Sistema de Base de Dados

Relatório

Autores:
Andreia Marques nº42184
Mara Felismino nº41890
Tânia Antunes nº41836

Professor:
José Júlio Alferes

1 de Junho de 2014

Conteúdo

1 Introdução **4**
- 1.1 Estrutura . . . 4
- 1.2 História . . . 5
- 1.3 Instalação . . . 5
  - 1.3.1 Windows . . . 5

2 Armazenamento e *File Structure* **6**
- 2.1 *Storage Engines* . . . 6
- 2.2 *Buffer Management* . . . 7
  - 2.2.1 Algoritmo LRU do InnoDB em detalhe: . . . 8
  - 2.2.2 Configurar a *buffer pool*: . . . 8
- 2.3 Ficheiros no sistema operativo . . . 9
- 2.4 Partições . . . 10
  - 2.4.1 *RANGE Partitioning* (Partição por intervalo) . . . 10
  - 2.4.2 *LIST Partitioning* (Partição em lista) . . . 10
  - 2.4.3 *HASH Partitioning* (Partição em *Hash*) . . . 10
  - 2.4.4 *KEY Partitioning* (Partição em chave) . . . 10
  - 2.4.5 *COLUMNS Partitioning* (Partição em colunas) . . . 11
  - 2.4.6 *Subpartitioning* (Sub-particionamento) . . . 11
  - 2.4.7 Vantagens do uso de partições . . . 11
- 2.5 Organização dos tuplos . . . 12
- 2.6 Comparação com o Oracle . . . 12

3 Indexação e Hashing **13**
- 3.1 Índices Ordenados . . . 13
  - 3.1.1 *Clustered Index* . . . 13
  - 3.1.2 *Secondary Index* . . . 13
  - 3.1.3 *Fulltext Index* . . . 14
  - 3.1.4 *Multiple-Column Index* . . . 14
- 3.2 Índices de *Hash* . . . 14
  - 3.2.1 *Adaptive Hash Index* . . . 15
- 3.3 Estruturas Inconsistentes . . . 15
- 3.4 Comparação com o Oracle . . . 15

4 Processamento e Optimização de Perguntas **16**
- 4.1 Optimização através de Índices . . . 17
- 4.2 Optimização de Operações Básicas . . . 17
  - 4.2.1 *WHERE* . . . 17

4.2.2 *HAVING* . . . 17
4.2.3 *DISTINCT* . . . 18
4.2.4 *JOIN* . . . 18
4.2.5 *ORDER BY* . . . 20
4.2.6 *GROUP BY* . . . 20
4.3 Optimização de Operações Complexas . . . 21
4.3.1 *Materialization* . . . 21
4.4 Estimativas . . . 21
4.5 Planos de Execução . . . 22
4.6 Comparação com o Oracle . . . 22

5 Gestão de Transacções e Controlo de Concorrência 23
5.1 Transacções . . . 23
5.1.1 *Nested Transactions* . . . 23
5.1.2 Transacções de longa duração . . . 23
5.2 Níveis de Isolamento . . . 24
5.2.1 Tipos de Isolamento . . . 25
5.2.2 *Savepoints* . . . 25
5.3 Protocolo de Isolamento . . . 26
5.3.1 *Deadlocks* . . . 26
5.3.2 Níveis de Granularidade . . . 28
5.4 Comparação com o Oracle . . . 29

6 Suporte para Bases de Dados Distribuídas 31
6.1 MySQL *Cluster* . . . 31
6.2 *Distributed Replicated Block Device (DRBD)* . . . 31
6.3 Fragmentação em *Federated* . . . 32
6.4 MySQL *Replication* . . . 33
6.5 Transacções Distribuídas (XA) . . . 34
6.6 Comparação com o Oracle . . . 35

7 Outras Características do MySQL 36
7.1 Suporte para WEB . . . 36
7.1.1 Suporte em XML . . . 36
7.1.2 Suporte em XPath . . . 37
7.2 *Stored Procedures* . . . 37
7.2.1 Criar *Procedures* . . . 37
7.2.2 Alterar uma *Procedure* . . . 38
7.2.3 Eliminar uma *Procedure* . . . 38
7.3 *Triggers* . . . 38
7.3.1 Criação de um *Trigger* . . . 38
7.3.2 Eliminação de um *Trigger* . . . 38
7.4 Tipos de Dados . . . 39
7.4.1 Numéricos . . . 39
7.4.2 *String* . . . 39
7.4.3 Data e Tempo . . . 40
7.5 Utilizadores e Segurança . . . 40
7.5.1 Criar Utilizador . . . 41
7.5.2 GRANT . . . 41
7.5.3 REVOKE . . . 42

7.6 MySQL Workbench 43

8 Conclusão **44**

Capítulo 1

Introdução

O sistema que decidimos escolher foi o MySQL v.5.6.
Escolhemos focar-nos sobre o MySQL porque nos dias de hoje é dos sistemas de gestão de base de dados, *open-source*, mais usado no mundo. Decidimos então, analisar neste trabalho o SGBD, MySQL - desenvolvido pelo MySQL AB. Visto que, como vai ser explicado mais à frente, o MySQL tem várias *storage engines* que no fundo trabalham como SGDB totalmente diferentes e independentes, tivemos de optar por uma *storage engine*. Após uma análise das várias *storage engines* existentes, acabámos por optar essencialmente pela Storage Engine InnoDB, isto porque, é a mais complexa e permite um maior leque de funcionalidades. Contudo, sempre que acharmos necessário, vamos referir as outras *storage engines* para uma maior percepção.

1.1 Estrutura

Neste trabalho irão ser abordados os seguintes tópicos: Armazenamento e *File Structure*, onde será explicado em maior detalhe quais as *Storage Engines*, como é gerido o *Buffer*, se o MySQL se baseia no *file system* do sistema operativo ou implementa o seu, quais os tipos de partições e como são organizados os tuplos; Indexação e Hashing, que terá como secções os tipos de índice (ordenados ou de hash), e as estruturas inconsistentes; Processamento e Optimização de Perguntas, onde serão explicados os vários tipos de optimização (através de índices, de operações básicas e de operações completas), as estimativas e os planos de execução; Gestão de Transacções e Controlo de Concorrência, que irão ser explicados a maior rigor as transacções, os níveis de isolamento, e o protocolo de isolamento que MySQL implementa; Suporte para Base de Dados Distribuídas, onde serão abordados o MySQL *cluster*, o *Distributed Replicated Block Device*, a Fragmentação existente na *storage engine* Federated, a replicação MySQL, e transacções distribuídas; e por último serão abordadas outras características importantes do MySQL. Mas antes de serem abordados os capítulos e secções acima mencionados, irá ser exposto a história do MySQL, bem como o procedimento à instalação do MySQL.

1.2 História

A sigla MySQL significa *My Structured Query Language* e o projecto tornou o seu código público sobre a licença GNU. O MySQL nasceu, em 1994, da necessidade que os programadores do ISAM estavam a ter, ao tentarem conectar uma base de dados mSQL com as tabelas do ISAM. Contudo, como o MySQL não era suficientemente rápido e flexível para as necessidades dos programadores, foi criada uma nova interface de SQL – o MySQL. Alguns dos seus marcos históricos foram:

1994 - Início do seu desenvolvimento, por parte de Michael Widenius e David Axmark.

1995 – Primeira publicação interna, a 23 de Maio de 1995.

1998 – Publicação de uma versão para o Windows, a 8 de Janeiro de 1998.

2001 – Publicação, em Janeiro de 2001, da versão 3.23, em beta desde Junho de 2000.

2003 – Publicação da versão 4.0, em beta desde Março de 2003;
– Jyoti adopta o MySQL v4.01.

2004 – Publicação da versão 4.1, em beta desde Junho de 2004. R-Tree e B-Tree, adicionadas ao sistema.

2005 – Publicação da versão 5.0, em beta desde Março de 2005. Cursores, procedimentos, *triggers*, *views* e transacções XA, adicionadas ao sistema.

2008 – O MySQL AB é adquirido pela Sun Microsystems;
– Publicação da versão 5.1, a 27 de Novembro de 2008. *Event scheduler*, particionamento, plugin API, replicação baseada em tuplos e tabelas de *server-log*, adicionadas ao sistema.

O MySQL hoje suporta *Unicode*, *Full Text Indexes*, replicação, Hot Backup, GIS OLAP e muitos outros recursos de banco de dados.A sua natureza de código aberto levou também a um crescente número de *plugins* e ferramentas disponíveis.

1.3 Instalação

A instalação do sistema MySQl pode ser feita em várias plataformas (Unix, Windows). A instalação é simples e pode-se recorrer a uma interface de graça para tal. Para o desenvolvimento do trabalho prático foi instalado o sistema a ser estudado em ambiente Windows.

1.3.1 Windows

Os novos utilizadores de MySQL podem instalá-lo a partir do Instalador Automatizado. Este instalador possui um assistente de instalação e configuração que guia o utilizador durante todo o processo. Os assistentes estão desenhados para instalar e configurar o MySQL, de modo a fique pronto a ser executado imediatamente.

Capítulo 2

Armazenamento e *File Structure*

Uma das características do MySQL é suportar vários *storage engines*, que tratam de toda a gestão do armazenamento de dados, fornecendo uma camada de abstracção que evita o contacto do utilizador com o sistema de ficheiros nativo do Sistema Operativo. O MySQL suporta também um mecanismo de *buffer* e *cache* próprio. Foi escolhido como foco principal a *Storage Engine* InnoDB [1].

2.1 *Storage Engines*

Tendo em conta as diferenças encontradas entre os vários *storage engines* do MySQL, começaremos com uma breve descrição destes, continuando então no resto das secções com uma análise detalhado de como o InnoDB gere o armazenamento de dados do MySQL.
Os *engines* existentes e as respectivas características são:

MyISAM: é o mais usado e é a opção por defeito do MySQL. É uma *engine* que tem como grande vantagem a rapidez de acesso a disco, tanto escrita como leitura.

InnoDB: *transaction-safe* e segue ACID. Permite fazer *commit*, *rollback* e *crash-recovery* para proteger os dados; Tem como principal vantagem a suportar restrições de integridade para o uso de chaves externa.

Merge: disponibiliza ao MySQL a operação de agrupar logicamente, tabelas MyISAM idênticas, de forma a ficarem referenciadas como uma só;

Memory (também conhecida como Heap): cria tabelas com o conteúdo que está em memória;

Example: é do interesse dos programadores de *storage engines*, isto porque, podemos criar tabelas usando-o, mas nenhuns tuplos podem ser criados ou lidos, servindo apenas para ilustrar aos programadores como se inicia o desenvolvimento de uma *storage engine*.

Federated: dá a possibilidade de fundir logicamente um grupo de servidores MySQL, num só; Boa solução para base de dados distribuídas visto que permite ligar vários servidores MySQL separados para criar uma única base de dados lógica.

Arquive: mais indicado para armazenar e adquirir, largos blocos de dados que poucas vezes são usados;

CSV: armazena os dados em ficheiros de dados do tipo .csv. Permite fácil troca de informação com outros processadores de texto;

Blackhole: recebe dados, mas não os armazena. Este mecanismo é usado em computação distribuída, onde os dados não são armazenados localmente, mas sim replicados.

BerkeleyDB: útil para utilizar transacções seguras.

NDBCLUSTER (também conhecida como NDB): foi desenhado para utilização em ambientes de computação distribuída que requerem alta redundância e disponibilidade.

Nota: A escolha de um *engine* adequado para uma determinada aplicação é importante pois a performance dessa mesma aplicação depende directamente da performance da base de dados. Assim, um programador deve pensar sempre em definir um *engine* que possa tirar o máximo partido e que permita uma melhor eficiência de acordo com a aplicação pretendida.
Para definir o *storage engine*:

```
SET storage_engine = INNODB;
```

É importante salientar que o utilizador não está restringido a utilizar o mesmo *storage engine* em todo o servidor. É possível atribuir um *storage engine* diferente para cada tabela, se assim for pretendido, da seguinte maneira:

```
CREATE TABLE t (i INT) ENGINE = INNODB;
ALTER TABLE t ENGINE = INNODB;
```

2.2 *Buffer Management*

O InnoDB possui o seu próprio *buffer* para guardar a informação e os índices na memória, denominado *"Buffer Pool"*. Este *buffer* funciona como uma lista, usando o algoritmo LRU (*Least Recently Used*), mas com uma pequena variante, utiliza o *midpoint insertion strategy*, que divide a lista em duas sub-listas:

- À cabeça da lista, uma sub-lista de blocos que foram acedidos recentemente (denominados "*young*").
- Na cauda da lista, uma sub-lista com os blocos que foram menos acedidos recentemente.

Com isto, ficam na primeira sub-lista os blocos que são constantemente usados por *queries* e na segunda sub-lista ficam os blocos menos usados e com maior probabilidade de serem removidos. [2].

2.2.1 Algoritmo LRU do InnoDB em detalhe:

- 3/8 do *buffer pool* é dedicado à segunda sub-lista (blocos menos usados);
- O centro da lista marca o ponto onde a cauda da sub-lista dos blocos mais usados coincide com a cabeça da sub-lista dos blocos menos usados;
- Quando o InnoDB lê um bloco para o *buffer*, esse bloco é inserido no ponto médio (cabeça da sub-lista dos blocos menos usados);
- Ao fazer um acesso a um bloco da sub-lista dos blocos menos usados, esse mesmo bloco é movido para a cabeça do *buffer pool* (para a cabeça da sub-lista dos blocos mais recentemente usados);
- Conforme forem feitas operações na base de dados, os blocos que não são acedidos, são movidos para a cauda da respectiva sub-lista. Eventualmente, um bloco (da sub-lista dos menos recentemente usados) que se mantenha inutilizado por um largo período de tempo, é descartado.

Através das instruções *Insert* e *Delete* o *buffer pool* guarda os dados alterados para que a escrita no disco seja bastante mais rápida.

2.2.2 Configurar a *buffer pool*:

Existem algumas variáveis do InnoDB que permitem configurar o *buffer pool* e por sua vez adaptar o algoritmo LRU.

`innodb_buffer_pool_size`

Esta variável especifica o tamanho de um *buffer pool* (aumenta o tamanho), tendo em conta que existe memória suficiente no sistema. Poderá aumentar a performance ao reduzir o número de chamadas directas ao disco.

`Innodb_buffer_pool_instances`

Quando um *buffer pool* tem uma dimensão a ultrapassar 1 gigabyte. Permite dividir o *buffer pool* em tantas regiões separadas quanto as especificadas e gere o seu próprio LRU e estruturas associadas.

`Innodb_old_blocks_pct`

Esta variável permite especificar a percentagem que o *buffer pool* usa para a sub-lista dos "velhos". Este valor tem de estar entre 5 e 95, sendo o valor padrão 37;

`Innodb_old_locks_time`

Especifica o tempo em milissegundos (ms) que um bloco inserido na lista antiga tem de esperar até poder passar para a lista nova. Se o bloco inserido for acedido novamente antes do tempo mínimo não poderá passar para a lista nova. Isto pode evitar que blocos que são lidos em pesquisas de tabelas e que nunca mais são usados, sejam inseridos na lista nova.

Resumindo, poderá dizer-se que a monitorização e os ajuste dos parâmetros do *buffer pool* do InnoDB poderão trazer grandes benefícios na performance geral do sistema.

2.3 Ficheiros no sistema operativo

O InnoDB foi planeado para suportar uma performance maximizada, quando se trata de processamento de grandes volumes de dados.Nesta *storage engine* estudada (InnoDB), todos os indicies e tabelas são armazenadas em *tablespaces* (estrutura de dados), o que pode corresponder a várias partições de disco ou a vários ficheiros, sendo que estes podem adquirir tamanhos elevados, mesmo quanto estão a funcionar por cima de um SO (limita o seu tamanho a 2GB). Por outro lado, se não for dado um valor ao tamanho, o sistema atribui um tamanho inicial de 10MB. Os *tablespaces* são organizados em páginas com um tamanho por defeito de 16KB e estas são agrupadas em grupos de 64 páginas consecutivas, ou seja, por grupos de páginas de 1MB. [3][4].
Um *tablespace* é constituído por vários segmentos que contêm:

- Dados de tabelas (se não se utilizar a opção de *tablespace* por tabela);
- Dois segmentos para cada índice - um para os nós que não são folhas na B+-Tree e outro param os que o são;
- *Data Dictionary*;
- *Rollback Segment* - contém um máximo de 128 segmentos de *rollback*;
- *Doublewrite buffer* - funciona como um segmento temporário. Antes dos dados serem escritos para as páginas, o InnoDB escreve-as neste *buffer* contido no *tablespace* e só depois é que escreve os dados na posição correcta do *tablespace*. Assim, caso haja algum erro, pode-se sempre ir buscar os novos dados ao *buffer* e copia-los para a página, durante a fase de recuperação.

À medida que um segmento cresce dentro do *tablespace*, o InnoDB reserva-lhe mais 32 páginas individualmente. Se a partir deste momento o segmento continuar a crescer, então o sistema já vai reservar um extent (64 páginas) de cada vez. Quanto à operação de delete de dados em tabelas, o InnoDB contrai o índice B+-Tree correspondente. Contudo, se esse espaço fica logo disponível para outros utilizadores ou não, depende da localização onde ficou o espaço livre. Por outro lado, se apagarmos toda a tabela, è garantido que o espaço fica logo livre para outros utilizadores. É de referir que o InnoDB guarda informação meta, o que torna impossível através de réplica, transferir as BD de um servidor para o outro. Por outro lado, se esta operação for no mesmo servidor, podemos usar o seguinte código para restaurar um *tablespace*:

```
ALTER TABLE tbl_name DISCARD TABLESPACE;
```

Agora que o *tablespace* antigo está descartado, podemos copiar o ficheiro *.ibd (InnoDB File) com o *tablespace*, para a pasta da BD e restaurá-lo com o seguinte código:

```
ALTER TABLE tbl_name IMPORT TABLESPACE;
```

2.4 Partições

Apesar de algumas *engines* não suportarem particionamento, o InnoDB é uma das *engines* que suporta este mecanismo. O sistema de partições serve essencialmente para fins de optimização. Vejamos os tipos de particionamento disponíveis, bem como a definição para cada uma delas: [5]

2.4.1 *RANGE Partitioning* (Partição por intervalo)

Este tipo de particionamento é útil quando queremos criar partições que verifiquem uma dada restrição. Ou seja cada partição vai ter todos os registos que verifiquem uma dada expressão de particionamento.

```
PARTITION BY RANGE (att1)(
PARTITION n1 VALUES LESS THAN(x),
PARTITION n2 VALUES LESS THAN(y),
PARTITION n3 VALUES LESS THAN(z)
);
```

2.4.2 *LIST Partitioning* (Partição em lista)

Semelhante à partição por *RANGE*, esta tem a diferença de em vez de ter uma expressão de restrição temos uma lista de valores para o qual queremos os registos que estão contidos nessa lista.

```
PARTITION BY LIST(att1)(
PARTITION n1 VALUES IN (a,b,c,d),
PARTITON n2 VALUES IN (e,f,g,h),
PARTITION n3 VALUES IN (i,j,k,l)
);
```

2.4.3 *HASH Partitioning* (Partição em *Hash*)

Este tipo de particionamento é diferente dos dois anteriores pois neste tipo de particionamento o utilizador não especifica uma expressão para filtrar os dados, usa sim uma expressão de particionamento e é o MySQL que se encarrega de definir a partição em que o registo se irá inserir.

```
PARTICION BY HASH(att1)
PARTITONS 4;
```

2.4.4 *KEY Partitioning* (Partição em chave)

Semelhante ao anterior, apenas com a diferença que neste caso o particionamento é feito com base na chave primária quando nenhum nome de coluna é dado.

```
PARTITION BY KEY()
PARTITIONS 4;
```

2.4.5 *COLUMNS Partitioning* (Partição em colunas)

Variantes das partições *Range* e *List*:

RANGE COLUMNS (Colunas por intervalos): Método de partição semelhante ao RANGE, com diferenças, tais como, a não aceitação de expressões, só nomes de colunas, aceita uma lista de uma ou mais colunas, é baseado em comparações entre tuplos em vez comparações entre valores escalares e não está restrito a colunas de inteiros, podendo-se usar colunas de strings, DATE e DATETIME.

```
PARTITION BY RANGE COLUMNS(a,d,c) (
PARTITION p0 VALUES LESS THAN (5,10,'ggg'),
PARTITION p1 VALUES LESS THAN (10,20,'mmmm'),
PARTITION p2 VALUES LESS THAN (15,30,'sss'),
PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE,MAXVALUE)
);
```

LIST COLUMNS (Colunas por intervalos): Método de partição semelhante ao *List*, as diferenças, tais como, a aceitação do uso de múltiplas colunas como chave e não está restrito a colunas de inteiros, podendo-se usar colunas de strings, DATE e DATETIME.

```
PARTITION BY LIST COLUMNS(city) (
PARTITION pRegion 1 VALUES IN('Oskarshamn','Hogsby','Monsteras'),
PARTITION pRegion 2 VALUES IN('Vimmerby','Hultsfred','Vastervik'),
PARTITION pRegion 3 VALUES IN('Nassjo','Eksjo','Vetlanda'),
PARTITION pRegion 4 VALUES IN('Uppvidinge','Alvesta','Vaxjo')
);
```

2.4.6 *Subpartitioning* (Sub-particionamento)

Também designado por particionamento composto, resume-se na divisão de cada partição em mais partições. Este particionamento pode ser feito sobre partições do tipo *RANGE* ou *LIST*, e as sub-partições têm de ser do tipo *KEY* ou *HASH*.

```
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
);.
```

2.4.7 Vantagens do uso de partições

As vantagens do uso de partições assentam essencialmente em 3 aspectos, no armazenar mais dados numa determinada tabela, na remoção e inserção facilitada, bastando remover/inserir a(s) partição(ões) e na optimização nas consultas da base de dados, podendo-se fazer os acessos somente as partições certas, excluindo dados que sejam desnecessários.

2.5 Organização dos tuplos

Tuplos (linhas de uma tabela onde estão os dados) são guardados em páginas de um *tablespace*. O tamanho máximo que cada tuplo pode ter é de aproximadamente metade de uma página (8000 bytes), não contando com objectos variáveis (VARBINARY, VARCHAR, BLOB e TEXT). Colunas do tipo LONGBLOB e LONGTEXT têm de ter menos 4GB e, o tamanho total da linha, incluindo colunas de tamanho variável, tem de ser menor que 4GB.
Para o InnoDB não perder a localização desses campos que foram para páginas externas, armazena os primeiros 768 bytes localmente na página interna e os restantes são armazenados em *overow-pages*. Esses 768 bytes armazenados internamente, são acompanhados por 20 bytes que guardam informação relativa ao verdadeiro tamanho desse campo e apontam para a *overow-page*, onde está o resto do valor desse campo. Os tuplos no InnoDB estão organizados sob a forma de *clustered index*, ordenados pela chave primária. Este *engine* não suporta *multitable clustering*. [6][7][8]

2.6 Comparação com o Oracle

O Oracle tem o seu próprio *buffer manager* no SGA (System Global Area) que tem uma lista LRU que funciona de forma idêntica à lista do *Buffer Pool*. A diferença é que o Oracle tem mais componentes (*buffers* e *pools*) no SGA que o *Buffer pool* do InnoDB. As tabelas no Oracle ao contrário do InnoDB são organizadas em heap, permitindo também a organização por índice, como o InnoDB. De referir, que o sistema estudado não permite a utilização de *Multitable Clustering*, contrariamente ao que se verifica em Oracle. Permite o particionamento de tabelas tal como o MySQL com a diferença que não tem o tipo *Key*, permitindo tal como no MySQL.

Capítulo 3

Indexação e Hashing

No InnoDB os dados de cada tabela são organizados em páginas, estando estas páginas organizadas na forma de um indice *B-tree*: os nós contêm a chave primária e folhas os valores das colunas.
A indexação é uma estrutura de dados que tem como principal objectivo melhorar a performance de operações sobre uma tabela, ou seja, tornar o acesso aos dados muito mais rápido e eficiente. Esta estrutura é particularmente útil quando o resultado pretendido, é um conjunto de registos em que os seus atributos:
- Contém um valor específico;
- Estão compreendidos entre um intervalo de valores.

3.1 Índices Ordenados

3.1.1 *Clustered Index*

Todas as tabelas do InnoDB têm um *clustered index*. Aquando da criação de uma tabela, caso seja definida uma chave primária, o InnoDB cria um índice ordenado por essa chave, caso contrário, o MySQL procura por um índice *unique* onde exista uma coluna (ou conjunto de colunas) não nulas para que o InnoDB as possa utilizar como índice. Em último caso, se não ocorrer nenhuma das situações acima, o InnoDB gera internamente um índice ordenado de *ROW ID's*. [6]

```
CREATE TABLE T1(
A INT PRIMARY KEY,
B INT,
C CHAR(1)
) ENGINE=InnoDB;
```

A tabela acima teria um *clustered index* sobre o atributo "A".

3.1.2 *Secondary Index*

O *secondary index* pode ser criado sobre um subconjunto de colunas de uma tabela, sendo que este contem sempre as colunas referentes quer à chave primária quer às especificadas para criação do índice.

Tem como principal vantagem encontrar tuplos cujos valores de determinados atributos satisfaçam uma certa condição, sendo que estes atributos não fazem parte da chave primária. [6]

```
CREATE TABLE T2(
A INT PRIMARY KEY,
B INT,
C CHAR(1),
INDEX (B)
) ENGINE=InnoDB;
```

A tabela acima teria um *secondary index* no atributo "B".

3.1.3 *Fulltext Index*

O *Fulltext Index* ajuda a aumentar a eficiência de perguntas que envolvem colunas baseadas em texto e palavras deste mesmo, pelo que este só pode ser criado em colunas do tipo CHAR, VARCHAR e TEXT. [9]

```
CREATE TABLE T3(
A INT PRIMARY KEY,
B VARCHAR(200),
C TEXT,
FULLTEXT (B)
) ENGINE=InnoDB;
```

A tabela acima teria um *fulltext index* no atributo "B".

3.1.4 *Multiple-Column Index*

O *multiple-column index* é bastante eficiente para perguntas em que seja necessário percorrer mais do que uma coluna, isto é, perguntas em que se verificam determinadas condições sobre o mesmo conjunto de colunas, pelo que é bastante importante a ordem, pela qual são passadas as colunas, na criação do índice. [10]

```
CREATE TABLE T4(
A INT PRIMARY KEY,
B CHAR(30),
C CHAR(30),
INDEX (B,C)
) ENGINE=InnoDB;
```

A tabela acima teria um *multiple-column* nos atributos "B" e "C", por esta ordem respectivamente.

3.2 Índices de *Hash*

Apesar do InnoDB não possuir a estrutura de dados *hash index*, suporta *hashing*. Já foi visto que é utilizado nas partições para permitir que a distribuição dos dados da tabela seja equilibrada de acordo com o número de partições.
É possível fazer *hashing* a mais do que um atributo, mas torna o processamento de *queries* mais lento.

3.2.1 *Adaptive Hash Index*

O *adaptive hash index* é uma implementação de *hash index* em memória, sendo utilizado quando a tabela, referênciada nas pesquisas, cabe toda em memória.
O InnoDB tem um mecanismo para monitorizar as pesquisas num índice, ou seja, se o sistema achar benéfico a construção de um índice disperso para tais consultas, este é criado automaticamente. O índice é construído baseando-se nos índices *B-Tree* já existentes da tabela. [11]
Para desactivar esta funcionalidade, é necessário executar no arranque, o seguinte comando:

```
--skip-innodb_adaptive_hash_index
```

3.3 Estruturas Inconsistentes

O MySQL possui uma variável denominada *autocommit* (que, por omissão, está activa), para fazer a gestão das alterações nos dados, ou seja, para garantir que todas as alterações aos dados são guardadas em disco.
Para desactivar este modo é necessário introduzir:

```
SET autocommit=0;
```

Depois de desactivado, o utilizador é responsável por guardar as respectivas alterações, bastando para isso, utilizar o comando *commit*. Aquando do uso deste, todas as restrições impostas sobre os dados, são verificadas, de forma a garantir a consistência. [12]

3.4 Comparação com o Oracle

Enquanto que o InnoDB apenas utiliza estruturas de dados *B-Tree*, o Oracle utiliza, para além desta, a estrutura *Bitmap*.
O facto da Oracle utilizar a estrutura *Bitmap* é uma grande vantagem visto esta estrutura ser bastante eficiente, por exemplo, para contagens de tuplos.
Em termos de índices de *hashing*, também não são suportados pelo Oracle, apesar deste ter um mecanismo para simular, através de partições, "*limited static hash*".

Capítulo 4

Processamento e Optimização de Perguntas

A linguagem, que está na base de todas as acções referentes à base de dados, é a álgebra relacional. Esta é uma linguagem de interrogação, onde as perguntas são feitas de uma forma *standard*, não tendo quaisquer tipos de restrições de como deva ser "representada/escrita" ou, até mesmo, processada. Para assegurar uma execução mais rápida e eficiente, é necessário recorrer a certas alterações na representação/escrita e/ou processamento desta mesma pergunta (através de diversos algoritmos). [13]

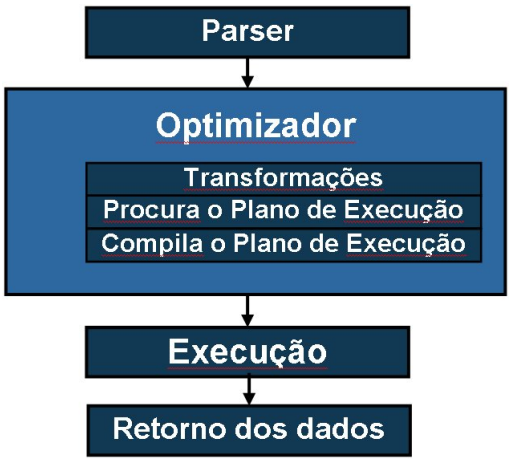

Figura 4.1: Etapas de Execução

Parser : O MySQL faz a leitura do comando SQL introduzido, de seguida converte para um formato interno binário e envia-o para o optimizador.

Optimizador : O optimizador decide a ordem de leitura das tabelas, qual o índice a ser utilizado (caso exista) e o tipo de leitura que será realizada na

tabela (algoritmos que serão utilizados na pesquisa dos dados). Decisões estas são baseadas em estatísticas armazenadas pelo próprio servidor.

Execução e Retorno de Dados : Depois de escolhidas as opções acima descritas, o MySQL executa e retorna os respectivos dados da consulta.

4.1 Optimização através de Índices

Um índice é uma tabela de pesquisa (muito mais pequena do que a tabela para o qual está construído) que permite obter a posição, na tabela original, dos registos com os dados pretendidos. Desta forma, evita-se o percorrer de toda a tabela (*full table scan*).
Sendo estes índices armazenados em *B-Tree*, o tempo de consulta é reduz bastante visto a iteração sobre esta estrutura ser muito rápida e fácil.
Para tirar vantagem destes índices para optimização, é necessário que estes sejam escolhidos e construídos correctamente. São especialmente importantes para as perguntas que referenciam várias tabelas, quer através da operação de junção, quer através de chaves estrangeiras.
Com o auxílio do comando *EXPLAIN* (explicado na secção "Planos de Execução") é possível descobrir quais tabelas devem conter índices. [14]

4.2 Optimização de Operações Básicas

4.2.1 *WHERE*

Nas consultas em que esta cláusula está presente, é necessário ter uma especial atenção às condições nela impostas. Apesar do MySQL optimizar automaticamente muitas destas condições, pode-se evitar tal, tendo como consequência uma melhor performance.
A optimização desta cláusula é feita através de várias transformações e operações, como por exemplo, a remoção de parêntesis desnecessários, a remoção de expressões constantes, a substituição de valores por constantes, entre outros. [15]

```
((a AND b) AND c OR (((a AND b) AND (c AND d))))
```

Através da optimização obteríamos:

```
(a AND b AND c) OR (a AND b AND c AND d)
```

Existe ainda, a opção *sql_small_result* que pode ser utilizada, nos casos em que o resultado é um conjuntos de poucos tuplos, para transmitir ao MySQL que deve utilizar como auxilio uma tabela temporária em memória, garantindo uma redução no custo temporal para o processamento da cláusula. [16]

4.2.2 *HAVING*

A cláusula *HAVING* é agregada à cláusula *WHERE* caso não exista uma função de agregação, ou uma expressão de agrupamento. [16]

4.2.3 *DISTINCT*

Existem muitos casos em que a cláusula *DISTINCT* pode ser considerada como um *GROUP BY* (através de equivalência de perguntas, como demonstrado no exemplo abaixo), podendo então as optimizações do *GROUP BY* ser aplicadas ao *DISTINCT*.

```
SELECT DISTINCT c1, c2, c3
FROM t1
WHERE c1 > const;
```

```
SELECT c1, c2, c3
FROM t1
WHERE c1 > const
GROUP BY c1, c2, c3;
```

Outro optimização é nos casos em que não são projectados atributos de todas as tabelas especificadas na pergunta:

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

No exemplo acima, apenas são projectados atributos da tabela t1, pelo que o MySQL põe fim à pesquisa em t2 assim que a cláusula *WHERE* seja satisfeita pela primeira vez. [17]

4.2.4 *JOIN*

O MySQL optimiza também as operações básicas de junção. Este utiliza o algoritmo de *Nested-Loop Join* ou variações deste (*Block Nested-Loop Join*) para fazer as junções. No entanto, é possível optimizar as próprias junções em si.

LEFT-JOIN e RIGHT-JOIN

Uma das funcionalidades do *join optimizer* do MySQL é calcular a ordem pela qual as tabelas devem fazer a junção. Se esta ordem satisfizer o conjunto de "passos/regras" (demonstrados(as) mais abaixo), o trabalho do *join optimizer* será bastante mais reduzido, obtendo-se uma execução muito mais rápida. [18] Considerando a seguinte junção:

```
A LEFT JOIN B join_condition
```

- A tabela B depende da tabela A e de todas as outras tabelas de que a tabela A depende;
- A tabela A depende de todas as tabelas que são utilizadas na *join_ condition*, com excepção da tabela B;
- A *join_ condition* é utilizada para escolher para o resultado, quais os tuplos da tabela B a aparecerem, ou seja, os tuplos onde a cláusula *WHERE* não é aplicada;

- Se existir um tuplo na tabela A que satisfaça a condição da cláusula *WHERE*, mas não existir nenhum tuplo em B que também satisfaça tal condição, então é gerado um tuplo em B, contendo *null* em todos os seus atributos;
- Todas as optimizações, explicadas ao longo do capítulo, são aplicáveis.

Nota: Aplicam-se os mesmos passos acima no caso de ser uma junção RIGHT JOIN, trocando apenas a ordem das tabelas.

OUTER JOIN

As expressões de *Right Outer Join* são convertidas, automaticamente, em expressões equivalentes [19], utilizando apenas os operadores *Left Join*:

```
(T1, ...) RIGHT JOIN (T2, ...) ON P(T1, ..., T2, ...)
=
(T2, ...) LEFT JOIN (T1, ...) ON P(T1, ..., T2, ...)
```

Adicionalmente, todas as operações *inner join* da forma:

```
T1 INNER JOIN T2 ON P(T1, T2)
```

São, também, substituídas pela lista:

```
T1, T2, P(T1, T2)
```

Nested Join Optimization

O primeiro passo da optimização deste algoritmo é a troca de qualquer lista de tabelas por referências:

```
SELECT *
FROM t1 LEFT JOIN (t2 , t3 , t4)
ON (t2.a = t1.a AND t3.b = t1.b AND t4.c = t1.c)
```

A pergunta acima é transformada em:

```
SELECT *
FROM t1 LEFT JOIN (t2 CROSS JOIN t3 CROSS JOIN t4)
ON (t2.a = t1.a AND t3.b = t1.b AND t4.c = t1.c)
```

Outro passo importante é o remover de parêntesis desnecessários, sendo que não deverão ser removidos:

- Ao operador que está à direita de uma operação de *LEFT OUTER JOIN*;
- Ao operador que está à esquerda de uma operação de *RIGHT OUTER JOIN*.

O remover de parêntesis tem de ser muito cuidadoso para que não exista afectação do resultado. Resumidamente, não se deve remover os parêntesis à tabela que seria utilizada como *inner*, aquando da execução do algoritmo.
No caso de expressões que envolvam apenas *inner joins*, os parêntesis podem ser removidos, sendo possível também, a avaliação da expressão quer da direita para a esquerda, quer no sentido inverso. [20]

4.2.5 *ORDER BY*

Existem certas circunstâncias em que o MySQL pode utilizar um índice para satisfazer directamente esta cláusula, sem ter de fazer mais nenhum tipo de ordenação extra. Nos casos em que tal não é possível, os índices podem ser utilizados para obter uma ordenação parcial ou, até mesmo, para apenas encontrar os tuplos que satisfaçam a cláusula *WHERE*. [21]

Filesort Optimization

Para a ordenação e obtenção de tuplos, este algoritmo apenas utiliza as colunas indicadas como argumento na cláusula *ORDER BY*. Para optimização deste algoritmo, foi criada uma nova versão, obrigando a que fossem utilizadas todas as colunas indicadas na expressão e não apenas as que constavam na cláusula. Desta forma evitou-se a leitura repetida de uma mesma linha, mas surgiu um novo problema: produção de tuplos demasiado grandes que aumentavam a probabilidade do seu processamento intermédio não caber no *buffer*.
Pode nem sempre ser vantajoso a utilização da nova versão, pelo que é o optimizador do MySQL, que fica responsável pela escolha de qual deva ser utilizado. [21]

4.2.6 *GROUP BY*

Por norma, satisfaz-se esta cláusula através de *full scan table*, criando uma tabela temporária com todos os tuplos de cada grupo, colocados consecutivamente. Aquando de operações de agregação, esta nova tabela temporária, seria utilizada. Sendo esta prática muito pouco eficiente, o MySQL tem dois algoritmos de processamento do *GROUP BY*, tendo eles uma pré-condição de aplicação: Todos os atributos referidos no *GROUP BY* têm de fazer parte do mesmo índice. [22]

Loose index scan Algorithm

A maneira mais eficiente para processar este operador é a utilizar índices para a obtenção directa das colunas do agrupamento. Através desta propriedade, o MySQL tira proveito do facto de alguns tipos de índices estarem ordenados por chaves. É, assim, possível procurar grupos num índice sem que seja necessário a consideração de todas as chaves que satisfaçam as condições da cláusula WHERE (daí o nome do algoritmo). Quando não existe a cláusula *WHERE*, são lidas tantas chaves quanto o número de grupos (que ainda assim, podem ser bastante menos do que o número total de chaves).[22] No entanto, esta técnica tem as seguintes restrições:

1. A pergunta apenas envolve uma tabela;
2. O operador apenas utiliza colunas do prefixo mais à esquerda do índice;
3. As únicas funções de agregação possíveis na cláusula *SELECT* são *MIN()* e *MAX()*, sendo ambas referentes à mesma coluna. Esta coluna deverá pertencer ao índice e ao grupo de colunas presentes na cláusula *GROUP BY*;

4. As restantes partes do índice, não referidas no *GROUP BY*, só podem aparecer na pergunta através de igualdades com constantes;

5. As colunas que pertencem ao índice têm de estar completamente indexadas. (No exemplo abaixo, a coluna c1 não está completamente indexada:

```
c1 varchar(20),
index(c1(10) )
```

Tight Index Scan Algorithm

Aplica-se este algoritmo quando todas as chaves, que satisfaçam as condições da cláusula *WHERE*, têm de ser lidas ou quando é necessário que seja efectuada uma leitura sobre todos os índices desta mesma cláusula, ou seja, quando as restrições do algoritmo acima não são aplicáveis.Continua a optimizar no facto de evitar as tabelas temporárias.
O algoritmo pode ser um *full index scan* ou um *range index scan*, dependendo das condições da pergunta. Caso não existam condições de *range* na cláusula *WHERE*, o algoritmo realiza um *full index scan*. A operação de agrupamento só é feita após todas as chaves, que satisfaçam esta cláusula, sejam encontradas. [22]

4.3 Optimização de Operações Complexas

4.3.1 *Materialization*

O optimizador do MySQL utiliza a materialização como estratégia de optimização de *subquerys*, escrevendo os seus resultados numa nova tabela temporária. Uma vez criada, esta terá um índice associado para um aumento na eficiência e velocidade da execução de futuras perguntas. O rescrever da *subquery* é também evitado. Sempre que o seu tamanho o permita, estas novas tabelas são guardadas em memória, caso contrário são guardadas em disco. [23]
Para que esta estratégia seja utilizada [24], a sua *flag* tem de ser activada:

```
SET [GLOBAL | SESSION] optimizer_switch = 'materialization = on'
```

4.4 Estimativas

Na maioria dos casos, a estimativa do custo de execução de uma pergunta é baseada no número de *disk seeks* efectuados. Enquanto que para tabelas pequenas é possível encontrar um registo com apenas um *seek*, para tabelas grandes, é necessário estimar o número de *seeks*. Para isso, o MySQL utiliza o auxílio de índices *B-Tree*, na seguinte fórmula:

$$nSeeks = \frac{log(numero_{tuplos})}{log(\frac{tamanho_{blocoIndice}}{3} \times \frac{2}{tamanho_{indice}+tamanho_{apontadorDados}})} + 1$$

4.5 Planos de Execução

Todo o processo de optimização é baseado em heurísticas, pelo que nem sempre é percorrido o melhor caminho. A diferença entre um "bom" ou um "mau" plano pode ser de uma magnitude enorme (ou seja, diferenças de compilação de segundos para horas ou dias).
Grande parte dos optimizadores tentam descobrir todas as combinações de planos possíveis, com o objectivo de encontrar o melhor. No entanto, esta prática pode ser bastante dispendiosa pois quando a pergunta envolve um número elevado de tabelas referenciadas (por norma mais do que 10), ao calcular todos os planos possíveis, o tempo de resposta seria ainda mais longo do que se fosse aplicado qualquer um dos primeiros planos gerados. [25]
Existem pequenas ajudas que podem marcar diferença no decorrer da geração de planos. Estas dicas poderão ajudar o optimizador quer na escolha do plano, quer no quão exaustiva deve ser a geração de todos os planos:

optimizer_prune_level : Permite que o optimizador salte certos tipos de planos baseados em estimativas de linhas acedidas por cada tabela. Por omissão, esta variável é inicializada com o valor "1". O valor pode ser alterado para "0", mas existe o risco do processamento da pergunta ser muito mais demorado.

optimizer_search_depth : Informa ao optimizador o quão aprofundada deve ser a geração de planos incompletos. Ou seja, quanto menor o valor desta variável, menor será o tempo de processamento da pergunta. Se o valor desta variável for "0", o optimizador determina automaticamente qual o valor a utilizar.

Existe ainda, um comando que permite ver qual o plano de execução (índices utilizados, tipos de junções e estimativa de número de linhas) [26] de uma *query*:

```
EXPLAIN [ EXTENDED | PARTITIONS ] query
```

EXTENDED : Obtenção do plano, como mais alguns detalhes

PARTITIONS : Obtenção do plano em tabelas particionadas

4.6 Comparação com o Oracle

Quanto à optimização de perguntas, o Oracle apresenta uma maior versatilidade ao incluir, para além da Materialização, *Pipelining* e Processamento Paralelo.
No que toca aos algoritmos de junção aquando do processamento de perguntas, o MySQL é mais limitado que o Oracle por apenas implementar *Nested-loop join* e uma variante do mesmo.
Quanto aos algoritmos de selecção, podemos concluir que os dois sistemas são bastante semelhantes visto ambos utilizarem o *Full Table Scan* e o *Index Scan*.
Por fim, o MySQL apenas permite especificar os índices a utilizar, enquanto que o Oracle permite tanto a especificação como a escolha de que algoritmo utilizar.

Capítulo 5

Gestão de Transacções e Controlo de Concorrência

5.1 Transacções

Uma transação é uma unidade atómica de operações da base de dados que pode utilizar valores, quer para escrita quer para leitura de uma ou mais base de dados. Isto significa que os efeitos feitos à base de dados ou são todos aceites (*committed*) ou são todos desfeitos (*rolled back*). Como já foi dito a MySQL suporta várias mecanismo de armazenamento (*storage engines*). A InnoDB segue as regras ACID. ACID é para Atomicidade, Consistência, Isolamento e Durabilidade. Transações confiáveis tem de seguir estas quatro propriedades. Operações dentro de uma transação têm de ser atómicas. Isto significa que ou todas as operações têm sucesso ou todas falham. Esta é uma regra de tudo ou nada. A propriedade d consistência garante que a base de dados está num estado consistente após a transação estar terminada. Os dados são válidos e não há registos meio-acabados. O Isolamento é o requisito que diz que as outras operações não podem aceder a dados que foram modificados durante uma transação que não está ainda completa. A questão do isolamento ocorre nos casos de transações concorrentes. Sem isolamento, os dados podem acabar num estado inconsistente. Durabilidade é a habilidade de um sistema de base de dados recuperar as alterações feitas pelas transações sucedidas (*committed*) caso ocorra qualquer tipo de falha do sistema[27]. De forma a utilização transações MySQL, primeiro é necessário partir os *statements* em porções lógicas e determinar quando é que os dados têm de ser *committed* ou *rollback*.

5.1.1 *Nested Transactions*

MySQL não implementa *nested transactions*, mas os seus conectores sim (a partir do conector MySQL Conector/NET 6.3.xx).

5.1.2 Transacções de longa duração

Usa-se transacções de longa duração quando as transacções podem precisar de correr durante um tempo excessivo ou quando não é necessário todas as propri-

edades ACID (isto é, quando não é necessário garantir o isolamento de dados entre transacções). Uma transacção de longa duração pode ter longos períodos de inactividade, maioritariamente devido à espera da chegada de mensagens externas. Transacções de longa duração seguem as regras da consistência e durabilidade, mas não da atomicidade e isolamento. Os dados dentro de uma transacção de longa duração não são locked; outros processos ou aplicações podem modifica-los. A propriedade de isolamento para o estado de actualização não é mantido porque locks durante muito tempo não é algo prático.[28] Infelizmente o MySQL não suporta transacções de longa duração.

5.2 Níveis de Isolamento

Num ambiente altamente concorrente, transações muito isoladas podem originar *deadlocks*. Um *deadlock* é uma situação, quando as transações competem pelos recursos e eficazmente previnem que cada uma acede a esses recursos. Existe também uma troca entre os níveis de isolamento e a *performance* da base de dados. E por isso, um sistema de base de dados pode oferecer vários níveis de isolamentos para as transações. MySQL oferece quatro níveis de isolamento nas transações:

- *Serializable*
- *Repeatable read*
- *Read committed*
- *Read uncommitted*

Indica-se da seguinte forma, no MySQL, qual o nível de isolamento pretendido para as transacções:

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

SESSION é opcional, indica que a transacção só será *serializable* durante aquela sessão, enquanto GLOBAL é para todas as seguintes sessões.
Phantom reads, *non-repeatable reads* e *dirty reads* são problemas. que podem ser encontrados, quando as transacções não estão completamente separadas.
Phantom read ocorre quando uma transacção volta a executar uma *query* que retorna um conjunto de linhas que satisfaz uma condição de procura e encontram que o conjunto de linhas, que satisfaz a condição, mudou devido a uma outra transacção recentemente *committed*.
Non-repeatable read ocorre quando uma transacção volta a ler dados que tenham sido previamente lidos e descobre-se que os dados foram modificados por outra transacção. Que tenham sido *committed* desde a primeira leitura.
Dirty read ocorre quando uma transacção lê dados de uma linha que tenha sido modificada por outra transacção, mas não tenha sido ainda *committed*.

Níveis de Isolamento	*Phantom read*	*Non-repeatable read*	*Dirty read*
Serializable	Não é possível	Não é possível	Não é possível
Repeatable read	Possível	Não é possível	Não é possível
Read committed	Possível	Possível	Não é possível
Read uncommitted	Possível	Possível	Possível

Tabela 5.1: Níveis de isolamento e os problemas que podem ser encontrados.

5.2.1 Tipos de Isolamento

Serializable

Isola completamente os efeitos de uma transacção das outras. Todas as transacções são executadas uma após a outra.

Repeatable read

Declarações não podem ler dados que tenham sido modificados mas ainda não *committed* por outras transacções. Nenhuma outra transacção pode modificar dados que tenham sido lidos pela transacção corrente até que a transacção corrente termine. É o nível de isolamento por omissão na InooDB.

Read committed

Declarações não podem ler dados que tenham sido modificados nas não tenham sido *committed* por outras transacções. As declarações têm de esperar até as linhas de dados que estão em *write-locked* por outras transacções sejam *unlocked* antes destas adquirirem os próprios *locks*. Isto previne que haja leituras de dados errados.

Read uncommitted

Declarações podem ler linhas de dados que tenham sido modificadas por outras transacções sem sequer serem *committed*.

5.2.2 *Savepoints*

No que diz respeito a *savepoints*, a InnoDB suporta as seguintes declarações SQL:

SAVEPOINT define uma certa transacção como *savepoint* com um nome de identificador. Se a transacção corrente tem um *savepoint* com o mesmo nome, o *savepoint* antigo é apagado e um novo é definido. Pode ser definido da seguinte maneira:

```
SAVEPOINT identificador
```

ROLLBACK TO SAVEPOINT faz *rollback* à transacção para o *savepoint* nomeado sem terminar a transacção. Modificações que a transacção corrente fez às linhas depois do *savepoint* foram definidas como *undone* no *rollback*, mas a InnoDB não liberta os *row locks* que estavam guardadas

em memória depois do *savepoint*. (Para as novas linhas inseridas, a informação do *lock* é passada pelo ID da transacção guardado na linha; o *lock* não é guardado separadamente em memória. Neste caso, o *row lock* é libertado no *undo*.) *Savepoints* que tenham sido definidos mais tarde que o *savepoint* nomeado são apagados. Se a declaração ROLLBACK TO SAVEPOINT retornar o erro seguinte, significa que não existe nenhum *savepoint* com o nome dado.

```
ERROR 1305 (42000): SAVEPOINT identifier does not exist
```

Esta declaração (ROLLBACK TO SAVEPOINT) pode ser definida da seguinte maneira:

```
ROLLBACK [WORK] TO [SAVEPOINT] identificador
```

Onde WORK é opcional.

RELEASE SAVEPOINT remove o *savepoint* nomeado dos conjuntos de *savepoints* da transacção corrente. Não ocorre nenhum *commit* ou *rollback*. Mas ocorre um erro caso o *savepoint* não exista. É definido da seguinte maneira:

```
RELEASE SAVEPOINT identificador
```

Todos os *savepoints* da transacção corrente são eliminados se houver COMMIT, ou ROLLBACK que não nomeie um *savepoint*. [29]

5.3 Protocolo de Isolamento

InnoDB usa o protocolo de *two-phase locking*. Em qualquer momento durante a execução de uma transacção, esta pode adquirir um *lock*, mas só depois de as declarações COMMIT ou ROLLBACK serem executadas é que se pode libertar estes *locks*. Irão também ser libertadas todos os *locks*. Os níveis de isolamento da InnoDB são baseados no manusear de *locks* automáticos pelo utilizador. A InnoDB também suporta o *display lock*.

5.3.1 *Deadlocks*

Deadlocks são um problema clássico de base de dados transaccionais, mas não são perigosos a menos que estes sejam tão frequentes que não se possa correr uma certa transacção de forma alguma. Normalmente, é necessário escrever a aplicação de maneira a estar-se preparado para voltar a correr a transacção se esta faz *rollback* por causa de um *deadlock*. InnoDB utiliza automaticamente *row-level locking*. Pode ocorrer *deadlocks* até mesmo no caso da transacção só ter inserido ou eliminado uma linha. Isto acontece porque estas operações não são deveras "atómicas"; é automaticamente colocados os *locks* num (ou mais, se possível) índice de registos da linha inserida ou eliminada.
Os *deadlocks* podem ser lidados e reduzir a percentagem de acontecimento dos mesmos com as seguintes técnicas:

- Em qualquer momento, usar o comando SHOW ENGINE INNODB STATUS para determinar a causa do último. Isto pode ajudar na afinação da aplicação para evitar *deadlocks*.
- Se avisos frequentes de *deadlock* provocarem preocupação, existe um comando que poderá ser usado para ter informação extensiva de debugging, mas o servidor terá de ser reiniciado, o comando é innodb_print_all_deadlocks. Informações sobre cada *deadlock*, não só o último, são guardadas no MySQL *error log*. Remover esta opção e reiniciar o servidor mais uma vez, assim que o debugging esteja feito.
- Estar sempre preparado para voltar a correr uma transacção se falha devido a um *deadlock*. *Deadlocks* não são perigosos. Tenta-se correr a transacção outra vez.
- Fazer *commit* às transacções imediatamente após ter sido realizado um conjunto de operações. Pequenas transacções têm menos probabilidade de que ocorra colisão. Em particular, não deixar uma sessão iteractiva de mysql aberta durante muito tempo com uma transacção *uncommitted*.
- Se se está a utilizar *locking reads* através de:

  ```
  SELECT ... FOR UPDATE
  ```

  ou

  ```
  SELECT ... LOCK IN SHARE MODE
  ```

  Talvez seja melhor reduzir o nível de isolamento, como é o caso de *read committed*.
- Quando se modifica várias tabelas dentro de uma transacção, ou diferentes conjuntos de linhas na mesma tabela, será melhor se essas operações forem realizadas numa ordem consistente, em todas as alturas. Com isto as transacções formam *queues* bem definidas e não *deadlocks*. Por exemplo, organizando as operações da base de dados em funções dentro da aplicação, ou fazer chamadas a *stored routines*, em vez de programar várias sequências semelhantes de declarações do tipo INSERT, UPDATE, e DELETE em sítios diferentes.
- Adicionar às tabelas índices bem escolhidos. Desta maneira as *queries* necessitam de fazer *scan* a menos *index records* e consequentemente fazem menos *locks*. Usar EXPLAIN SELECT para determinar quais os índices que o MySQL *server* acha que são os mais apropriados para as *queries* feitas.
- Utilizar menos *locks*. Se for possível permitir que um SELECT retorne dados de um *snapshot* antigo, não adicionar a clausula FOR UPDATE ou LOCK IN SHARE MODE. O uso do nível de isolamento READ COMMITTED é bom para estes casos, porque para cada leitura consistente dentro da mesma transação, é lido do *snapshot* mais recente. Deve ser também posto o valor de innodb_support_xa para 0, o que irá reduzir o numero de *disk flushes* devido à sincronização no disco de dados e o *binary log*.

- Se nada mais ajudar na resolução dos *deadlocks*, serialize as transacções com *table-level locks*. A maneira correcta para usar LOCK TABLES com tabelas transaccionais, como é o exemplo das tabelas da InnoDB, é começando a transacção com:

  ```
  SET autocommit = 0 // não é START TRANSACTION
  ```

  Seguido de LOCK TABLES, e até as transacções estarem *committed* explicitamente, convém não chamar UNLOCK TABLES.
  Por exemplo, se for necessário escrever na tabela t1 e ler da tabela t2, pode-se fazer o seguinte:

  ```
  SET autocommit=0;
  LOCK TABLES t1 WRITE, t2 READ, ...;
  ... fazer qualquer coisa com as tableas t1 e t2 aqui ...
  COMMIT;
  UNLOCK TABLES;
  ```

  Table-level locks previnem actualizações concorrentes à tabela, evitando assim *deadlocks* à custa de resposta menos imediata devido a um sistema mais ocupado.

- Outra maneira de serializar transacções é criando um tabela *"semaphore"* auxiliar que contem apenas um única linha. Cada transacção terá de actualizar aquela linha antes de aceder a outras tabelas. Dessa maneira, todas as transacções ocorrem de uma maneira serializavel. Note que o algoritmo de detecção automática de *deadlocks* do InnoDB também funciona neste caso, porque um *lock* serializado é um row-level lock. Com MySQL *table-level locks*, o método de *timeout* tem de ser usado para resolver *deadlocks*[30].

5.3.2 Níveis de Granularidade

InnoDB implementa *standard Row-Level Locking* onde existem dois tipos diferentes de *locks*, *shared locks*(S) e *exclusive locks* (X).

***shared locks*(S)** permite à transacção que tem este *lock*, ler uma linha

***exclusive locks*(X)** permite à transacção que tem este *lock*, actualizar, inserir ou eliminar uma linha.

Se a transacção T1 tem um *shared*(S) *lock* na linha r, depois um pedido de uma outra transacção distinta T2 de um *lock* na linha r é lidado da seguinte maneira:

- Um pedido por T2 para um S *lock* pode ser concedido imediatamente. No fim, quer T1 quer T2 têm um S *lock* em r.

- Um pedido por T2 para um X *lock* não pode ser concedido imediatamente.

Se uma transacção T1 têm um exclusive (X) *lock* na linha r, um pedido de uma transacção distinta T2 para um *lock* de qualquer tipo em r, não pode ser concedido imediatamente. Em vez disso, a transacção T2 tem de esperar até a transacção T1 liberte o *lock* que lhe foi atribuído na linha r.

	X	IX	S	IS
X	Conflicto	Conflicto	Conflicto	Conflicto
IX	Conflicto	Compactível	Conflicto	Compactível
S	Conflicto	Conflicto	Compactível	Compactível
IS	Conflicto	Compactível	Compactível	Compactível

Tabela 5.2: Compatibilidade entre *locks* diferentes

Intention Locks

Adicionalmente, a InnoDB suporta *locks* com múltipla granularidade que por sua vez permite coexistência de *locks* de registos e locks em tabelas inteiras. Para fazer com que o *locking* a nível de múltipla granularidade seja pratico, tipos adicionais de *locks* chamados de *intention locks* são usados. *Intention locks* são *table locks* na InnoDB que indicam qual é o tipo de *lock* (*shared* ou *exclusive*) que a transacção irá precisar mais tarde para a linha naquela tabela. Há dois tipos de *intention locks* usados na InnoDB (assuma que transacção T pediu um *lock* do tipo indicado para a tabela t):

***Intention shared* (IS)** : Transacção T tenciona fazer um S *locks* numa linha individual na tabela t. Que pode ser definido da seguinte maneira:

```
SELECT ... LOCK IN SHARE MODE
```

***Intention exclusive* (IX)** : Transacção T tenciona fazer um X locks nas linhas da tabela t. E pode ser definido desta forma:

```
SELECT ... FOR UPDATE
```

O *intention locking protocol* diz o seguinte:

- Antes de uma transacção adquirir um S *lock* numa linha da tabela t, tem de primeiro adquirir um IS *lock* ou um *lock* mais forte em t.
- Antes de uma transacção adquirir um X *lock* numa linha, tem de primeiro adquirir um IX *lock* em t.

Um *lock* é concedido a um pedido de uma transacção se é compactível com os *locks* já concedidos anteriormente. Uma transacção terá de esperar até o lock existente que faz conflicto seja libertado. Se um pedido de *lock* faz conflicto com um *lock* já existente e não pode ser concedido por causa de *deadlock*, ocorre um erro. Assim, *intention locks* não bloqueiam algo excepto *full table requests* (por exemplo, LOCK TABLES ... WRITE). O objectivo principal de IX e IS locks é mostrar que alguém está a fazer *locking* a uma linha, ou vai fazer *lock* a uma linha na tabela[31].

5.4 Comparação com o Oracle

O Oracle 11g e o MySQL são muito parecidos no que diz respeito às transacções, ou seja, ambos seguem as propriedades ACID, ambos utilizam o mesmo

protocolo de isolamento, têm ambos detecção de *deadlocks*, ambos suportam os 4 níveis de isolamento e ambos suportam *Row Level Locking* e *Table Level Locking*.

Capítulo 6

Suporte para Bases de Dados Distribuídas

Primeiro vamos conhecer que tipo Base de Dados Distribuída é esta, se é *homogénea* (composta pelo mesmo tipo de bases de dados) ou *heterogénea* (compostas por diferentes tipos de bases de dados). O tipo de distribuição é homogéneo, visto que no MySQL existem várias ferramentas para permitir a distribuição por vários servidores e como a estrutura da BD é mantida e a diferença essencial está no tipo de permissões (escrita ou leitura) que são atribuídas pela ferramenta em uso, aos servidores. [32]
O objectivo principal de Bases de Dados Distribuídas é sempre o mesmo, cooperação no processamento de pedidos, entre os vários servidores. Sendo que o MySQL é um dos Sistemas de Gestão de Bases de Dados mais populares, é óbvio que inclua mecanismos de suporte para Bases de Dados Distribuídas (*MySQL Cluster*,*Distributed Replicated Block Device*, Fragmentação em *Federated* e MySQL *Replication*). Este capítulo contém uma breve introdução aos dois primeiros enquanto que os outros são aprofundados um pouco mais.

6.1 MySQL *Cluster*

É uma solução síncrona que permite que vários servidores partilhem uma base de dados. Contrariamente ao MySQL *Replication*, qualquer escrita ou leitura pode ser executada em qualquer nó, visto ser um sistema síncrono. O MySQL *Cluster* permite fragmentação, oferecendo elevada disponibilidade ao eliminar o ponto de falha centralizado. O seu protocolo conta com vários *masters* e utiliza *Two-Phase Commit* para garantir transacções que contemplam as propriedades ACID. Esta tecnologia encontra-se implementada para o motor NDB. [33]

6.2 *Distributed Replicated Block Device (DRBD)*

É uma solução suportada apenas em Linux. Este mecanismo guarda a base de dados numa pasta virtual, facilitando assim a replicação do estado da base de dados, copiando apenas essa pasta. O seu funcionamento consiste na criação

de um bloco virtual que pode ser replicado de um servidor primário para um servidor secundário.

6.3 Fragmentação em *Federated*

Ao contrário do que acontece em outras *storage engines*, como no InnoDB, as tabelas *Federated* são armazenas num servidor remoto. Todo o resto é igual. As tabelas *Federated* são constituídas por um servidor remoto com a tabela, que consiste na definição desta e a tabela associada (esta pode estar representada noutra *storage engine*), e um servidor local com a tabela, onde a definição desta, corresponde à da tabela no servidor remoto. Todavia, no ficheiro que guarda a tabela, não estão lá nenhuns dados, está simplesmente uma string com uma ligação para a tabela remota. No que toca ao processamento de consultas e *statements* numa tabela *Federated* (servidor local), as operações que normalmente inseriam, actualizavam ou apagavam tuplos de uma tabela local, são enviadas para o servidor remoto onde são executadas e onde actualizam os tuplos ou devolvem os tuplos correspondentes (no caso dos *statements*) no servidor remoto. Quando um cliente realiza um *statement* que se refere a uma tabela *Federated*, a *storage engine* olha para todas as colunas da tabela e procura refazer o *statement* de forma que a tabela fique a referenciar a tabela remota, usando a *string* (nome) que ficou atribuída na criação da tabela. Depois, o *statement* é enviado para o servidor remoto de MySQL, este processa-o e o servidor local retém qualquer resultado que este produza. Se o resultado contiver referências a atributos, o *Federated* converte os mesmos para a formatação interna do servidor local, levando a que toda esta troca seja transparente para o cliente, que realizou um simples *statement*. [34][35][36]
Para criar uma tabela remota é necessário realizar dois passos:

1. Criar a tabela no servidor remoto, ou anotar a definição de uma tabela já existe usando o SHOW CREATE TABLE:

   ```
   CREATE TABLE test_table (
   id INT(20) NOT NULL AUTO_INCREMENT,
   name VARCHAR(32) NOT NULL DEFAULT '',
   other INT(20) NOT NULL DEFAULT '0',
   PRIMARY KEY (id),
   INDEX name (name),
   INDEX other_key (other)
   )
   ENGINE=InnoDB
   DEFAULT CHARSET=latin1;
   ```

2. Criar uma tabela no servidor local com a mesma definição que a tabela remota, mas adicionando uma informação relativamente à string de conexão (nome de ligação, username e password), usando CONNECTION:

   ```
   CREATE TABLE federated_table (
   id INT(20) NOT NULL AUTO_INCREMENT,
   name VARCHAR(32) NOT NULL DEFAULT '',
   other INT(20) NOT NULL DEFAULT '0',
   ```

```
PRIMARY KEY (id),
INDEX name (name),
INDEX other_key (other)
)
ENGINE=FEDERATED
DEFAULT CHARSET=latin1
CONNECTION ='mysql://fed_user@remote_host:9306/federated/test_table';
```

O comando CONNECTION deve ter a seguinte formatação:

```
esquema://username[:password]@nome_host[:num_porta]/nome_bd/nome_tabela
```

Para além destes comandos referidos, existe ainda outra forma de criar tabelas *Federated*, quando nos deparamos com uma situação em que vamos usar várias tabelas remotas. Esta forma é designada pela criação de um SERVER:

```
CREATE SERVER nome_servidor
FOREIGN DATA WRAPPER nome_wrapper
OPTIONS (opcao [, opcao] ...)
```

6.4 MySQL *Replication*

Replica a informação entre servidores, sem usar grandes configurações. Todavia, esta só pode ser feita do servidor *master* para outros servidores *slave*. A replicação é assíncrona, logo a sincronização não ocorre em tempo real e não há garantia que a replicação do *master* foi feita para os *slaves*. [37][38]
No MySQL *Replication* existem dois métodos para realizar a replicação:

- SBR (*Statement Based Replication*): replica os *statements* em SQL por inteiro;
- RBR (*Row Based Replication*): faz a replicação dos dados de acordo com os tuplos que foram alterados.

Para **configurar o *Master***, é necessária a execução de uma sequência de passos no servidor mestre. O primeiro passo consiste na activação do *Binary Logging* e na definição de um ID único para o servidor, o que pode ser feito através da seguinte sequência de comandos:

```
[mysqld]
log-bin=mysql-bin
server-id = 1
```

No InnoDB com transacções, para obtermos uma configuração da replicação duradoira e consistente, devemos aplicar também as seguintes referências:

```
[mysqld]
innodb_flush_log_at_trx_commit = 1
sync_binlog = 1
```

Estes comandos garantem que o *log* é varrido após cada *commit*. Finalmente basta atribuir um endereço e porta ao servidor:

```
CHANGE MASTER TO
MASTER_HOST= 'master_host_name'
MASTER_PORT= num_port
```

Para **configurar os *Slaves***, o processo é o mesmo, temos de lhe dar um *server-id* único (não esteja atribuído ao *master* ou a um *slave*). Finalmente basta atribuir um endereço e porta ao servidor:

```
[mysqld]
server-id = 2
```

De seguida, é necessário registar no *slave* os valores que ligam ao *master*:

```
CHANGE MASTER TO
MASTER_HOST= 'master_host_name'
MASTER_PORT= num_port
MASTER_LOG_FILE= 'recorded_log_file_name'
MASTER_LOG_POS= recorded_log_position
```

Antes de ser **iniciado o processo dos *Slaves***, é preciso aceder ao *binary log* do *master* e verificar pela ordem dos eventos no *binary log*, o evento inicial a replicar. Depois, para iniciar o processo de replicação do *master*, é necessário parar qualquer processo de *statements* no *master*, fazer um backup dos dados e só depois iniciar a replicação. Isto porque, no fim da replicação, os dados não iriam ficar equivalentes entre os *slaves* e o *master*. Levando as BD à inconsistência. Para tal, é necessário fazer um global *read lock*, usando o comando: FLUSH TABLES WITH READ LOCK que bloqueia também os *commit* (no InnoDB) e, iniciar uma cópia do *master*. Para obter a informação do *binary log* de eventos, usa-se o comando: SHOW MASTER STATUS.
Em seguida, usamos o *mysqldump* para fazer um backup das BD, por exemplo:

```
mysqldump --all-databases --lock-all-tables >dbdump.db
```

Por fim executamos o comando: START SLAVE. [39][40][41][42]

6.5 Transacções Distribuídas (XA)

O motor InnoDB tem suporte para transacções distribuídas, conhecidas como as transacções XA. Uma transacção g*commit* da transacção global é necessário que todas as singulares tenham sido efectuadas com sucesso, ou caso uma falhe, todas falharão. Este facto está de acordo com o standard ACID. [43][44]
As aplicações que utilizam transacções globais necessitam de um ou mais Resource Managers, e um Transaction Manager:

- *Resource Manager* (RM): Disponibiliza acesso a recursos transaccionais; um servidor de base de dados é visto como um *resource manager*. É necessário que seja capaz de efectuar operações de *commit* ou *rollback* sobre transacções iniciadas pelo *resource manager*.
- *Transaction Manager* (TM): Coordena as transacções que constituem a transacção global. Comunica com os *resource managers* de modo a que efectuem determinadas transacção e as operações de *commit* e *rollback*.

De modo a conseguir concretizar os requisitos ACID o MySQL usa o protocolo *Two Phase Commit* (2PC):

1. Na primeira fase todas as transacções são "avisadas" para prepararem a operação de *commit*, ou seja, o *resource manager* deverá escrever as operações em memória não volátil, de seguida devem indicar se obtiveram sucesso neste acto, e caso tal suceda serão estes dados que irão ser usados na segunda fase.

2. Na segunda fase o *transaction manager* indica a todos os *resource managers* se devem efectuar *commit* ou *rollback*. A decisão do *transaction manager* passa pelo estado de todos os *resource managers* na primeira fase.

Em alguns casos uma transacção global pode usar o protocolo *One-Phase Commit* (1PC). Isto acontece quando o *transaction manager* verifica que a transacção global consiste apenas em uma transacção, podendo esta efectuar o *commit* simultaneamente.
Para realizar transações XA no MySQL:

```
CHANGE MASTER TO
XA {START|BEGIN} xid [JOIN|RESUME]
XA END xid [SUSPEND [FOR MIGRATE]]
XA PREPARE xid
XA COMMIT xid [ONE PHASE]
XA ROLLBACK xid
XA RECOVER
```

6.6 Comparação com o Oracle

O Oracle, quanto à replicação, inclui mais opções que o MySQL (limitado ao InnoDB). Este inclui a possibilidade de fragmentação de bases de dados e possibilita a replicação de forma síncrona através do protocolo *Two-Phase Commit*, apresentando mais versatilidade no que toca a suporte para bases de dados distribuídas.

Capítulo 7

Outras Características do MySQL

Existem outras características que não foram mencionadas nos capítulos anteriores que serão abordados em grande maioria neste último capítulo.

7.1 Suporte para WEB

MySQL tem suporte para WEB através de suporte que fornece em XML e em XPath.

7.1.1 Suporte em XML

Exportação de XML

É possível obter um output formatado em XML do MySQL nos clientes mysql ou mysqldump envocando-os com a opção:

```
--xml
```

Por exemplo, caso seja no mysql, então teria de ser algo com a seguintes estrutura:

```
mysql --xml -e "SHOW VARIABLES LIKE 'version%'"
```

No caso de mysqldump teria de ser algo da seguinte forma:

```
mysqldump --xml -u path database_name table_structure_name
```

Importação de XML

MySQL permite fazer imports de ficheiros XML para tabelas na base de dados. Para isso basta correr um código parecido a este de exemplo:

```
LOAD XML LOCAL INFILE '/pathtofile/file.xml'
INTO TABLE table_name(atributo1, atributo2, ...);
```

7.1.2 Suporte em XPath

O MySQL fornece duas XML funções que providenciam capacidades de utilizar com o XPath. Uma é extrair um valor de uma *string* XML usando notação XPath:

```
ExtractValue(xml_frag, xpath_expr)
```

ExtractValue() recebe duas *strings* como argumentos, um fragmento de *XML markup* xml_frag e uma expressão XPath xpath_expr (também conhecida como *locator*); retorna um texto (CDATA) do primeiro nó onde é *child* dos elementos ou elementos que coincidam com a expressão XPath.
A outra função retorna um fragmento XML modificado:

```
UpdateXML(xml_target, xpath_expr, new_xml)
```

UpdateXML() troca uma pequena parte do fragmento de *XML markup* dado xml_target com um novo fragmento XML new_xml, e retorna um XML modificado. A parte do xml_target que é trocada coincide com uma expressão XPath xpath_expr dada pelo utilizador [45].

7.2 *Stored Procedures*

Uma *stored procedure* é um segmento de uma declaração SQL declarativa guardada dentro do catalogo da base de dados. Uma *stored procedure* pode ser invocada por *triggers*, outras *stored procedures* ou aplicações com Java, C#, PHP, e outros tantos. Uma *stored procedure* que se chama a ela própria é conhecida como *stored procedure* recursiva. Maior parte dos sistemas de gestão da base de dados suporta *stored procedures* recursivas [46].

7.2.1 Criar *Procedures*

Observemos primeiro como se pode criar uma *procedure*:

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

type:
    Any valid MySQL data type

characteristic:
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
```

7.2.2 Alterar uma *Procedure*

No caso de ser necessário alterar as características de uma *procedure*, o MySQL também suporta essa possibilidade com o seguinte código[47]:

```
ALTER PROCEDURE proc_name [characteristic ...]

characteristic:
    COMMENT 'string'
  | LANGUAGE SQL
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
```

7.2.3 Eliminar uma *Procedure*

Caso se queira eliminar uma *procedure*, então poderá ser usado o código seguinte [48]:

```
DROP PROCEDURE [IF EXISTS] sp_name
```

7.3 *Triggers*

Um *trigger* é um objecto da base de dados com um nome que está associado a uma tabela, e que despoleta um evento em particular para essa tabela. O *trigger* fica associado com a tabela tbl_name, que terá de referir a uma tabela permanente. Não é possível associar um *trigger* a uma tabela temporária ou a uma *view*. Os nomes dos *triggers* têm de existir no *schema namespace*, ou seja, todos os *triggers* têm de ter um nome único dentro de um *schema*. *Triggers* em *schemas* diferentes podem ter o mesmo nome.

7.3.1 Criação de um *Trigger*

Um *trigger* é criado da seguinte maneira [49]:

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    TRIGGER trigger_name
    trigger_time trigger_event
    ON tbl_name FOR EACH ROW
    trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }
```

7.3.2 Eliminação de um *Trigger*

A seguinte declaração é usada para a eliminação de *triggers*:

```
DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name
```

O nome do *schema* (de base de dados) é opcional. Se um *schema* é omitido, o *trigger* é eliminado do *default schema*. DROP TRIGGER requer que os privilégios de TRIGGER da tabela associada com esse *trigger*. Utiliza-se IF EXISTS para prevenir que ocorram erros devido ao *trigger* não existir. *Triggers* de uma tabela não eliminados se a tabela a que estão associados é eliminada [50].

7.4 Tipos de Dados

O MySQL vários tipos de dados, do tipo numéricos, do tipo *string* e do tipo data ou tempo, para representar valores dentro da base de dados.

7.4.1 Numéricos

Tipos numéricos de dados que permitam atributos UNSIGNED também permitem atributos do tipo SIGNED. No entanto, este tipo de dados são *signed* por omissão, daí que um atributo SIGNED fica sem efeito. Se for especificado ZEROFILL para uma coluna numérica, o MySQL adiciona automaticamente o atributo UNSIGNED à coluna. SERIAL é um alas para:

```
BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE
```

SERIAL DEFAULT VALUE na definição de uma coluna do tipo inteiros é um alas para:

```
NOT NULL AUTO_INCREMENT UNIQUE
```

De seguida serão apresentados os tipos, mas que seja levado em consideração que M indica a largura máxima de *display* para tipos inteiros. O máximo legal de largura de *display* é 255. Largura de *display* não está relacionado com o intervalo de valores que um tipo pode conter. Para tipos de*floating-point* e *fixed-point*, M é o número total de dígitos que podem ser guardados. Existem então os seguintes tipos numéricos[51]:

- BIT
- TINYINT
- BOOL/BOOLEAN
- SMALLINT
- MEDIUMINT
- INT ou INTEGER
- BIGINT
- DECIMAL
- FLOAT
- DOUBLE

7.4.2 *String*

MySQL interpreta nas especificações de tamanho nas colunas *character* definições em unidades do tipo *character*. Isto é aplicado a CHAR, VARCHAR, e às do tipo TEXT. Definições de columas para muitos dos dados do tipo string incluem atributos que especificam o conjunto de caracteres ou colação da coluna. Estes atributos são para aplicáveis para dados do tipo CHAR, VARCHAR, TEXT, ENUM, e SET[52]:

- CHARACTER SET (ou CHARSET)
  - binary

- latin1 (ASCII)
- ucs2 (UNICODE)

Existem então os seguintes tipos de dados:

- CHAR
- VARCHAR
- BINARY
- VARBINARY
- TINYBLOB
- TINYTEXT
- BLOB
- TEXT
- MEDIUMBLOB
- MEDIUMTEXT
- LONGBLOB
- LONGTEXT
- ENUM
- SET

7.4.3 Data e Tempo

MySQL 5.6 permite fracção de segundos para valores de TIME, DATETIME, e TIMESTAMP, até uma precisão de micro-segundos (6 dígitos). Para definir uma coluna que inclui uma parte fraccionária de segundo, usa-se a seguinte sintaxe:

```
type_name(fsp)
```

Onde type_name é TIME, DATETIME, ou TIMESTAMP, e fsp é a precisão fraccionária em segundos. Existem então os seguintes tipos de dados[53]:

- DATE
- DATETIME
- TIMESTAMP
- TIME
- YEAR

7.5 Utilizadores e Segurança

MySQL usa segurança baseada em *Access Control Lists* (ACLs) para todas as conecções, *queries*, e outras operações que os utilizadores possam tentar correr. Também existe suporte para conecções SSL-*encrypted* entre clientes e servidores MySQL. Quando se for a correr MySQL, para que haja segurança é necessário seguir as seguintes regras [54]:

- Não dar acesso às tabelas de utilizador da base de dados mysql a qualquer pessoa.
- Saber como é que o sistema MySQL *access privilege* funciona. Usando declarações de GRANT e REVOKE para controlar o acesso. Estas serão explicadas mais à frente.
- Não guardar *passwords* em forma de texto dentro da base de dados. Em vez disso, usar SHA2(), SHA1(), MD5(), ou outra função de *one-way hashing* e guardar o valor *hash*.
- Não escolher *passwords* tiradas de dicionários. São facilmente descobertas, visto que à programas especiais que se destinam somente a isso, descobrir *passwords*.

- Não transmitir dados *unencrypted* dentro da Internet. A informação é acessível por qualquer pessoa que tenha tempo e habilidade para a interceptar e por sua vez, usa-la para seu beneficio. Em vez disso, usar *encrypted protocol* como é o exemplo do SSL ou SSH.O MySQL suporta conecções SSL internas.
- Saber usar tcpdump e *strings utilities*. Na maioria dos casos, pode ser verificado se as *streams* de dados estão *unencrypted* usando um comando parecido ao seguinte:

  ```
  shell> tcpdump -l -i eth0 -w - src or dst port 3306 | strings
  ```

7.5.1 Criar Utilizador

Tendo em conta que antes de puder dar GRANT a um utilizador é necessário a criação do mesmo, irá ser mostrado primeiro como se cria um utilizador e só depois as declarações GRANT e REVOKE [55].

```
CREATE USER user_specification [, user_specification] ...

user_specification:
    user
    [
      | IDENTIFIED WITH auth_plugin [AS 'auth_string']
        IDENTIFIED BY [PASSWORD] 'password'
    ]
```

Caso seja preciso alterar algo, então o MySQL suporta também o ALTER USER:

```
ALTER USER user_specification [, user_specification] ...

user_specification:
    user PASSWORD EXPIRE
```

No caso de se querer eliminar o utilizador, existe o DROP USER:

```
DROP USER user [, user] ...
```

7.5.2 GRANT

A declaração GRANT dá privilégios a contas de utilizadores MySQL. GRANT também serve para especificar outras características de contas como por exemplo, o uso de conecções seguras e os limites de acesso aos recursos do servidor. Para usar GRANT, é necessário que o utilizador tenha um privilégio de GRANT OPTION, e também têm de ter privilégios para os privilégios que está a dar.

```
GRANT
    priv_type [(column_list)]
      [, priv_type [(column_list)]] ...
    ON [object_type] priv_level
    TO user_specification [, user_specification] ...
    [REQUIRE {NONE | ssl_option [[AND] ssl_option] ...}]
    [WITH with_option ...]
```

```
GRANT PROXY ON user_specification
    TO user_specification [, user_specification] ...
    [WITH GRANT OPTION]

object_type:
    TABLE
  | FUNCTION
  | PROCEDURE

priv_level:
    *
  | *.*
  | db_name.*
  | db_name.tbl_name
  | tbl_name
  | db_name.routine_name

user_specification:
    user
    [
      | IDENTIFIED WITH auth_plugin [AS 'auth_string']
        IDENTIFIED BY [PASSWORD] 'password'
    ]

ssl_option:
    SSL
  | X509
  | CIPHER 'cipher'
  | ISSUER 'issuer'
  | SUBJECT 'subject'

with_option:
    GRANT OPTION
  | MAX_QUERIES_PER_HOUR count
  | MAX_UPDATES_PER_HOUR count
  | MAX_CONNECTIONS_PER_HOUR count
  | MAX_USER_CONNECTIONS count
```

7.5.3 REVOKE

A declaração REVOKE permite aos administradores tirarem certos privilégios de contas MySQL.

```
REVOKE
    priv_type [(column_list)]
      [, priv_type [(column_list)]] ...
    ON [object_type] priv_level
    FROM user [, user] ...
```

```
REVOKE PROXY ON user
    FROM user [, user] ...
```

Para remover todos os privilégios, é utilizado uma sintaxe diferente, que elimina todos os privilégios globais, à base de dados, às tabelas, às colunas, e aos privilégios de rotinas para o utilizador ou utilizadores especificado com o nome deste(s):

```
REVOKE ALL PRIVILEGES, GRANT OPTION
    FROM user [, user] ...
```

7.6 MySQL Workbench

O MySQL Workbench fornece uma ferramenta gráfica para trabalhar com os servidores e base de dados MySQL. O MySQL Workbench tem cinco áreas principais de funcionalidades:

SQL *Development* Permite criar e gerir conecções para os servidores de base de dados. Juntamente com a capacidade de configurar os parâmetros de conecção, o MySQL Workbench providencia a capacidade de executar SQL *queries* nas conecções de base de dados usando um SQL Editor interno.

Modelo de Dados Permite criar modelos de esquemas de base de dados graficamente, *reverse* e *forward engineer* entre o esquema e a base de dados per si, e editar todos os aspectos da base de dados usando um Table Editor de fácil manuseamento. O Table Editor permite editar Tabelas, Colunas, Índices, *Triggers*, *Partitioning*, Opções, *Inserts* and Privilégios, Rotinas e *Views*.

Administração de Servidores Permite administrar instâncias do servidor MySQL por utiliziadores de administração, realizando backup e recuperações, inspeccionando dados *audit*, visualizando o estado da base de dados, e monitorizando a performance do servidor MySQL.

Migração de Dados Permite migrar de servidores do tipo Microsoft SQL, Sybase ASE, SQLite, SQL Anywhere, PostreSQL, e outras tabelas RDBMS, objectos e dados para o MySQL. Também suporta migração de versões anteriores do MySQL para versões mais recentes.

MySQL *Enterprise Support* Fornece suporte para productos Enterprise como é o caso de MySQL *Enterprise Backup* e MySQL *Audit*.

Capítulo 8

Conclusão

O trabalho foi importante pois foi explorado um sistema sobre o qual não tínhamos grande conhecimento, passando a conhecer um pouco mais da estrutura interna deste. Ganhou-se conhecimento do número de técnicas usadas para suportar o sistema de base de dados que conhecemos, e o quanto isso pode ser complexo. Foi um trabalho com o qual estamos satisfeitos e esperemos que nos venha a servir no desenvolvimento dos projectos de software.

Bibliografia

[1] MySQL Manual, "Alternative Storage Engines: http://dev.mysql.com/doc/refman/5.6/en/storage-engines.html," 2014.

[2] MySQL Manual, "The InnoDB Buffer Pool: http://dev.mysql.com/doc/refman/5.6/en/innodb-buffer-pool.html," 2014.

[3] MySQL Manual, "Glossary - System tablespace: http://dev.mysql.com/doc/refman/5.6/en/glossary.html#glos_system_tablespace," 2014.

[4] MySQL Manual, "Limits on Table Size: http://dev.mysql.com/doc/refman/5.6/en/table-size-limit.html," 2014.

[5] MySQL Manual, "Partitioning Types: http://dev.mysql.com/doc/refman/5.6/en/partitioning-types.html," 2014.

[6] MySQL Manual, "Clustered and Secondary Indexes: http://http://dev.mysql.com/doc/refman/5.6/en/innodb-index-types.html," 2014.

[7] MySQL Manual, "Overview of InnoDB Row Storage: http://dev.mysql.com/doc/refman/5.6/en/innodb-row-format-overview.html," 2014.

[8] MySQL Manual, "Physical Row Structure: http://dev.mysql.com/doc/refman/5.6/en/innodb-physical-record.html," 2014.

[9] MySQL Manual, "Fulltext Index: http://dev.mysql.com/doc/refman/5.6/en/fulltext-search.html, urldate = 26-05-2014, year = 2014."

[10] MySQL Manual, "Multiple-Column Index: http://dev.mysql.com/doc/refman/5.6/en/multiple-column-indexes.html," 2014.

[11] MySQL Manual, "Adaptive Hash Index: http://dev.mysql.com/doc/refman/5.6/en/innodb-adaptive-hash.html," 2014.

[12] MySQL Manual, "Start Transaction, Commit, and Rollback Syntax: http://dev.mysql.com/doc/refman/5.6/en/commit.html," 2014.

[13] MySQL Manual, "Optimização do MySQL: http://www.devmedia.com.br/otimizacao-do-mysql-parte-i/147," 2014.

[14] MySQL Manual, "Speed of Select Statements: http://dev.mysql.com/doc/refman/5.6/en/select-speed.html," 2014.

[15] MySQL Manual, "How MySQL Optimizes Where Clauses: http://dev.mysql.com/doc/refman/5.6/en/where-optimizations.html," 2014.

[16] MySQL Manual, "SQL_Small_Result: http://bugs.mysql.com/bug.php?id=46211," 2014.

[17] MySQL Manual, "Distinct Optimization: http://dev.mysql.com/doc/refman/5.6/en/distinct-optimization.html," 2014.

[18] MySQL Manual, "LEFT JOIN and RIGHT JOIN Optimization: http://dev.mysql.com/doc/refman/5.6/en/left-join-optimization.html," 2014.

[19] MySQL Manual, "Outer Join Simplification: http://dev.mysql.com/doc/refman/5.6/en/outer-join-simplification.html," 2014.

[20] MySQL Manual, "Nested Join Optimization: http://dev.mysql.com/doc/refman/5.6/en/nested-join-optimization.html," 2014.

[21] MySQL Manual, "ORDER BY Optimization: http://dev.mysql.com/doc/refman/5.6/en/order-by-optimization.html," 2014.

[22] MySQL Manual, "GROUP BY Optimization: http://dev.mysql.com/doc/refman/5.6/en/group-by-optimization.html," 2014.

[23] MySQL Manual, "Subquery Optimization: http://dev.mysql.com/doc/refman/5.6/en/subquery-optimization.html," 2014.

[24] MySQL Manual, "Controlling Switchable Optimizations: http://dev.mysql.com/doc/refman/5.6/en/switchable-optimizations.html," 2014.

[25] MySQL Manual, "Controlling Query Plan Evaluation: http://dev.mysql.com/doc/refman/5.6/en/controlling-query-plan-evaluation.html," 2014.

[26] MySQL Manual, "EXPLAIN Syntax: http://dev.mysql.com/doc/refman/5.6/en/explain.html," 2014.

[27] J. Bodnar, "Transactions in MySQL: http://zetcode.com/databases/mysqltutorial/transactions/," 2011.

[28] Microsoft, "Long-Running Transactions: http://msdn.microsoft.com/en-us/library/ee253582(v=bts.10).aspx," 2004.

[29] MySQL Manual, "SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT Syntax: https://dev.mysql.com/doc/refman/5.6/en/savepoint.html," 2014.

[30] MySQL Manual, "How to Cope with Deadlocks: https://dev.mysql.com/doc/refman/5.6/en/innodb-deadlocks.html," 2014.

[31] MySQL Manual, "InnoDB Lock Modes: https://dev.mysql.com/doc/refman/5.6/en/innodb-lock-modes.html," 2014.

[32] Wikipedia, "Banco de dados distribuídos: http://pt.wikipedia.org/wiki/Banco_de_dados_distribu%C3%ADdos," 2014.

[33] MySQL Manual, "MySQL Cluster NDB 7.3: http://dev.mysql.com/doc/refman/5.6/en/mysql-cluster.html," 2014.

[34] MySQL Manual, "Creating a Federated Table Using Create Server: http://dev.mysql.com/doc/refman/5.6/en/federated-create-server.html," 2014.

[35] MySQL Manual, "Federated Storage Engine Overview: http://dev.mysql.com/doc/refman/5.6/en/federated-description.html," 2014.

[36] MySQL Manual, "How to Create Federated Tables: http://dev.mysql.com/doc/refman/5.6/en/federated-create.html," 2014.

[37] MySQL Manual, "Replication Implementation: http://dev.mysql.com/doc/refman/5.6/en/replication-implementation.html," 2014.

[38] MySQL Manual, "Semisynchronous Replication: http://dev.mysql.com/doc/refman/5.6/en/replication-semisync.html," 2014.

[39] MySQL Manual, "Troubleshooting Replication: http://dev.mysql.com/doc/refman/5.6/en/replication-problems.html," 2014.

[40] MySQL Manual, "Upgrading a Replication Setup: http://dev.mysql.com/doc/refman/5.6/en/replication-upgrade.html," 2014.

[41] MySQL Manual, "Obtaining the Replication Master Binary Log Coordinates: http://dev.mysql.com/doc/refman/5.6/en/replication-howto-masterstatus.html," 2014.

[42] MySQL Manual, "Creating a Data Snapshot Using mysqldump: http://dev.mysql.com/doc/refman/5.6/en/replication-howto-mysqldump.html," 2014.

[43] MySQL Manual, "MySQL Transactional and Locking Statements: http://dev.mysql.com/doc/refman/5.6/en/sql-syntax-transactions.html," 2014.

[44] MySQL Manual, "XA Transactions: http://dev.mysql.com/doc/refman/5.6/en/xa.html," 2014.

[45] MySQL Manual, "XML Functions: https://dev.mysql.com/doc/refman/5.6/en/xml-functions.html," 2014.

[46] MySQL Tutorial, "Introduction to MySQL Stored Procedures: http://www.mysqltutorial.org/introduction-to-sql-stored-procedures.aspx," 2014.

[47] MySQL Manual, "ALTER PROCEDURE Syntax: https://dev.mysql.com/doc/refman/5.6/en/alter-procedure.html," 2014.

[48] MySQL Manual, "DROP PROCEDURE and DROP FUNCTION Syntax: https://dev.mysql.com/doc/refman/5.6/en/drop-procedure.html," 2014.

[49] MySQL Manual, "CREATE TRIGGER Syntax: https://dev.mysql.com/doc/refman/5.6/en/create-trigger.html," 2014.

[50] MySQL Manual, "DROP TRIGGER Syntax: https://dev.mysql.com/doc/refman/5.6/en/drop-trigger.html," 2014.

[51] MySQL Manual, "Numeric Type Overview: https://dev.mysql.com/doc/refman/5.6/en/numeric-type-overview.html," 2014.

[52] MySQL Manual, "String Type Overview: https://dev.mysql.com/doc/refman/5.6/en/string-type-overview.html," 2014.

[53] MySQL Manual, "Date and Time Type Overview: https://dev.mysql.com/doc/refman/5.6/en/date-and-time-type-overview.html," 2014.

[54] MySQL Manual, "Security: https://dev.mysql.com/doc/refman/5.6/en/security.html," 2014.

[55] MySQL Manual, "Account Management Statements: https://dev.mysql.com/doc/refman/5.6/en/account-management-sql.html," 2014.