos bloqueios de tabela e ajudarão em casos onde elas ocupam muito tempo para enviar o conjunto de resultados para o cliente.

- **SQL_SAFE_UPDATES = 0 | 1**

  Se configurado com **1**, o MySQL irá aborar se tentarmos fazer um **UPDATE** ou **DELETE** sem utilizar uma chave ou **LIMIT** na cláusula **WHERE**. Desta forma é possível capturar atualizações erradas ao criarmos comandos SQL manualmente.

- **SQL_SELECT_LIMIT = valor | DEFAULT**

O número máximo de registros para retornar de instruções **SELECT**. Se uma **SELECT** tem uma cláusula **LIMIT**, o **LIMIT** tem precedêencia sobre o valor de **SQL_SELECT_LIMIT**. O valor padrão para uma nova conexão é ``unlimited" (ilimitado). Se você alterou o limite, o valor padrão pode ser restaurado atribuindo o valor **DEFAULT** a **SQL_SELECT_LIMIT**.

- **SQL_LOG_OFF = 0 | 1**

  Se configurado com **1**, nenhum registro será feito no log padrão para este cliente, se o cliente tiver o privilégio **SUPER**.

- **SQL_LOG_BIN = 0 | 1**

  Se configurada com **0**, nenhum registro é feito no log binário para o cliente, se o cliente tiver o privilégio **SUPER**.

- **SQL_LOG_UPDATE = 0 | 1**

  Se configurado com **0**, nenhum registro será feito no log de atualizações para o cliente, se o cliente tiver o privilégio **SUPPER**. Esta variável está obsoleta a partir da versão 5.0.

- **SQL_QUOTE_SHOW_CREATE = 0 | 1**

  Se configurado com **1**, **SHOW CREATE TABLE** irá colocar os nomes de tabela e colunas entre aspas. Está ***ligado*** por padrão, para que replicação de tabelas com nomes de colunas estranhos funcione.

- **TIMESTAMP = valor_timestamp | DEFAULT**

  Configura a hora/data para este cliente. É usado para obter a hora e data original se você utiliza o log de atualizações para restaurar registros. **valor_timestamp** deve ser um timestamp UNIX Epoch, não um timestamp MySQL.

- **TIME_FORMAT = format_str**

  Determina como o servidor converte valores **TIME** para string. Esta variável está disponível como uma opção global, local ou de linha de comando. **format_str** pode ser especificada convenientemente usando a função **GET_FORMAT()**

6. Detalhes de Disco

6.1. Utilizando Links Simbólicos

Como mencionado acima, pesquisas em disco são o maior gargalo de desempenho. Estes problemas ficam cada vez mais aparentes quando os dados começam a crescer tanto que efetivo armazenamento em cache se torna impossível. Para grandes bancos de dados, onde você acessa dados mais ou menos aleatoriamente, você pode ter certeza de que precisará de pelo menos uma busca em disco para ler e várias para gravar os dados. Para minimizar este problema, utilize discos com menor tempo de pesquisa.

- Aumente o número de eixo de discos disponíveis (e então reduza a sobrecarga da pesquisa) ligando arquivos simbolicamente em diferentes discos ou utilizando striping de discos.

  - ***Usando links simbólicos***

    Significa que, para tabelas MyISAM, você liga simbolicamente o índice e/ou arquivos de dados ao local comum no diretório de dados em outro disco (que pode também ser striped). Isto torna os tempos de pesquisa e leitura melhor (Se os discos não são usados para outras coisas).

  - ***Striping***

    Striping significa que você possui vários discos e coloca o primeiro bloco no primeiro disco, o segundo bloco no segundo disco, e o N-simo no (N módulo número_de_discos) disco, e assim por diante. Isto significa que se o seu tamanho de dados normais é menos que o tamanho do bloco (ou perfeitamente alinhado) você irá obter um desempenho muito melhor. Striping é muito dependente do SO e do tamanho do bloco. Portanto meça a performance de sua aplicação com diferentes tamanhos de blocos.

    Perceba que a diferença de velocidade para striping é ***muito*** dependente dos parâmetros. Dependendo de como você configura os parâmetros do striping e do número de discos você pode obter uma diferença de várias ordens de grandeza. Note que você deve escolher a otimização randômica ou pelo acesso sequencial.

- Para confiabilidade você pode desejar utilizar RAID 0+1 (striping + espelhamento) mas neste caso você irá precisar de 2*N discos para armazenar N discos de dados. Isto é provavelmente a melhor opção se você possuir dinheiro! Você pode também, entretanto, ter que investir em algum software gerenciador de volumes para lidar com isto eficientemente.

Uma boa opção é variar os níveis de RAID de acordo com a importância do dado. a Por exemplo, ter dados com alguma importância que podem ser regenerados em um armazenamento RAID 0 enquanto os dados realemtente importantes como informações de máquinas e logs em um sistema RAID 0+1 ou RAID de N discos. RAID N pode ser um problema se você tem várias escritas devido ao tempo para atualizar os bits de paridade.

- No Linux, você pode obter um desempenho muito melhor (cerca de 100% sobre carga pode ser comum) utilizando hdparm para configurar sua interface de disco! O exemplo a seguir deve ser muito útil para o MySQL (e provavelmente várias outras aplicações):

```
hdparm -m 16 -d 1
```

Perceba que o desempenho e confiança ao utilizar o exemplo acima depende de seu hardware, portanto nós sugerimos que você teste bem seu sistema depois de utilizar **hdparm**! Por favor consulte a página do manual (man) do **hdparm** para maiores informações! Se o **hdparm** não for usado corretamente, poderá resultar em corrupção do sistema de arquivos, assim realize backups de tudo antes de experimentar!

- Você pode também configurar os parâmetros para o sistema de arquivos que o banco de dados usa:

  - Se você não precisa saber quando os arquivos foram acessados pela última vez (o que é realmente útil em um servidor de banco de dados), você pode montar o seu sistema de arquivos com a opção **-o noatime**. Isto faz com que ele evite a atualização do último tempo de acesso no inode e com isto também evita algumas buscas em disco.
  - Em vários sistemas operacionais os discos podem ser montados com a opção 'async' para configurar o sistema de arquivos a ser atualizado de modo assíncrono. Se o seu computador é razoavelmente estável, isto deve fornecer mais desempenho sem sacrificar a segurança. (Esta opção é ligada por padrão no Linux.)

6.1. Utilizando Links Simbólicos

6.1.1. Utilizando Links Simbólicos para Bancos de Dados
6.1.2. Utilizando Links Simbólicos para Tabelas
6.1.3. Usando Links Simbólicos para Bancos de Dados no Windows

Você pode mover tabelas e bancos de dados do diretório de banco de dados para outras localizações e trocá-los por links simbólicas para os novos locais. Você pode fazer isto, por exemplo, para mover um banco de dados para um sistema de arquivos com mais espaço livre ou aumentar a velocidade de seu sistema espalhando suas tabelas para discos diferentes.

A maneira recomendada de se fazer isto é ligar simbolicamente bancos de dados a discos diferentes e só ligar tabelas como último recurso.

6.1.1. Utilizando Links Simbólicos para Bancos de Dados

No Unix, a maneira de ligar simbolicamente um banco de dados é, primeiramente, criar um diretório em algum disco onde você possui espaço livre e então criar uma ligação simbólica para ele a partir do diretório do banco de dados do MySQL.

```
shell> mkdir /dr1/databases/test
shell> ln -s /dr1/databases/test mysqld-datadir
```

O MySQL não suporta que você ligue um diretório a vários bancos de dados. Trocando um diretório de banco de dados com uma ligação simbólica irá funcionar bem desde que não sejam feitos links simbólicos entre os bancos de dados. Suponha que você tenha um banco de dados **db1** sob o diretório de dados do MySQL, e então criar uma ligação simbólica **db2** que aponte para **db1**.

```
shell> cd /caminho/para/diretorio/dados
shell> ln -s db1 db2
```

Agora, para qualquer tabela **tbl_a** em **db1**, também aparecerá uma tabela **tbl_a** em **db2**. Se uma thread atualizar **db1.tbl_a** e outra atualizar **db2.tbl_a**, ocorrerão porblemas.

Se você realmente precisar disto, você deve alterar o código seguinte em mysys/mf_format.c:

```
if (flag & 32 || (!lstat(to,&stat_buff) &&
S_ISLNK(stat_buff.st_mode)))
```

para

```
if (1)
```

No Windows você pode utilizar links simbólicos para diretórios compilando o MySQL com **-DUSE_SYMDIR**. Isto lhe permite colocar diferentes bancos de dados em discos diferentes.

6.1.2. Utilizando Links Simbólicos para Tabelas

Antes do MySQL 4.0 você não deve utilizar tabelas com ligações simbólicas, se você não tiver muito cuidado com as mesmas. O problema é que se você executar **ALTER TABLE**, **REPAIR TABLE** ou **OPTIMIZE TABLE** em uma tabela ligada simbolicamente, os links simbólicos serão removidas e substituidos pelos arquivos originiais. Isto acontece porque o comando acima funcinoa criando um arquivo temporário no diretório de banco de dados e quando o comando é completo, substitui o arquivo original pelo arquivo temporário.

Você não deve ligar simbolicamente tabelas em um sistema que não possui uma chamada **realpath()** completa. (Pelo menos Linux e Solaris suportam **realpath()**

No MySQL 4.0 links simbólicos só são suportados completamente por tabelas **MyISAM**. Para outros tipos de tabelas você provavelmente obterá problemas estranhos ao fazer qualquer um dos comandos mencionados acima.

O tratamento de links simbólicos no MySQL 4.0 funciona da seguinte maneira (isto é mais relevante somente para tabelas **MyISAM**.

- No diretório de dados você sempre terá o arquivo de definições das tabelas e os arquivos de índice e o arquivo de dados. O arquivo de dados e o arquivo de índice podem ser movidos para qualquer lugar e substituidos no diretorio de dados pelos links simbólicos. O arquivo de definição não pode.
- Você pode ligar simbolicamente o arquivo índice e o arquivo de dados para diretórios diferentes, independente do outro arquivo.
- A ligação pode ser feita partir do sistema operacional (se o **mysqld** não estiver em execução) ou usando as opções **DATA DIRECTORY** ou **INDEX DIRECTORY** em **CREATE TABLE**.
- **myisamchk** não irá substituir um link simbólico pelo índice/arquivo. Ele funciona diretamente nos arquivos apontados pelos links simbólicos. Qualquer arquivo temporário será criado no mesmo diretório que o arquivo de dados/índice está.
- Quando você remove uma tabela que está usando links simbólicos, o link e o arquivo para o qual ela aponta são apagados. Esta é uma boa razão pela qual você **não** deve executar **mysqld** como **root** e não deve permitir que pessoas tenham acesso de escrita ao diretórios de bancos de dados do MySQL.
- Se você renomear uma tabela com **ALTER TABLE RENAME** e não deseja alterar o banco de dados, o link simbólico para o diretório de banco de dados será renomeada corretamente.
- Se você utiliza **ALTER TABLE RENAME** para mover uma tabela para outro banco de dados, então a tabela será movida para outro diretório de banco de

dados e os links simbólicos antigos e os arquivos para os quais eles apontam serão removidos.

- Se você não utiliza links simbólicos, você deve usar a opção --skip-symlink do **mysqld** para garantir que ninguém pode usar **mysqld** para apagar ou renomear um arquivo fora do diretório de dados.

O que ainda não é suportado:

- **ALTER TABLE** ignora todas as opções de tabela **DATA DIRECTORY** e **INDEX DIRECTORY**.
- **SHOW CREATE TABLE** não relata se a tabela possui links simbólicos antes do MySQL 4.0.15. Isto também é verdade para **mysqldump** que usa **SHOW CREATE TABLE** para gerar instruções **CREATE TABLE**.
- **BACKUP TABLE** e **RESTORE TABLE** não respeitam links simbólicos.
- O arquivo **frm *nunca*** deve ser um link simbólico (como dito anteriormente, apenas os dados e índices podem ser links simbólicos). Fazer isto (por exemplo para fazer sinônimos), produzirá resultados errados. Suponha que você tenha um banco de dados **db1** sob o diretório de dados do MySQL, uma tabela **tbl1** neste banco de dados e você faça um link simbólico **tbl2** no diretório **db1** que aponmta para **tbl1**:

```
shell> cd /path/to/datadir/db1
shell> ln -s tbl1.frm tbl2.frm
shell> ln -s tbl1.MYD tbl2.MYD
shell> ln -s tbl1.MYI tbl2.MYI
```

Agora se uma thread lê **db1.tbl1** e outra thread atualiza **db1.tbl2**, haverá problemas: a cache de consultas será enganada (ela acreditará que **tbl1** não foi atualizado e retornará resultados desatualizados), o comando **ALTER** em **tbl2** também irá falhar.

6.1.3. Usando Links Simbólicos para Bancos de Dados no Windows

A partir do MySQL versão 3.23.16, o **mysqld-max** e servidores **mysql-max-nt** na distribuição MySQL são compilados com a opção **-DUSE_SYMDIR**. Isto permite que você coloque um diretório de banco de dados em discos diferentes adicionando um link simbólico para ele. (Isto é parecido com o a com que links simbólicos funcionam no Unix, embora o procedimento para configurar o link seja diferente).

No Windows, você cria um link simbólico para um banco de dados MySQL criando um arquivo que contem o caminho para o diretório de destino. Salve o arquivo no diretório de dados usando o nome de arquivo nome_bd.sym, onde **nome_bd** é o nome do banco de dados.

Por exemplo, se o diretório de dados do MySQL é C:\mysql\data e você precisa ter o banco de dados **foo** localizado em D:\data\foo, você deve criar o arquivo C:\mysql\data\foo.sym que contêm o caminho **D:\data\foo**. Depois disto, todas tabelas criadas no banco de dados **foo** serão criadas no D:\data\foo. O diretório D:\data\foo deve existir para ele funcionar. Note também que o link simbólico não será usado se um diretório com o nome do banco de dados existe no diretório de dados MySQL. Isto significa que se você já tem um diretório de banco de dados chamado foo no direorio de

dados, você deve movê-lo para D:\data antes do link simbólico ser efetivado. (Para evitar problemas, o servidor não deve estar executando quando você mover o diretório do banco de dados.)

Note que devido a penalidade que você tem na velocidade quando abre todas as tabelas, nós não habilitamos esta opção por padrão, mesmo se você compilar o MySQL com suporte a isto. Para habilitar links simbólicos você deve colocar no seu arquivo **my.cnf** ou **my.ini** a seguinte entrada:

```
[mysqld]
symbolic-links
```

No MySQL 4.0 --simbolic-links está habilitado por padrão. Se você não precisa usá-lo você pode usar a opção **skip-symbolic-linkd**.

Para mais informações acesse o site: www.mysql.com