Ano Lectivo 2013/2014

Sistemas de Bases de Dados

Professor: José Júlio Alferes

Análise do Sistema de Gestão de Bases de Dados:

Trabalho Realizado pelo Grupo 15:

41872, Igor Valente

41882, Mónica Silvestre

42044, Filipe Milheiro

Monte da Caparica, 31 de Maio de 2014

Índice

1. Introdução

1.1. Enquadramento do Trabalho

Este trabalho tem como objectivo consolidar a matéria estudada na cadeira de Sistemas de Bases de Dados. Para tal, analisaremos um Sistema de Gestão de Bases de Dados Relacional, o MySQL, que utiliza a linguagem de consulta SQL.

O MySQL é o Sistema de Gestão de Bases de Dados *open source* mais popular do mundo. Foi inicialmente projectado para trabalhar com aplicações de dimensões mais pequenas, mas hoje em dia é utilizado em aplicações com grandes dimensões de dados. Exemplo disso são organizações muito conhecidas como a NASA, a Google, o Facebook, o YouTube, o LinkedIn, a Nokia, entre muitas outras que o usam.

1.2. Estrutura do Documento

A nossa análise do MySQL incidirá principalmente sobre os conceitos aprendidos na cadeira de Sistema de Bases de Dados. Veremos como o MySQL gere o armazenamento de dados, as estruturas de Indexação e Hashing, a optimização de perguntas, a gestão de transacções, o controlo de concorrência e o suporte para Bases de Dados Distribuídas.

Por fim faremos uma comparação com o Oracle 11g, o sistema usado nas aulas práticas da cadeira.

1.3. História do MySQL

Na década de 90, David Axmark, Allan Larsson e Michael "Monty" Widenius precisaram de utilizar a API do mSQL, no entanto esta API não se mostrou suficientemente rápida para o que era preciso desenvolverem. Então, resolveram utilizar a API do mSQL para escrever em C e C++ uma nova API que deu origem ao MySQL. O seu desenvolvimento perdura até aos dias de hoje e emprega centenas de profissionais por todo o mundo. Em 2009 passou a ser propriedade da Oracle, quando esta comprou a Sun Microsystems. À medida que novos requisitos têm surgido na Internet, o MySQL tem sido desenvolvido e optimizado para aplicações *Web*, tendo-se tornado a plataforma escolhida por programadores *Web*.

O MySQL responde às necessidades que uma base de dados de grandes dimensões tem, o que lhe dá vantagens em relação aos seus concorrentes. As suas características estão na origem do seu enorme sucesso, tais como:

- Ser Relacional, proporcionando velocidade e flexibilidade;
- Portabilidade, suportando quase todas as plataformas actuais;
- Facilidade de utilização;
- Ser compatível com linguagens populares (Java, C++, C, Python, Perl, PHP, Ruby, Eiffel, entre outras);
- Ter um excelente desempenho e estabilidade;

- Suportar multi-threads, sendo possível utilizar vários CPUs;
- Escalabilidade, podendo lidar com bases de dados enormes.
- Segurança, oferecendo um sistema de privilégios e senhas criptografadas;

2. Armazenamento e File Structure

O MySQL usa várias estratégias de armazenamento de dados em buffers de memória, para aumentar o seu desempenho. Dispõe de um mecanismo muito fiável e com um óptimo desempenho: o *InnoDB*, que é usado por defeito desde a versão 5.5. O mecanismo usado por defeito até então era o *MyISAM.*

Pode-se mudar o *engine* por defeito numa secção, através do comando:

```
SET default_storage_engine=MYISAM;
```

Também se pode alterar o engine só para uma tabela:

```
ALTER TABLE t ENGINE = InnoBD;
```

2.1 Controlo de buffer management

O InnoDB mantém um ***buffer pool*** para *caching* de dados e índices na memória, semelhante a uma lista. Um buffer melhora o desempenho de um sistema de bases de dados, ao permitir que os dados sejam acedidos pela memória. Sem um buffer, os acessos seriam sempre ao disco, o que se torna muito mais dispendioso, dado que o acesso à memória é muito mais rápido do que o acesso ao disco. Quantos mais blocos de dados houverem na memória, menos acessos ao disco terão que ser realizados pelo buffer manager.

O *buffer manager* é responsável por receber pedidos de programas que precisam de blocos do disco. A sua função é procurá-los em memória, ou seja, no buffer. Se os encontrar, então devolve o seu endereço na memória aos programas que fizeram o pedido. Se não encontrar o pretendido, copia os blocos do disco para a memória. Para o fazer, o *buffer manager* tem que se certificar que há espaço em memória e se não houver, apaga blocos mais antigos ou que aparentemente não sejam necessários, para ter espaço para blocos novos. Depois de ter uma cópia em memória, retorna o seu endereço aos programas que solicitaram os dados.

Tal como a maioria dos Sistemas Operativos, quando o buffer está cheio, O InnoDB aplica o algoritmo **LRU** (*least recently used*), que elimina da memória o que não é usado há mais tempo. O InnoDB tem incorporado no algoritmo LRU uma estratégia de inserção no meio da lista. Ou seja, quando é necessário espaço para adicionar um novo bloco, o InnoDB elimina o bloco menos usado recentemente e adiciona o novo bloco no meio da sua lista. Esta estratégia tem como efeito encontrarem-se à cabeça da lista os blocos mais usados recentemente, e na cauda os blocos menos usados.

A lista é tratada como se de duas sublistas se tratasse, tendo um "***ponto médio***" onde ocorre a separação do mais recente e do mais antigo na memória. Este algoritmo permite que os blocos contidos no buffer sejam os mais utilizados por *queries*.

Tem 3/8 da lista dedicados a blocos antigos, e o espaço restante dedicado à sublista de blocos utilizados mais recentemente.

Um acesso a um bloco que está na parte "velha" da lista, torna-o "novo" e é então movido para a cabeça da lista. Porém, se o bloco não foi lido por uma *querie*, mas antecipadamente, por ter sido previsto que seria utilizado em breve, não passa logo para a cabeça e pode mesmo ser apagado se não chegar a ser lido.

Os blocos que não são acedidos vão passando progressivameente para a cauda da lista. Um bloco que não seja usado há muito tempo, acaba por chegar à cauda e é apagado quando for inserido um novo.

O InnoDB oferece variáveis para manipular o LRU:

innodb_buffer_pool_size: Para especificar o tamanho da pool do buffer. Idealmente, quer-se um buffer maior do que o necessário, se isso for possível, para que se possa melhorar o desempenho, reduzindo os acessos ao disco.

Quanto o *buffer pool* é grande, o InnoDB até pode guardar dados modificados por inserções e actualizações, para que as escritas no disco possam ser agrupadas, tendo como consequência um melhor desempenho.

innodb_olf_blocks_pct: Para especificar a percentagem do buffer que queremos para a sublista de blocos menos usados recentemente. Os valores variam entre 5% e 95%, e o valor definido por defeiro é 37%(3/8).

innodb_old_blocks_time: Para determinar quanto tempo em milisegundos, é que um bloco tem que estar na cauda depois do primeiro acesso, antes de poder ser passado para a cabeça. O valor pré-definido é 0.

Por definição, blocos lidos por *queries* são postos na sublista nova, o que significa que vão ficar no buffer pool bastante tempo. Se fizermos *mysqldump* ou um *scan* (um *select* a uma tabela, que não tenha cláusula *where)*, vão ser introduzidos muitos dados para o buffer. Pode acontecer que dados importantes e recentemente usados sejam despejados da memória para dar entrada a esses valores, que talvez nunca venham a ser novamente utilizados. Para que isso não aconteça o **innodb_old_blocks_time** deve ser maior do que zero. O seu valor pode ser alterado temporáriamente enquanto se efectuam *scans* e *dumps*:

```
SET GLOBAL innodb_old_blocks_time = 500;
/* queries que afectam o funcionamento LRU do buffer */
SET GLOBAL innodb_old_blocks_time = 0;
```

innodb_buffer_pool_instances: Divide o buffer pool num número de regiões especificado pelo utilizador. Cada região tem a sua própria lista LRU e estrutudas de dados, para reduzir a contenção durante leituras e escritas concorrentes na memória. Esta opção só tem efeito se o tamanho do buffer pool for de 1 gigabyte ou mais.

2.2 Sistema de ficheiros

Todas as tabelas e índices são guardados num sistema *tablespace*. O sistema tablespace contem um pequeno conjunto de ficheiros de dados, os ficheiros **ibdata**, que contêm os metadados de objectos relacionados com o InnoDB (o *data dictionary*), e as áreas de armazenamento para o *undo log*, o *change buffer* e o *doublewrite buffer*.

Uma *tablespace* consiste em *pages.* Cada *tablespace* tem uma página com um tamanho pré-definido de 16KB. O tamanho da página pode ser reduzido para 8KB ou 4KB, usando a opção **innodb_page_size**. As páginas estão agrupadas em *extents:* grupos de 1MB, contendo 64 páginas consecutivas de 16KB, por exemplo.

Os ficheiros numa tablespace chamam-se segmentos. Quando um segmento cresce dentro de uma tablespace, o InnoDB distribui as primeiras 32 páginas para ele, uma por uma. Depois, começa a distribuir extents ao segmento. Pode adicionar a um segmento grande mais de 4 extents de cada vez, para assegurar que os dados fiquem bem sequenciados.

Por cada *index* no InnoDB, dois segmentos são repartidos: Um para nós que não sejam folhas da *B-Tree;* Outro para as folhas. Assim, consegue-se melhor sequencialidade para os nós folha, que contém os dados.

Algumas páginas na tablespace contém bitmaps de outras páginas, e portanto, alguns extents não podem ser distribuídos em segmentos como um todo, mas apenas como páginas individuais.

O InnoDB suporta clustered index, estando a organização feita de acordo com a chave-primária das colunas, mas não suporta ***multitable clustering***.

2.2.1 Modo File-Per-Table

O file-per-table do InnoDB é uma alternativa flexivel. Cada tabela e os seus índices são guardados num ficheiro diferente. Cada ficheiro **.ibd** representa uma tablespace única.

Por defeito, nas versões mais recentes do MySQL, o **innodb_file_per_table** está activado. Cada nova tabela InnoDB e os seus indices associados, têm a sua própria *tablespace*, com um ficheiro de dados (*.ibd*) separado para cada tabela. Ter múltiplas tablespaces é crucial para usar muitas características do MySQL. Cada tablespace tem 64TB de tamanho limite.

Quando o **innodb_file_per_table** está desactivado, o InnoDB guarda todas as suas tabelas e indices num único *tablespace*, que pode consistir em vários ficheiros ou partições no disco. Esta única tablespace tem 64TB de espaço limite, tal como as múltiplas. Esta era a definição de origem em versões anteriores do MySQL, que criando uma tablespace enorme, podia originar problemas quando grandes quantidades de dados temporários eram carregados e depois apagados.

2.3. Tuplos de Tamanho Variável

O tamanho máximo de um tuplo é um pouco menor do que metade de uma página da base de dados, à excepção de colunas de tamanho variável (*VARBINARY*, *VARCHAR*, *BLOB* e *TEXT*). Ou seja, no máximo existem tuplos com aproximadamente 8KB. Colunas LONGBLOB e LONGTEXT têm que ter menos de 4GB e o tamanho total dos tuplos, incluindo colunas BLOB e TEXT, deve ser menos do que 4GB.

Se um tuplo é menor do que meia página, então está todo guardado na página, se excede a meia página, então são escolhidas colunas de tamanho variável para armazenamento *external off-page*, até que o tuplo caiba dentro da meia página. Para uma coluna que tenha sido escolhida para armazenamento *off-page*, os primeiros 768 bytes são guardados localmente no tuplo, e o resto é guardado externamente em ***overflow pages***. Cada coluna destas tem a sua própria lista de *overflow pages*. Para

além do prefico de 768 bytes, existe um valor de 20 bytes que armazena o comprimento da coluna e aponta para o sitio na lista overflow onde o resto do valor é guardado.

É possível especificar o formato de um tuplo para uma tabela, usando por exemplo, a seguinte expressão, com o ROW_FORMAT:

```
CREATE TABLE t1 (f1 int unsigned) ROW_FORMAT=COMPACT ENGINE=INNODB;
```

O ROW_FORMAT pode ser ***compact***, *reduntant*, *dynamic* e *compressed*. Por defeito, em tabelas InnoDB, os tuplos são guardados no formato compacto. O formato tem uma representação para nulls e campos de tamanho variável mais compacta. Como os B-tree *indexes* tornam a procura de tuplos tão rápida, há poucos benefícios em manter todos os tuplos do mesmo tamanho no InnoDB.

2.4. Mecanismo de Partições

O MySQL dispõe de um mecanismo de partições, que permite a distribuição de porções de tabelas por um sistema de ficheiros, conforme os tamanhos pretendidos. Diferentes porções de uma tabela são guardadas como tabelas separadas em localizações diferentes. A regra seleccionada pelo utilizador responsável por dividir os dados é conhecida por função de partição, e depende do tipo de partição que utilizador deseja. No MySQL pode ser o módulo, uma função de hashing interna ou uma função de hashing linear.

A função é seleccionada de acordo com o tipo de partição especificado pelo utilizador e tem como parametro o valor de uma expressão também fornecida pelo utilizador. Esta expressão pode ser um valor de uma coluna, uma função que age sobre um ou mais valores de colunas, ou um conjunto de um ou mais valores de colunas, dependendo do tipo de partição que é usado. A função retorna um inteiro, ou NULL, que representa o número da partição onde o registo deve ser guardado.

2.4.1. Tipos de Partições

RANGE: Cada partição da tabela contém registos de acordo com a expressão de partição que tem um dado intervalo. Exemplo de 4 partições, cada uma com um intervalo:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    store_id INT NOT NULL
)
PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN (21)
```

```
);
```

LIST: É semelhante ao particionamento por RANGE, mas a partição é seleccionada de acordo com as colunas que correspondem a um ou mais conjunto de valores. Exemplo, em que as lojas ficam separadas por zona:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY LIST(store_id) (
    PARTITION pNorth VALUES IN (3,5,6,9,17),
    PARTITION pEast VALUES IN (1,2,10,11,19,20),
    PARTITION pWest VALUES IN (4,12,13,14,18),
    PARTITION pCentral VALUES IN (7,8,15,16)
);
```

HASH: A partição é seleccionada com base no valor retornado por uma expressão definida pelo utilizador que opera sob valores de colunas em registos a inserir na tabela. Os dados são distribuídos de forma equilibrada pelas partições. Exemplo, em que são feitas 4 partições:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY HASH(store_id)
PARTITIONS 4;
```

KEY: Semelhante ao HASH, mas é particionado pela chave-primária da tabela. Exemplo:

```
CREATE TABLE k1 (
    id INT NOT NULL PRIMARY KEY,
    name VARCHAR(20)
)
PARTITION BY KEY()
PARTITIONS 2;
```

Caso não haja chave-primária, mas haja uma chave única, então a chave única é usada para fazer a partição por Key.

Subpartitioning: É a divisão de cada partição numa tabela particionada. É possível subparticionar tabelas que estão particionadas por Range ou List. Subpartições podem usar partição Hash ou Key. Exemplo:

```
CREATE TABLE ts (id INT, purchased DATE)
    PARTITION BY RANGE( YEAR(purchased) )
```

```
    SUBPARTITION BY HASH( TO_DAYS(purchased) )
    SUBPARTITIONS 2 (
        PARTITION p0 VALUES LESS THAN (1990),
        PARTITION p1 VALUES LESS THAN (2000),
        PARTITION p2 VALUES LESS THAN MAXVALUE
    );
```

3. Indexação e hashing

3.1.Tipos de Índices

O gestor de armazenamento InnoDB tem dois tipos de índices:

- Índices “clustered” – são índices que usam a chave primária como índice, seja por fornecimento do administrador ou criadas artificialmente pelo InnoDB, através da escolha de um atributo UNIQUE que não permita colunas NULL ou, em caso de ausência de uma capaz, pela criação de uma coluna artificial com um ID gerado pelo sistema;
- Índices secundários – são todos os índices que não os “clustered”. Incluem em cada registo os atributos que compõem a chave primária, assim como os atributos escolhidos para o índice secundário.

Dentro destes tipos, o MySQL suporta B-Tree, por definição, e Hash e Bitmaps.

Sintaxe usada para criar um índice em MySQL:

```
CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name
      [index_type]
      ON tbl_name (index_col_name,...)
      [index_type]
index_col_name:
      col_name [(length)] [ASC | DESC]
index_type:
      USING {BTREE | HASH}
```

3.2. Tipo de Estruturas Implementadas

O InnoDB usa como estruturas de dados as B-Trees. Estas também são usadas para armazenamento dos dados, contendo cada B-Tree índices cujos dados estão nas folhas.

3.3. Optimização

Este gestor de armazenamento permite também a utilização de mais do que um índice, por forma a acelerar as pesquisas.

No entanto, a utilização de dois ou mais índices pode provocar problemas, ao invés de agilizar. Há que ter em conta, principalmente, que dado que o InnoDB tem sempre uma chave primária, devemos escolher para os índices as colunas constituintes da chave primária mais importantes e que queiramos sempre rapidamente disponíveis. A utilização de demasiadas ou demasiado longas colunas pode prejudicar severamente a performance, dado que os valores destas colunas duplicam em cada índice secundário. Importante também criar um número pequeno de índices concatenados, ao invés de muitos índices com uma coluna apenas, pois índices baseados em colunas raramente visitadas podem prejudicar a pesquisa.

3.4. Estruturas Inconsistentes

Por norma, o InnoDB tem a função *autocommit*, que faz com que qualquer alteração pós-instrução seja efectivamente guardada na base de dados, impedindo que a mesma fique em estado inconsistente. No entanto, é possível ser desactivada.

Garante também, através do método de escrita *doublewrite*, que traz segurança na recuperação após um problema externo ou interno, a preservação dos dados em estado consistente. Este método traduz-se em que, antes de escrever num ficheiro de dados, o sistema aguarda pela escrita e vazamento de uma zona contígua especial denominada *doublewrite buffer* para que, caso falhe uma escrita, se consiga encontrar uma cópia fiável da mesma na recuperação.

3.5. Mecanismos para ver o Plano de Execução

Em MySQL, o mecanismo EXPLAIN fornece informação sobre o plano de execução para uma pesquisa, devolvendo uma linha de informação sobre cada tabela usada no SELECT. Lista as informações pela ordem em que as leria caso o MySQL caso viesse a processar a pesquisa.

Em baixo estão indicados os campos que o EXPLAIN devolve:

Coluna	Significado
id	O identificador do SELECT
select type	O tipo do SELECT
table	A tabela para a linha de resultado
partitions	As partições de resultado
type	O tipo de join
Possible keys	Os indices por onde escolher
key	O índice escolhido
key len	O comprimento da chave escolhida
ref	As colunas comparadas com o índice
rows	Estimativa das linhas a serem verificadas
filtered	Percentagem de linhas filtradas pela condição da tabela
extra	Informação adicional

3.6. Exemplos de aplicação em MySQL

Em baixo demonstra-se a aplicação de índices numa pesquisa da base de dados USCENSUS, utilizada nas aulas práticas, e o plano EXPLAIN.

- *Caso com um índice*

```
select * from uscensus where id = 9000;
create index idindex on uscensus(id);
alter table uscensus drop index idindex;
```

Primeiro sem o índice, mais lento…

Explain

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	uscensus	ALL					32575	Using where

E depois com índice activado em id.

Explain

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	uscensus	ref	idindex	idindex	5	const	1	Using where

- *Casos com dois índices*

Caso 1:

```
select * from uscensus where maritalstatus = 'Divorced' and education = 'HS-grad';
```

Caso 2:

```
select * from uscensus where maritalstatus = 'Never-married' and education = 'Masters';
create index eduindex on uscensus(education);
create index maritalindex on uscensus(maritalstatus);

alter table uscensus drop index eduindex;
alter table uscensus drop index maritalindex;
```

Primeiro sem os índices, em 1 e 2…

Explain

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	uscensus	ALL					32690	Using where

Por fim, com os índices activados em maritalstatus e education, em simultâneo. Em 1…

Explain

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	uscensus	ref	eduindex,maritalindex	maritalindex	28	const	4442	Using where

E em 2.

Explain

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	uscensus	index_merge	eduindex,maritalindex	eduindex,maritalindex	18,28		861	Using intersect(eduindex,maritalindex); Using where

4. Processamento e Optimização de Perguntas

4.1. Optimização de Operações Básicas

Para tornar uma pesquisa do tipo SELECT…WHERE mais rápida no MySQL, este tem à disposição vários mecanismos:

Where

No caso das cláusulas WHERE, a optimização passa por, principalmente:

- Eliminação de parêntesis desnecessários;

- Compactação e simplificação de expressões;

- Remoção de condições desnecessárias:

- Avaliação única de expressões constantes utilizadas por índices;

- Rápida e atempada detecção de expressões inválidas;

- Para cada tabela em WHERE é construído um WHERE mais simples, de forma a fazer uma avaliação mais rápida da tabela e saltar registos o mais cedo possível.

Para cada expressão **SELECT** existem três otimizações gerais possíveis:

RANGE – método que utiliza apenas um índice para obter um subconjunto de registos contidos num ou mais intervalos de valores de índices, que pode ser utilizado para um índice com um ou mais atributos;

INDEX MERGE – método que é usado para obter registos com diversos varrimentos de intervalos e juntá-los num só;

ENGINE CONDITION PUSHDOWN – esta operação melhora a eficiência de comparações directas entre uma coluna não indexada e uma constante; Nestes casos, a condição é “empurrada para baixo”, ou seja, é precipitada a sua execução.

No entanto, existem optimizações mais concretas para as expressões:

IS NULL – usada para verificar se cada coluna tem valores nulos. Utiliza índices e intervalos para a verificação. Nos casos em que procura valores nulos numa coluna NOT NULL, salta essa mesma coluna.

LEFT E RIGHT JOIN – o optimizador de joins calcula a ordem pela qual um join deve ser executado. A leitura forçada por LEFT ou RIGHT JOIN ajuda o optimizador muito mais rapidamente, visto que tem que verificar menos permutações entre tabelas.

NESTED-LOOP JOIN ALGORITHM – este algoritmo utiliza a colocação em buffer de registos lidos no ciclo externo para reduzir o número de vezes que as tabelas no ciclo interno devem ser lidas. A redução do número de vezes que as tabelas internas devem ser lidas pode ser de uma magnitude.

NESTED JOIN – no caso de joins que envolvem apenas inner joins, os parêntesis podem ser removidos. Para as pesquisas com outer joins, o optimizador pode escolher uma ordem em que os loops para tabelas externas precedem os loops de tabelas internas.

OUTER – na fase de parsing, pesquisas com right outer join são convertidas em pesquisas equivalentes, contendo apenas operações left outer join. Quando o optimizador avalia planos para pesquisas de join com outer join, toma em consideração apenas os planos em que, para cada operação desse tipo, as tabelas de fora são acedidas antes das tabelas de dentro.

ORDER BY – nalguns casos, o MySQL consegue usar um índice para satisfazer uma cláusula ORDER BY sem qualquer tipo de ordenação extra. O índice pode inclusive ser usado mesmo que o ORDER BY não corresponda exactamente ao índice, desde que as partes não usadas no índice e as colunas extra do ORDER BY sejam constantes na cláusula WHERE.

GROUP BY – normalmente, a forma de satisfazer a cláusula GROUP BY passa pela criação de uma tabela temporária onde estejam contidos todos os registos consecutivos, posteriormente descobrindo grupos e aplicando funções de agregação. Nalguns casos, o MySQL consegue fazer muito melhor e inclusive evitar o uso de tabelas temporárias, através do uso de índices. Existem dois métodos para analisar o GROUP BY:

LOOSE INDEX SCAN – o modo mais eficiente de processar um GROUP BY é quando um índice é usado para obter as colunas a agrupar. Com este método de acesso, o MySQL utiliza a propriedade da ordenação de chaves de alguns índices, como por exemplo, a B-Tree. Esta propriedade permite o uso de grupos de pesquisa num índice sem ter que considerar todas as chaves no índice que satisfazem todas as condições do WHERE.

TIGHT INDEX SCAN – pode ser um FULL INDEX SCAN ou um RANGE INDEX SCAN, dependendo das condições da pesquisa. Quando não se reúnem as condições para um LOOSE INDEX SCAN, pode ainda ser possível evitar tabelas temporárias. Se

existirem condições de intervalos na cláusula WHERE, este método lê apenas as chaves que satisfazem essas condições. Caso contrário, faz um INDEX SCAN.

DISTINCT – em muitos casos, a cláusula DISTINCT é considerada um caso especial do GROUP BY. Aplicam-se as mesmas regras para optimizar.

EXISTS – certas optimizações são aplicáveis a comparações que usam o operador IN para testar resultados de subconsultas. Podem ser feitas várias optimizações, tendo em conta os desafios que os valores NULL apresentam.

4.2. Estimativas para as pesquisas

No caso do MySQL e na maioria dos casos que este trata, pode-se estimar a performance de uma pesquisa através do número de pesquisas em disco. Dado que o índice se encontra provavelmente em cache, a pesquisa de uma tabela pequena necessita apenas de uma pesquisa em disco. No caso de tabelas maiores, pode-se estimar que, usando os índices B-Tree, se encontra um registo num determinado número de pesquisas através da seguinte fórmula:

```
log(row_count) / log(index_block_length / 3 * 2 / (index_length + data_pointer_length)) + 1
```

5. Gestão de transações e controlo de concorrência

5.1. Transações

O InnoDB tem suporte para transações com propriedades ACID (Atomicidade, Coerência, Isolamento e Durabilidade).

O modelo de transações usado tem como objetivo combinar o melhor das *multi-versionig databases* com o tradicional protocolo *Two-phase locking.* Por omissão o nível de granularidade nos *lockings* é *row level* e as operações de *read* são não-bloqueantes. Os utilizadores podem realizar de *lock* a cada entrada da tabela ou a um conjunto de entradas aleatórias, sem causar quaisquer problemas de exaustão de memória.

O InnoDB não possui suporte para *nested transactions*.

Apresentamos em seguida as sintaxes das diversas transações suportadas:
`START TRANSACTION [WITH CONSISTENT SNAPSHOT] | BEGIN [WORK]` - Início de forma explícita de uma transação nova.
`COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]` – Fim de uma transação com atualização das operações que ainda não foram commit com sucesso.
`ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]` - Fim de uma transação sem sucesso e por isso tem de fazer rollback à transação corrente, cancelando a mesma.
`SET autocommit = {0 | 1}` – por defeito o MySql começa em modo autocommit = 1 (ligado, que faz commit após cada operação feita com sucesso em caso de erro é feito rollback) caso se queira mudar o modo como o MySql trata as operações basta fazer autocommit = 0 (desligado).
`[WITH CONSISTENT SNAPSHOT]` - Dá início a uma leitura consistente. Produz o mesmo efeito que um START *TRANSACTION* seguido de um SELECT de qualquer tabela InnoDB. Esta opção não causa a alteração do nível de isolamento transacional corrente e, portanto, garante que a cópia (*snapshot*) é consistente apenas se o nível e isolamento atual for um que permita leituras consistentes (*REPEATABLE READ* ou *SERIALIZABLE*).
`[AND CHAIN]` – Esta cláusula faz com que uma nova transação (em espera) inicie logo de imediato ao término da transação corrente. O nível de isolamento da transação que se inicia, será igual ao da corrente.
`[RELEASE]` – Esta cláusula faz com que o servidor se desconecte da atual sessão do cliente assim que termine transação.

5.2 – Isolamento

Em termos de isolamento o MySql InnoDB apenas suporta protocolos baseados em locks (protocolo pessimista). Sendo que por omissão, o nível de granularidade é *row-level*, para o qual há dois modos de fazer *lock*:

- ***Shared (S)*** – permite a uma transacção ler um tuplo;
- ***Exclusive (X)*** – permite a uma transacção actualizar ou remover um tuplo.

Adicionalmente em termos de granularidade é suportada multiple granularity locking, para a qual também existem dois modos de fazer lock:

- ***Intention shared (IS):*** se uma transação pretender fazer *shared Locks* a tuplos da tabela.
- ***Intention exclusive (IX):*** se uma transação pretender fazer exclusive *locks* a tuplos da tabela.

Em seguida apresentamos a sua sintaxe:

```
SELECT … LOCK IN SHARE MODE (IS Lock).
SELECT … FOR UPDATE (IX Lock).
```

Estes *locks* aplicados a tabelas são tratados da seguinte forma:

- Antes de uma transação poder adquirir um *shared lock* sobre tuplo de uma tabela, deve primeiro adquirir um *IS lock* na tabela.
- Antes de uma transação poder adquirir um *exclusive lock* num tuplo de uma tabela, deve primeiro adquirir um *IX lock* na tabela.

O principal objetivo deste tipo de *locks (IS e IX)* é apenas mostrar que tem um tuplo bloqueado, ou então vai bloquear um tuplo numa tabela.
Um *lock* é concedido a uma transação se for compatível com os *locks* existentes. Caso haja algum conflito com algum *lock* já existente, a transação tem que esperar até que os (s) *lock* (s) que estão a provocar o conflito sejam libertados. Se houver um conflito entre a solicitação de um *lock* com um *lock* existente e este não pode ser concedido, uma vez que isso causaria um *deadlock*, ocorre um erro.

5.3. Detecção e Recuperação

O *InnoDB* automaticamente detecta situações de *deadlock*, e quando tal acontece, faz *rollback* à (s) transacção (ões) que deram origem ao *deadlock*, faz isso para minimizar o número de *rollbacks* necessários. Por outro lado, há situações em que não é possível detectar *deadlocks*:

- Quando é feito um *table lock*, usando a instrução LOCK TABLES;
- Ou quando são feitos *locks,* por outra *storage engine,* que não o *InnoDB.*

Nestes casos, o problema pode ser resolvido pelo utilizador através da indicação do parâmetro *innodb_lock_wait_timeout*, que define o tempo máximo que uma transacção pode ficar a espera de um *lock* antes de causar *rollback* na transacção que detém o *lock.*

Por sua vez, há outros casos em que os *locks* são mantidos e, isso acontece se houver apenas uma instrução *SQL,* que tem como resultado um erro e, consequentemente um *rollback*. Tal é possível, porque o *InnoDB* armazena os *row locks* num formato que, seja impossível determinar por qual instrução o bloqueio foi gerado.

Em seguida é apresentado um exemplo de **deadlock**:

T1	T2
`insert(7,'seven');`	
	`insert(8,'eight');`
	`insert(7,'seven');`
`insert(8,'eight');`	
`commit;`	
	`commit;`

5.4. Níveis de granularidade

O *InnoDB* suporta essencialmente dois níveis de granularidade através de protocolos de *lock*, o *Row-Level lock* e o *Table-Level lock*. Por omissão usa o *row-level locking*.

5.4.1. Row-level Locking

Este nível de granularidade subdivide-se em três modos:

· ***Record lock*** – o *lock* é feito sobre os registos de índice, mesmo que na tabela não estejam definidos. Quando estes não estão definidos, o *InnoDB* cria um *clustered index* oculto e usa-o para a operação de *lock;*

· ***Gap lock*** – é um *lock* que funciona como um mecanismo de reserva, isto porque, bloqueia espaços livres entre registos de índice, ou bloqueia espaços livres no início ou depois do último registo de índice;

· ***Next-key locking*** – Combina os dois modos anteriores. Quando é feita uma pesquisa, faz *locks* partilhados ou exclusivos nos registos de índice que encontra. No entanto, como este modo também afecta os espaços livres entre registos de índice, os registos que antecedem o registo de índice, são afectados também. Isto origina que, quando uma sessão tem o *lock (S* ou *X)* sobre um registo *reg*, uma outra sessão é incapaz de fazer inserções dum novo registo índice no espaço livre imediatamente anterior a *reg*. Em certas circunstâncias, as leituras consistentes (não bloqueantes) não são convenientes, sendo necessárias leituras bloqueantes. No *InnoDB* existem dois tipos de leituras bloqueantes:

o `SELECT ... LOCK IN SHARE MODE` – define um *share mode lock* nos registos que são lidos. Este modo permite que outras sessões tenham acesso

aos registos sem os modificar. Cada leitura vê a informação mais recente e cria um *shared mode lock* nos registos lidos. Assim, previne-se que outras sessões actualizem ou eliminem esses registos lidos. Além disso, se a informação mais recente pertencer a uma transacção ainda incompleta (*uncommited*) doutra sessão, têm-se que esperar que ela termine.
o SELECT ... FOR UPDATE – permite bloquear outras sessões de realizarem operações SELECT ... LOCK IN SHARE MODE ou de ler em certos níveis de isolamento. As leituras consistentes ignoram quaisquer *locks* feitos sobre registos que estão presentes numa vista de leitura.

Quando a transacção termina com *commit* ou *rollback* estes *locks* são descartados.

5.4.2. Table-level Locking

O *InnoDB* suporta também *locks* à tabela toda, em vez dum conjunto específico como no *row-level*. Quando é usada a *instrução LOCK TABLES*, é usado este tipo de bloqueio. E a sua sintaxe é a seguinte:

```
LOCK TABLES  tbl_name [[AS] alias] lock_type
[ tbl_name [[AS] alias] lock_type]
... lock_type: READ [LOCAL]
| [LOW_PRIORITY] WRITE UNLOCK TABLES
```

5.5. Níveis de Isolamento

O *InnoDB* possui 4 níveis de isolamento:

· **READ UNCOMMITTED** - as operações de seleção atuam duma forma não bloqueante, mas é possível visualizar uma versão mais recente dos dados. Assim, usando este nível de isolamento, as leituras não são de todo consistentes.

· **READ COMMITTED** – Cada leitura consistente (não bloqueante), dentro da mesma transação, atualiza e lê a sua própria cópia mais recente. No caso das leituras bloqueantes (quando é usada granularidade em *SHARE MODE*), o *InnoDB* apenas bloqueia os registos de índice, não os espaços livres existentes entre eles. Assim, é possível fazer inserções de registos no espaço livre a seguir aos registos que se encontram bloqueados. Nas operações de *UPDATE* e *DELETE*, o bloqueio depende de, se a instrução usa um *unique index* na condição de pesquisa:

- Se a condição *WHERE* for uma igualdade então bloqueia apenas o registo que é encontrado na pesquisa.
- Se a condição *WHERE* for uma comparação então bloqueia os registos de índice com *gap lock* ou *next-key locking,* para os registos que estão abrangidos pelo intervalo de pesquisa.

· **REPEATABLE READ** – Nível de isolamento por defeito no *InnoDB*. Nas leituras consistentes, a principal diferença com o *READ COMMITED* é que, todas as leituras consistentes dentro da mesma transacção lêem a mesma cópia que é estabelecida pela primeira leitura. Isto levanta uma situação em que, se forem efetuadas várias operações de *SELECT* dentro da mesma transação, essas operações são consistentes em relação umas às outras. No caso das leituras bloqueantes, o comportamento é idêntico ao nível de isolamento descrito anteriormente.

· **SERIALIZABLE** – Este nível é idêntico ao *REPEATABLE READ*. No caso da opção *autocommit* estar desactivada, o *InnoDB* implicitamente converte todas as operações de *SELECT* para *SELECT...LOCK IN SHARE MODE*. Caso esteja ativa, cada operação de seleção é ela própria uma transação.

Apresentamos em seguida as sintaxes dos diversos níveis de isolamento suportados:

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL
{
        READ UNCOMMITTED | READ COMMITTED
        | REPEATABLE READ | SERIALIZABLE
}
```

- Alterar o nível de isolamento

GLOBAL – permite definir o nível de isolamento atual para todas as sessões que não sejam a sessão atual.

SESSION – define o nível de isolamento atual para todas as operações realizadas dentro da sessão atual.

```
transaction-isolation = [READ-UNCOMMITED | READ-COMMITED | REPEATABLE-READ
| SERIALIZABLE]
```

- Definir o nível de isolamento no início da execução do servidor.

`SELECT @@GLOBAL.tx_isolation, @@tx_isolation` - Consultar o nível de isolamento atual (no caso de ser global).

Em seguida mostramos um exemplo da utilização de dois destes niveis de isolamento (READ COMMITTED e SERIALIZABLE):

Table t

1 a

2 b

T1

```
insert into t values(3,'c');

insert into b select count(*) from t;
```

T2

```
insert into t values(4, 'd');


commit;
```

```
insert into a select count(*) from t;

commit;
```

Sendo que o resultado de efetuar estas duas transações utilizando cada nível de isolamento seria o seguinte:

READ COMMITTED	**SERIALIZABLE**
a -> 3	a -> 2
b -> 4	b -> 3

5.6. Consistência

No *InnoDB* os problemas de inconsistência são chamados de *Phantom Problem*. Este tipo de problema acontece dentro duma transação, quando uma mesma consulta produz resultados diferentes. Exemplo disso é quando uma instrução SELECT é executada duas vezes, mas retorna um tuplo na segunda chamada que não foi mostrada na primeira, a este problema chamado de *phantom row*. Para prever a ocorrência deste tipo de problemas o *InnoDB* usa o algoritmo *next-key locking*. Esta forma de bloqueio, faz com que os registos de índice, que obedeçam aos critérios de pesquisa sejam bloqueados e, também outros registos que estejam num intervalo de índices ou antes ou depois dos registos de índices. É desta forma que, o *InnoDB* previne a ocorrência de registos "*ghost*" no sistema e também para implementar restrições de unicidade.

5.7. Atomicidade e Durabilidade

O *InnoDB* é uma *muti-versioned storage engine* então, mantém informação sobre estados anteriores dos registos no *tablespace*, mais especificamente, numa zona conhecida como *rollback segment.*
Internamente, o *InnoDB* adiciona três campos a cada registo, que é armazenado na base de dados:

· **DB_TRX_ID** - Um registo de 6 bytes, que indica a transacção que mais recentemente inseriu ou actualizou o tuplo. No caso da remoção, é tratado como um UPDATE, recorrendo a um bit para identificar tal operação.

· **DB_ROLL_PTR** – Um registo de 7 bytes conhecido como *roll pointer*. Este registo serve como "apontador" para um registo de *log* escrito no segmento de *rollback*. Quando o tuplo é actualizado, este *undo log* contem a informação necessária para repor o conteúdo do tuplo antes de ser actualizado.

· **DB_ROW_ID** – Um registo de 6 bytes, que contem o *row ID* que incrementa automaticamente assim que um tuplo é inserido. Caso sejam usados *clustered index,* o

índice contém como valor os *row ID*, senão a coluna DB_ROW_ID nem sequer aparece em qualquer índice.

O *InnoDB* usa este segmento de *rollback* para repor a base de dados num estado anterior, aquando duma transação ou para lidar com consistência nas transações. O mecanismo de *rollback* usado é o *log-based recovery*.

5.7.1. Rollback segment

Os *logs* de *rollback* estão divididos em duas partes:

· ***Insert Undo Log*** – necessários apenas num *rollback* duma transacção e são descartados assim que é feito *commit*;

· ***Update Undo Log*** – usados para a consistência de transações. Estes também podem descartados, mas só após existir a garantia que não há nenhuma transação para a qual foi "tirado" um *snapshot,* que possa conter informação necessária para repor o estado mais recente da base de dados. É preciso ter em conta que se senão forem feitos *commit* regularmente, isso leva a que estes *logs* ocupem muito espaço no *tablespace*, facto que compromete a boa eficiência da base de dados.

5.7.2. Sintaxe de Savepoints

SAVEPOINT *identifier*
[ROLLBACK [WORK] TO [SAVEPOINT] *identifier* RELEASE
| SAVEPOINT *identifier*]

· **SAVEPOINT *identifier*** – Define um ponto de segurança com o nome *identifier*. Caso o nome já exista, o *savepoint* mais antigo é substituído por este novo.

· **ROLLBACK [WORK] TO [SAVEPOINT] *identifier* –** permite fazer um *rollback* para o ponto indicado por *identifier*, sem que termine a transação actual. As alterações feitas pela transação após o *savepoint* são marcadas como não-concluidas, mas os *locks* são mantidos.

· **RELEASE SAVEPOINT *identifier*** – permite eliminar um dado *savepoint* anteriormente criado do conjunto atual de *savepoint* dessa transação. Ao proceder à eliminação, nenhum *commit* ou *rollback* é feito.

Todos os *savepoints* de uma transação são apagados quando esta fazer *commit.*

6. Suporte para Bases de Dados Distribuídas

O nosso trabalho é sobre o *engine InnoDB*, este em si não tem qualquer suporte para bases de dados distribuídas, mas sendo suportado pelo MySQL de forma a lidar com os diferentes tipos de *store engine.*

7. Outras características do sistema estudado

7.1. Web Semântica

Neste subcapítulo vamos abordar o tema Web Semântica com foco na representação de dados RDF num Sistema Gerenciador de Banco de Dados Relacional (RDBM -> Relational database management) utilizando Jena SBD.
Para tal precisamos de ter instalados os seguintes pacotes:

- MySQL
- MySQLWorkBench
- MySQL Connector/J

E ainda fazer o download da SDB do site Apache Jena.

Em seguida deve ter que definir as variáveis de ambiente no caso de ser Linux são as seguintes:

```
export SDBROOT=/path/to/SDB-1.3.4
export PATH=$SDBROOT/bin:$PATH
export SDB_USER=root
export SDB_PASSWORD=root
export SDB_JDBC=/path/to/mysql-connector-java-5.1.18/mysql-connector-java-5.1.18-bin.jar
```

Agora já pode utilizar a SDB framework em Java.

7.2. Segurança e Multi-utilizador

Em termos de segurança o MySQL utiliza para conexões, queries e para quaisquer operações feitas pelo utilizador umas *Listas de controle de acessos* (ACLs -> Access Control Lists) que é uma lista que tem os utilizadores que tem acesso a certos serviços expecificos.

Suporta também entre as conexões entre clientes e servidores um *Protocolo de Camada de Sockets Segura* (SSL -> Secure Sockets Layer) que é um protocolo criptográfico que confere segurança na comunicação.

No que toca a vários utilizadores tem Row-level locking que consiste numa técnica que bloqueia uma linha para escrita para prevenir outros utilizadores de acederem a informação que está a ser actualizada logo que aumenta a concorrência. Isto é utilizado ao nível das tabelas para suportar simultâneas escritas por vários utilizadores (na Engine InnoDB) .

A utilização de vários utilizadores é considerada muitas vezes como de não totalmente segura mas isso de facto não é verdade porque vários clientes podem fazer update e obter o valor (ou valores) relativo ao seu cliente sem alterar o valor dos outros clientes.

Todo o software MySQL tem multi-threaded, é tambem bastante rápido e robusto.

8. MySQL - InnoDB VS Oracle

- A base de dados do Oracle também é armazenada em várias *tablespaces,* tal como acontece no InnoDB.
- Tem *Database Buffer Cache.* Os buffers na cache estão organizados em duas listas:
    - Write list
    - LRU - least recently used list

  A lista de escrita contém *dirty* buffers, que contém dados que já foram modificados mas ainda não foram escritos no disco. A lista LRU contém buffers livres, *pinned* buffers e *dirty* buffers que ainda não foram movidos para a lista de escrita. Buffers livres não contêm dados importantes e estão prontos a ser usados. Pinned buffers estão a ser acedidos de momento.

  Quando um buffer é acedido, é movido para o fim da lista LRU que contém os MRU. À medida que mais buffers são movidos para a parte MRU da lista LRU , os dirty buffers "envelhecem" para o lado oposto. É semelhante ao InnoDB do MySQL, que também se baseia no LRU.
- O Oracle também tem um mecanismo de partições.
- Ao contrário do InnoDB, o Oracle permite multitable clustering, o que se torna numa vantagem. Quem quiser armazenar diferentes relações no mesmo ficheiro para minimizar o acesso ao disco, deve escolher o Oracle, face ao MySQL usando InnoDB.
- Os tuplos são guardados num Heap, ou seja, sem uma ordem específica.
- São permitidos registos de tamanho variável, tal como no MySQL.
- O Oracle, em comparação com o MySQL com InnoDB, é mais completo no que toca a ferramentas de indexação e hashing.
- O Oracle usa árvores B+, ao passo que o InnoDB do MySQL usa B. Em Gestão de Sistemas de Bases de Dados, uma B+ é mais vantajosa.
- Tem apenas que se ter em consideração as diferenças face ao standard SQL entre as duas plataformas.
- O Oracle tem uma forma diferente, mais intuitiva, de mostrar o plano de execução.
- Em termos de optimização, o InnoDB utiliza muitas estratégias semelhantes de simplificação de consultas.
- Os dois tem suporte para transações com propriedades ACID.

- O InnoBD tem quatro níveis de isolamento (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE) enquanto o oracle tem apenas 3 niveis de isolamento (READ COMMITTED, SERIALIZABLE, READ ONLY). Por defeito funcionam os dois em READ COMMITTED.
- O InnoBD só suporta protocolos baseados em locks (pessimista) enquanto o Oracle suporta protocolos baseados em locks e em timestamp (otimista). Logo o InnoBD apenas suporta a versão pessimista que para quando exitem potenciais conflitos, até que esse conflito deixe de existir. No caso do Oracle se utilizarmos o protocolo otimista que não para em potenciais conflitos e se algo correr mal pode sempre fazer rollback.
- Embora o InnoBD não suporte Nested Transactions, isso não traz grandes consequências porque isso pode ser substituido por savepoints que são mais baixo nível**.** Sendo que o Oracle suporta estes dois mecanismos.
- Sendo que o InnoBD não suporta qualquer tipo de base dados distribuídas. Neste caso parece claro que caso se queira utilizar um sistema que tenha suporte para base de dados distribuídas o Oracle é uma boa opção.

9. Referências / Bibliografia

Manual de Referência do MySQL:

http://dev.mysql.com/doc/refman/5.7/en/innodb-multiple-tablespaces.html
http://dev.mysql.com/doc/refman/5.7/en/innodb-file-space.html
http://dev.mysql.com/doc/refman/5.7/en/innodb-row-format.html
http://dev.mysql.com/doc/refman/5.7/en/storage-engine-setting.html
http://dev.mysql.com/doc/refman/5.7/en/innodb-buffer-pool.html
http://dev.mysql.com/doc/refman/5.7/en/innodb-default-se.html
http://dev.mysql.com/doc/refman/5.7/en/partitioning.html
http://dev.mysql.com/doc/refman/5.7/en

Manuais de Referência do Oracle:

http://www.orafaq.com/wiki/Heap-organized_table
http://docs.oracle.com/cd/B28359_01/server.111/b28318/physical.htm#CNCPT003
http://docs.oracle.com/cd/B28359_01/server.111/b28318/memory.htm#CNCPT1223
http://www.oracle.com/br/products/database/options/partitioning/overview/index.html
https://novatec.com.br/livros/mysqlcompleto/capitulo8575221035.pdf