FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Análise e Estudo do Sistema MySQL

Sistemas de Bases de Dados (2014)

Departamento de Informática

Trabalho do Grupo 27
David Duarte Nº 41775
Carlos Martins Nº 41748
André Sampaio Nº 41972

Tabela de Conteúdos

Capítulo 1

Introdução

Neste trabalho decidimos abordar o sistema de gerenciamento de bases de dados MySQL. Decidimo-nos sobre esta base de dados por ser uma escolha muito popular com aplicações variadas, sendo usada pela NASA, HP, Nokia, Sony, Google, entre outros. O MySQL é uma SGBD que tem o seu código fonte disponivel sobre os termos da GNU General Pubic License mas também está disponível para uso comercial.

O MySQL teve origem na década de 90 na Suécia por Michael Widenius, David Axmark e Allan Larsson, por uma empresa chamada MySQL AB sendo mais tarde adquirida pela Oracle.

O sucesso do MySQL deve-se em grande parte à fácil integração com o PHP que é uma solução usada frequentemente em serviços de hospedagem de sites, sendo um componente central no conjunto de ferramentas LAMP - Linux, Apache, MySQL, Perl/PHP/Python.

Esta base de dados tem várias interfaces disponíveis usadas para interagir com o sistema em si. O conjunto oficial de ferramentas front-end fornecidas pelo MySQL é o MySQL Workbench que é distribuida abertamente. Esta ferramenta fornece um ambiente gráfico para administrar a base de dados.

Capítulo 2

Armazenamento e Estrutura de Ficheiros

2.1 Mecanismos de Armazenamento

O aspeto que diferencia o MySQL das outras bases de dados relacionais é a capacidade de ter diferentes mecanismos próprios de armazenamento de informação. O utilizador da BD pode escolher o mecanismo consoante as suas necessidades. (MYISAM, INNODB, MEMORY, CSV, ARCHIVE, NDB, BLACKHOLE, MERGE, FEDERATED e EXAMPLE).

É possível escolher o mecanismo de armazenamento na altura em que se cria ou altera-se as tabelas, através da opção ENGINE ou TYPE em versões anteriores. Se esta opção não estiver explícita o mecanismo de armazenamento será o *default* do MySQL:

```
CREATE TABLE account(
username CHAR(25)
,pass CHAR(25) NOT NULL
,userID INT NOT NULL
,age INT NOT NULL
,PRIMARY KEY (username)
,INDEX pwd_idx(pass)
)ENGINE = MyISAM;
```

```
CREATE TABLE account(
username CHAR(25)
,pass CHAR(25) NOT NULL
,userID INT NOT NULL
,age INT NOT NULL
,PRIMARY KEY (username)
,INDEX pwd_idx(pass)
)ENGINE = InnoDB;
```

```
ALTER TABLE User ENGINE = BDB;
```

Também é possível alterar o mecanismo de armazenamento da sessão corrente com o seguinte comando:

```
SET storage_engine = MyISAM;
```

Independentemente do mecanismo de armazenamento, o MySQL cria um ficheiro do tipo *.frm* para guardar as definições das tabelas e colunas. Os dados e índices são guardados em tabelas adicionais dependendo do tipo de mecanismo de armazenamento.

Como o MySQL tem um vasto conjunto de mecanismos de armazenamento, ao longo deste capítulo e, por sua vez, do relatório iremos abordar apenas alguns dos mais utilizados.

2.1.1 Storage Engine MyISAM

O mecanismo de armazenamento MyISAM não gere os dados em páginas (apenas os índices) implicando a um acesso sequencial, através de *Record Caching*.

Aspetos Positivos	Aspetos Negativos
• Simplicidade; • Rapidez; • Procura em Fulltext; • Ficheiros de dados da tabela relativamente menores que ficheiros idênticos geridos pelo InnoDB	• Não assegura a integridade referencial dos dados; • Não garante a verificação das propriedades ACID; • Fraca recuperação de dados em caso de falha; • *Locks* ao nível de relações;

2.1.1.1 File Structure MyISAM

O MyISAM baseia-se em *file systems* do sistema operativo.

O MyISAM utiliza dois ficheiros para guardar os dados e os índices. As extensões são respectivamente MyData (**.MYD**) e MyIndex (**.MYI)**.

- O formato físico do ficheiro MyData depende do tamanho de cada registo, sendo possíveis dois tipos: tamanho variável (VARCHAR,TEXT,BLOB) ou fixo.

- O ficheiro MyIndex contém informação que permite navegar pela árvore utilizada para a indexação dos dados;

- A organização dos dados é sequencial;

2.1.1.2 Registos de Tamanho Fixo no MyISAM

Quando os registos são de tamanho fixo, os ficheiros MyData têm para cada linha, um *bitmap header* composto pelos seguintes *bits* pela ordem que se segue:

- Um bit que indica se o registo foi apagado (0 = Sim | 1 = Não);
- Um bit para cada coluna da tabela, que indica se o valor é nulo. A quantidade de bits depende da quantidade de atributos NULLABLE (1 = NULL);

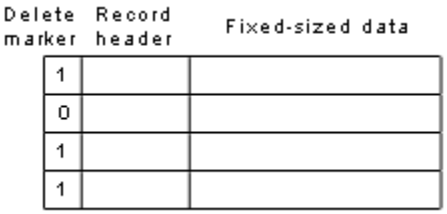

Figura 2.1.1.2 – Dados de tamanho fixo e *header*

2.1.1.3 Registos de Tamanho Variável no MyISAM

Tal como os registos de tamanho fixo, os ficheiros MyData têm para cada registo um bitmap header.

Porém o *header* é composto por mais elementos, nomeadamente:

- O primeiro byte que contém um código descrevendo o subtipo do registo;
- Um ou mais bytes que guardam o tamanho atual do registo;
- Um ou mais bytes indicando o tamanho do espaço não utilizado;
- Um *bitmap* semelhante ao dos registos de tamanho fixo, contendo um bit que indica se a linha foi apagada e um conjunto de campos assinalando se o valor é nulo;
- Um apontador para casos em que o registo é atualizado e o novo tamanho excede o anterior. Este apontador tem o endereço do registo que guarda os restantes dados;

Depois do *header*, os dados são guardados, seguido de um espaço não utilizado;

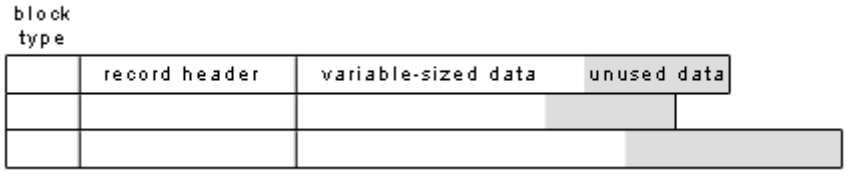

Figura 2.1.1.3 A – Dados de tamanho variável e header

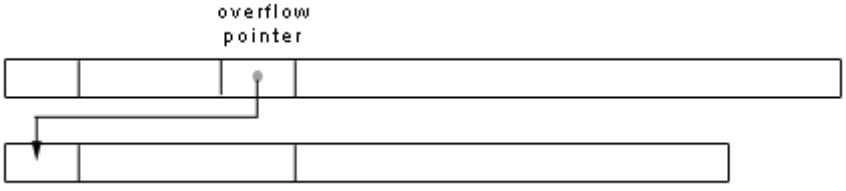

Figura 2.1.1.3 B – Caso de *Overflow*: O primeiro registo não tem espaço inutilizado;

No caso de registos de tamanho variável, um armazenamento excessivo de dados que provoquem *overflow*, pode levar à fragmentação dos dados. Cenário em que se pode correr o comando ***OPTIMIZE TABLE***

2.1.2 Storage Engine InnoDB

O InnoDB é um dos mecanismos mais complexos e importantes do MySQL e cobre algumas das desvantagens do MyISAM.
Nomeadamente permite a utilização de chaves estrangeiras e assegura as propriedades ACID.
Em oposto ao MyISAM, o InnoDB trata dos dados em *pages*.

Aspetos Positivos	Aspetos Negativos
• Permite chaves estrangeiras. • Assegura as propriedades ACID. • Boa recuperação de falhas ao nível de software/hardware. • MVCC *Lock* ao nível das linhas. • Nas versões a partir do 5.6 é possível usar o recurso *Fulltext search.*	• Grande complexidade derivada da garantia da integridade referencial. • Mais lento que o MyISAM.

2.1.2.1 Organização dos ficheiros InnoDB

A organização dos ficheiros utilizando o mecanismo InnoDB é diferente do MyISAM, excepto no ficheiro .frm que é independente do tipo de motor de armazenamento.

O innoDB utiliza o conceito de *Tablespaces*. O *System Tablespace* é um conjunto de ficheiros ibdata que contêm meta-dados de tabelas (data dictionary), *storage areas* para segmentos undo.log, *insert buffer* e *doublewrite buffer*. Fisicamente podem existir vários ficheiros ibdata, mas ao nível lógico são tratados como se fosse um.

Dependendo da varável innodb_file_per_table no ficheiro de configuração (my.cnf),se a mesma estiver descativada o ficheiro ibdata adicionalmente armazena e ordena os dados de todas as tabelas e índices, em 64 *extents* consecutivos. Cada tabela e índice tem uma capacidade de 16KB.

Se a variável innodb_file_per_table estiver ativada, o motor de armazenamento gere um *Tablespace* para cada tabela, isto é, um ficheiro .ibd que guarda os registos de dados e índices dos mesmos. Esses ficheiros vão residir na diretoria da base de dados juntamente com o ficheiro .frm, fora do *System Tablespace*.

Se cada tabela tiver um ficheiro .ibd , é possível apagar o *Tablespace* correndo o seguinte comando:

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

O InnoDB guarda também ficheiros log, limitados a um tamanho máximo de 4GB para operações de *Redo.*

2.1.2.2 Formato dos Registos InnoDB

Os registos podem ser de dois formatos, nas versões mais antigas (pré 5.0.3) os registos são menos compactos e nas mais recentes (5.0.3 e posterior), que guardam praticamente a mesma informação que nas versões anteriores, mas num formato que ocupa menos espaço.

Esses dois formatos, REDUNDANT e COMPACT respetivamente, podem ser especificados pelo utilizador durante a criação das tabelas:

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

Recentemente, o InnoDB permite mais dois formatos, porém é necessário alterar o valor da variável do formato do ficheiro no repositório das configurações (my.cnf). Esses formatos são o DYNAMIC E COMPRESSED.

2.1.2.3 Registos de Tamanho Variável

Nos casos em que existem colunas de tamanho variável, quando um registo não cabe na página de dados, o mecanismo tem que criar uma página externa.

Cada página tem um limite de 16KB e um limite no tamanho dos registos de cerca de 8000 bytes.

No caso em que se tem definido o formato do ficheiro que permite apenas os modos COMPACT ou REDUNDANT (Antologue), o InnoDB cria uma página externa para parte do atributo variável. Por exemplo um registo que não caiba inteiramente na página para os registos e que tenha colunas do tipo BLOB, 768 bytes de cada atributo dinâmico juntamente com o restante registo vão ser guardados na página enquanto os restantes bytes do atributo variável são guardados na página externa. Este método para guardar registos de tamanho variável não é eficaz e geralmente produz erros.

No caso em que se utiliza o novo formato de ficheiro Barracuda, que permite registos do tipo COMPACT, REDUNDANT, DYNAMIC e COMPRESSED, o mecanismo reserva um apontador de 20 bytes para os atributos de tamanho variável, resolvendo os problemas do formato do ficheiro Antologue.

2.2 Caching e Buffer Management

O MySQL tem um subsistema dedicado para o controlo de *caching* e devolução de dados que permite ao sistema diminuir o número de acessos ao disco e de forma a otimizar o tempo de resposta.

Este subsistema tem um conjunto variado de tipos de caches, sendo o que os diferencia é o tipo de informação que é guardado e qual o mecanismo que os utiliza.

2.2.1 Record Cache

Record Caching é usado para processos de leitura e escrita de registos de dados em modo sequencial. Em termos de implementação, é utilizada uma estrutura que guarda um *buffer*, o IO_CACHE.

2.2.2 Key Cache

Key Caching é um repositório para guardar blocos do ficheiro MyIndex. É utilizado no mecanismo MyISAM.

Em termos de implementação, é utilizado uma estrutura que contém um conjunto de blocos de índice. Estes blocos são de tamanho fixo, normalmente tamanho igual a capacidade dos nós da árvore B.

Key Caching utiliza uma técnica do tipo LRU, na medida que é mantido em memória os blocos mais utilizados, os que não são frequentemente utilizados são descartados do *buffer*.

2.2.3 Buffer Pool

O InnoDB utiliza um espaço de armazenamento, o *buffer pool*, para guardar dados e índices em memória. Para eficiência de grandes volumes de operações de leitura, a *buffer pool* é dividida em *pages* que suportam várias linhas de dados.

O *buffer pool* é gerido como se fosse uma lista, segundo uma variante do método LRU.

Em termos de implementação a lista é subdividida em duas: uma sublista com os blocos "young", os mais utilizados e recentes e a segunda com os "old", os menos utilizados e mais antigos. Na inserção de um novo bloco, este é inserido no meio da lista, ou seja, na fronteira entre a lista com os blocos "young" e a com os blocos "old".

Um bloco "old" é transferido para a lista de blocos "young", quando é acedido.

É reservado 3/8 da capacidade do *pool* para a *"old" list*, pelo que se o *buffer* estiver cheio, o bloco mais antigo é descartado do mesmo.

2.3 Mecanismo de Partições no MySQL

O MySQL permite particionar as tabelas no sistema de ficheiros. A função de particionamento das tabelas depende inteiramente do tipo de algoritmo de partição escolhido pelo utilizador. Existem quatro algoritmos de partição: Range, List, Hash e Key.

2.3.1 Range Partitioning

Este algoritmo distribui os registos pelas várias partições com base nos valores dos atributos que pertencem a um dado intervalo:

```
CREATE TABLE account(
username CHAR(25)
,pass CHAR(25) NOT NULL
,userID INT NOT NULL
,age INT NOT NULL
,PRIMARY KEY (username)
,INDEX pwd_idx(pass)
);
```

```
PARTITION BY RANGE (age) (
    PARTITION p0 VALUES LESS THAN (16),
    PARTITION p1 VALUES LESS THAN (18),
    PARTITION p2 VALUES LESS THAN (26),
    PARTITION p3 VALUES LESS THAN (42)
    PARTITION p3 VALUES LESS THAN
MAXVALUE;

);
```

2.3.2 List Partitioning

Semelhante ao Range Partitioning, este algoritmo agrupa os registos pelas várias partições segundo um conjunto de valores dos atributos:

```
CREATE TABLE product(
productname CHAR(25)
,prod_type INT NOT NULL
,price INT NOT NULL
,codP INT NOT NULL
,INDEX prod_idx(prod_type)
);
```

```
PARTITION BY LIST (userID) (
    PARTITION Food VALUES IN (1,2,5),
    PARTITION SelfCare VALUES IN (6,7),
    PARTITION Media VALUES IN (9,10,20,21),
    PARTITION Acessories VALUES IN(30,32,44)
);
```

Durante a inserção de um novo tuplo, pode ser lançado um erro se o valor do atributo que define a partição do registo, não pertencer a nenhuma partição definida. Porém o utilizador só se apercebe desse erro se for utilizado um mecanismo de armazenamento que garante as propriedades ACID, como o InnoDB. Se for utilizado um mecanismo como o MyISAM, nenhuma mensagem de erro é enviada ao utilizador e este não se apercebe do erro.

Type	Stored product types
Food	1,2,5
SelfCare	6,7
Media	9,10,20,21
Acessories	30,32,44

2.3.3 Hash Partitioning

Enquanto na partição por intervalo e por listas o utilizador tem que especificar explicitamente que valor ou conjunto de valores da coluna devem ser inseridos nas partições, neste algoritmo apenas tem que ser especificado a coluna, os valores serão distribuídos com base no resto da divisão do parâmetro pelo número de partições especificadas pelo utilizador.

```
CREATE TABLE…
ENGINE = InnoDB
PARTITION BY HASH(age)
    PARTITIONS 4;
```

```
V = age mod nPart                    nPart = # partições
                                especificadas pelo utilizador
```

Existe também o linear_hashing que utiliza um algoritmo baseado em potências de dois para distribuição dos registos.

```
CREATE TABLE…
ENGINE = InnoDB
PARTITION BY LINEAR HASH(age)
    PARTITIONS 4;
```

$V = 2^{\lceil \log_2 nPart \rceil}$

2.3.4 Key Partitioning

Semelhante ao *Hash Partitioning*, mas para chaves primárias,

```
CREATE TABLE…
ENGINE = InnoDB
PARTITION BY KEY()
PARTITIONS 2;
```

2.3.5 Sub Partitions

Sub-partições de partições:

```
CREATE TABLE …
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
);
```

A tabela tem 3 partições (p0,p1,p2), cada uma é sub-dividida em duas partições.

2.4 Comparação com o Oracle

- Tal como o MySQL, possui o seu próprio *buffer management;*
- Ao contrário do MySQL, não se baseia em *file systems* de Sistemas Operativos;
- Tal como o mecanismo InnoDB do MySQL, a Oracle organiza a base de dados em *Tablespaces*.
- As tabelas são organizadas em heap;
- É possível particionar as tabelas *(Range,Hash e Composite/Sub Partition)*;
- Ao contrário do MyISAM do MySQL, a Oracle permite MultiTable Clustering para *joins* frequentemente utilizados;

Capítulo 3

Indexação e Hashing

O MySQL permite a criação de índices em todo o tipo de colunas. Uma escolha certa do atributo a ser indexado permite uma boa *performance* na execução de *queries*.

3.1 Estruturas de Dados

Os motores de armazenamento do MySQL fazem uso de um conjunto de algoritmos e estruturas para indexação dos dados:

Storage Engine	B Tree	R Tree	Hash	FullText
InnoDB	Todas as versões	A partir da versão 5.7.4 LAB	Adaptive	A partir da versão 5.6
MyISAM	Todas as versões	A partir da versão 4.1	Não	Todas as versões

Figura 3.1 - Formatos dos índices MySQL

3.1.1 R-Tree

O MyISAM e o InnoDB e versões recentes suportam indexamento em árvores R para os dados geográficos/espaciais.

Os dados espaciais são coordenadas geográficas ou dados tridimensionais. As Árvores R são estruturas que se baseiam nas Árvores B, porém implementam uma comparação de valores diferente;

3.1.2 FullText Search

Este tipo de índice e utilizado para aumentar a performance da procura em campos CHAR,VARCHAR, ou TEXT.

Com este índice é possível fazer pesquisas eficientes na pesquisa de campos de texto, utilizando o seguinte comando representado no exemplo:

```
select * from articles WHERE MATCH(title,body) AGAINST ('texto a
procurar');
```

Em que title e body são as colunas em que queremos procurar, anteriormente especificadas no índice aquando a criação da tabela:

```
CREATE TABLE articles (
      ,id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY
      ,title VARCHAR(200)
      ,body TEXT
FULLTEXT (title,body) ) ENGINE=InnoDB;
```

Apenas esta disponível no *Engine MyISAM* em versões anteriores a 5.5 e no *InnoDB* a partir da versão 5.6.

3.1.3 Hash

No MySQL apenas os mecanismos MEMORY e InnoDB suportam o índice por *hashing*.

O MEMORY dá possibilidade de escolha ao utilizador entre o uso de índices por árvores ou por *hash*. Neste mecanismo o hash é dinâmico.

O mecanismo de dispersão no InnoDB é denominado de *Adaptive Hashing*, estes são criados automaticamente quando o engine deteta que pode ser proveitoso.
Esta estrutura de indexação também é utilizada para organização dos ficheiros.

```
CREATE TABLE…
PARTITION BY HASH(age)
PARTITIONS 4;
```

3.1.4 B+-Tree

As árvores B+ são vastamente utilizadas pelo MyISAM e InnoDB. Não existe grande diferença na forma como os mecanismos a implementam, mas sim onde os índices são guardados. Como visto no capítulo anterior, o MyISAM guarda os índices no ficheiro MyIndex (.MYI), enquanto o InnoDB guarda em ficheiros denominados de *Segments.*

O InnoDB utiliza as chaves primárias como índices *clustered*, neste caso as *leaf nodes* da arvore B+ no InnoDB apontam para blocos de dados.

Para o caso dos índices secundários, as *leaf nodes* contêm o próprio índice secundário e as chaves primárias dos registos, utilizando as mesmas para encontrar os registos.

3.2 Ficheiros de índices

O MyISAM mantém ficheiros de índices por tabelas (ficheiro MyDATA). O InnoDB utiliza o conceito de *Tablespaces* para guardar as *index pages*.

3.2.1 Estrutura física dos índices InnoDB

Os índices no InnoDB são do tipo B-Tree, onde os registos são armazenados nas *leaf pages* da árvore.

O tamanho por defeito de um *índex page* é de 16KB (um extent), do qual o *engine* tenta reservar 1/16 aquando de inserções de novos registos.
Se a inserção for ordenada o *fill-factor* será de 15/16 , enquanto que no caso oposto, será de 1/2 à 15/16. No pior caso o mesmo factor será menor que 1/2 e o InnoDB irá "contrair" a árvore de modo a libertar a *page.*

3.2.2 Compression

É uma estratégia de compressão de dados, eliminando bits redundantes ou que ocorrem com mais frequência no conjunto de dados, neste processo não existe perda de informação.

O MyISAM utiliza o myisampack para comprimir uma tabela. Essa funcionalidade baseia-se numa variação do algoritmo de *Huffman* para a compressão de informação. Este algoritmo utiliza estruturas, nomeadamente árvores binárias para analisar com que frequência um determinado dado está presente num bloco e atribuir a esse dado uma sequência de bits. Reduzindo no final a quantidade de bits necessários para representar esse conjunto de informação. O myisampack normalmente comprime a tabela em cerca de 40 a 70% em relação ao tamanho original. Depois da compressão e tabela fica em modo *Read-Only.*

O InnoDB utiliza a biblioteca *zlib* para o efeito. A *zlib* implementa um algoritmo de compressão, o LZ77, sem perdas, robusto e eficiente. O processo é semelhante ao algoritmo de Huffnam, analisa com que frequência a sequência de dados está presente no bloco. O InnoDB utiliza este algoritmo não só para comprimir tabelas, mas também índices. Normalmente os ficheiros são reduzidos em 50%, ou mais em relação ao tamanho original.

A mais-valia deste algoritmo no MySQL reside no facto de quanto mais comprimida for uma tabela, mais dados podem ser armazenados numa única *page*.

3.2.3 Desfragmentação

Esta otimização reorganiza o armazenamento físico da tabela e dos indices associados de modo a reduzir espaço e a melhorar a eficiência I/O.
As modificações exatas variam consoante o *Engine* utilizado.

Para desfragmentar uma tabela executa-se o seguinte comando:

```
OPTIMIZE table_name;
```

3.2.4 Criação e parametrização dos índices

A especificação dos índices pode ser feita criando um índice e mapeando-o para o comando ALTER TABLE na tabela em questão:

```
CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name
    [index_type]
    ON tbl_name (index_col_name,...)
    [index_option]
    [algorithm_option | lock_option] ...

ALTER [IGNORE] TABLE tbl_name
       ADD {INDEX|KEY} [index_name]
```

Uma parametrização de colunas do tipo (col1,col2,…) leva a que a chave do índice seja uma concatenação dos valores dessas colunas.

- **ALTER TABLE tbl_name ADD PRIMARY KEY (column_list):** Este commando adiciona uma chave primária, o que significa que os valores indexados têm que ser únicos e diferentes de NULL;
- **ALTER TABLE tbl_name ADD UNIQUE index_name (column_list):** Cria um índice, tal que os valores têm que ser únicos;
- **ALTER TABLE tbl_name ADD INDEX index_name (column_list):** Cria um índice sem restrições;
- **ALTER TABLE tbl_name ADD FULLTEXT index_name (column_list):** Cria um índice FULLTEXT sobre o atributo;

Alternativamente pode-se especificar a atribuição de um índice à tabela, durante a criação da mesma:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    { LIKE old_tbl_name | (LIKE old_tbl_name) }
create_definition:
    col_name column_definition
(…)
  | {INDEX|KEY} [index_name] [index_type] (index_col_name,...)
      [index_option] ...
(…)
```

Por exemplo:

Para o MyISAM e InnoDB, o único parâmetro possível para o [*index_type*] é o BTREE.

```
CREATE TABLE account(
username CHAR(25)
,pass CHAR(25) NOT NULL
,userID INT NOT NULL
,age INT NOT NULL
,PRIMARY KEY (username)
,INDEX BTREE pwd_idx(pass)
)ENGINE = InnoDB;
```

Para apagar o índice basta especifica-lo na opção *Drop* do comando *Alter Table*:

```
ALTER TABLE tbl_name DROP INDEX index_name;
```

Para consultar informação de todos os índices corre-se o seguinte comando:

```
SHOW INDEX FROM table_name;
```

Para compressão dos dados no InnoDB existe a opção ROW_FORMAT = COMPRESSED, disponível no formato do ficheiro Barracuda. Para comprimir a tabela a opção innodb_file_per_table tem de estar ativa pois a *Tablespace* escolhida por defeito, a do sistema (ibdataN), tem informação vital da base de dados que não pode ser comprimida.

O InnoDB comprime as *pages* de 16KB para 1 KB, 2KB, 4KB ou 8KB, consoante a especificação do KEY_BLOCK_SIZE. Por defeito esse valor é 8.

```
SET GLOBAL innodb_file_per_table=1;
SET GLOBAL innodb_file_format=Barracuda;
CREATE TABLE t1
 (c1 INT PRIMARY KEY)
 ROW_FORMAT=COMPRESSED;
KEY_BLOCK_SIZE=8;
```

Para execução da compressão das tabelas MyISAM especifica-se o seguinte comando:

```
myisampack [options] file_name ...
```

em que file_name deverá ser o nome de um ficheiro MyINDEX da tabela. Depois da compressão, recomenda-se a execução do comando **myisamchk –rq para reconstruir os índices.**

3.3 Comparação com Oracle

Por semelhança ao MySQL, a Oracle possui a Btree para indexação dos dados. Adicionalmente permite o uso de *Bitmap*, tipo de indexação que não está presente no MySQL. Tal como o InnoDB, a Oracle não tem a funcionalidade de *Hashing* para indexação, porém o InnoDB tem uma variante, o *Adaptive Hashing* que preenche essa ausência. A Oracle tal como o MySQL utiliza algoritmos de *Hashing* para organização dos ficheiros.

Capítulo 4

Processamento e Otimização de Perguntas

A otimização de perguntas tem como objetivo encontrar a expressão que minimiza o tempo de execução através da procura de expressões algébricas equivalentes e com um estudo dos seus tempos de execução.

4.1 Operações

4.1.1 *SELECT*

Os *SELECT* em MySQL são realizados como se tratasse de uma junção. O primeiro passo a efetuar na otimização destas operações é a verificação da possibilidade de usar um índice. Esta verificação acontece porque a utilização de índices é muito importante na referenciação de tabelas diferentes, isto porque a sua utilização permite um elevado grau de otimização. Porque efetuar uma operação de junção com auxilio de chaves estrangeiras é mais eficaz do que a junção de atributos que não são chave.

Os passos principais da execução deste operador são:

Preparação: Onde se executam diversas inicializações e procede-se a reorganizações da árvore sintática caso existam clausulas WHERE, ORDER, HAVING ou GROUP BY.

Otimização: A pergunta é reorganizada para ficar otimizada e é determinado o seu plano de execução.

Execução: A pergunta é realizada

Limpeza: Libertação dos recursos que a pergunta utilizou.

Há que ter em conta também, que esta operação é executada através da utilização dos algoritmos de full table scan ou índex scan.

4.1.2 *JOIN*

As junções em MySQL são feitas ou com o *Nested-Loop Join* ou com as suas variações como o *Block Nested-Loop Join*. Estes dois tipos de join diferem na maneira que são realizados assim como nos recursos que são usados.

O Nested-Loop Join passa cada linha da primeira tabela para o loop interno para processamento. Esta passagem da primeira tabela para a segunda tabela

acontece repetidamente até a tabela ser lida. Isto resulta em leituras consecutivas da tabela do loop interior.

Um Nested-Loop Join é processado da seguinte maneira:

```
       for each row in t1 matching range {
  for each row in t2 matching reference key {
    for each row in t3 {
      if row satisfies join conditions,
      send to client
    }
  }
}
```

O Block Nested-Loop Join utiliza um sistema de buffer, guardando as entradas lidas nos loops exteriores para reduzir o número de vezes que as tabelas nos loops interiores tem de ser lidas.

Com um sistema de buffer, o Block Nested-Loop Join é feito da seguinte maneira:

```
for each row in t1 matching range {
  for each row in t2 matching reference key {
    store used columns from t1, t2 in join buffer
    if buffer is full {
      for each row in t3 {
        for each t1, t2 combination in join buffer {
          if row satisfies join conditions,
          send to client
        }
      }
      empty buffer
    }
  }
}

if buffer is not empty {
  for each row in t3 {
    for each t1, t2 combination in join buffer {
      if row satisfies join conditions,
      send to client
    }
  }
}
```

O tamanho de cada buffer de junção é determinado por join_buffer_size. Um determinado buffer é alocado especialmente para cada junção que pode ter

um buffer, logo uma dada pergunta pode ser processada recorrendo-se a junções de múltiplos buffers.

4.1.2.1 Optimização do Left e Right Join

Tanto o Left-Join como o Right-Join são otimizados pela mesma sequência de passos, havendo uma alteração de objectivos para cada uma das duas tabelas consoante o join.

A utilização destes joins mais específicos produz um resultado mais rápido do que a utilização de joins mais gerais, como simplesmente join.

Tendo em conta a junção:

A LEFT JOIN B condition_of_join

A sequência de passos da optimização é:

1. A tabela B depende da tabela A e de todas as suas dependências.
2. A tabela A depende de todas as tabelas que são usadas na condição de junção, à excepção de B.
3. A condição decide como retirar as entradas da tabela B
4. As operações usuais de junção são efectuadas.
5. As clausulas Where são otimizadas.
6. É gerado as entradas com todos os atributos a NULL em B caso existam entradas em B que não tenham uma correspondência a cláusulas WHERE em A

4.1.3 Ordenação – ORDER BY

A ordenação é feita através do comando order by. O MySQL por padrão ordena todas as operações de agrupamento mesmo que tal não tenha sido pedido. Pode-se parar esta ordenação no momento que se efetua o agrupamento recorrendo a GROUP BY NULL.

O MySQL tem dois algoritmos filesort para ordenação e obtenção de resultados:

- Método original que utiliza as colunas indicadas na claúsula *ORDER BY*

- Método modificado que utiliza todos as colunas indicadas na expressão, incorporando otimizações para evitar leituras repetidas.

O método modificado é geralmente mais eficiente porém produz tuplos maiores o que significa que existe possibilidade de o buffer não ter espaço para o processamento intermédio. Este método é o normalmente usado pelo

otimizador à excepção de quando existem colunas de BLOB ou TEXT envolvidas, usando o algoritmo original nestes casos.

Se um sistema usa ordenação frequentemente então existe um comando para aumentar o tamanho deste tal buffer, possibilitando assim mais vezes a execução do algoritmo. O comando é *sort_buffer_size*.

O algoritmo original *filesort* funciona da seguinte maneira:

1. Leitura de todas as entradas de acordo com a chave ou por table scan. Saltar linhas que não têm correspondências com a cláusula WHERE.

2. Para cada entrada, guardar um par de valores no sort buffer (Valor da Sort Key e o ID da entrada)

3. Se todos os pares caberem no buffer, não se cria um ficheiro temporário. Caso contrário, quando o sort buffer ficar cheio, corre-se um quicksort na memória e escreve-se o resultado para um ficheiro temporário. Guarda-se um apontador para o block ordenado.

4. Repetir os passos anteriores até todas as entradas estarem lidas.

5. Fazer um multi-merge até um número de regiões igual a *MEGABUFF* para um outro ficheiro temporário. Repetindo-se até todos os blocos do primeiro ficheiro estarem no segundo.

6. Repetir os próximos passos até haverem menos de *MERGEBUFF2* blocos restantes.

7. No último multi-merge, apenas o ID da entrada é escrito no ficheiro de resultados.

8. Ler as entradas por ordem usando os IDs das entradas no ficheiro de resultados. Para otimizar, faz-se a leitura em blocos que IDs, ordenado-os e usando-os para ler as entradas ordenadamente para um buffer de entradas. O buffer de entradas tem o valor de *read_rnd_buffer_size.*

Este algoritmo tem o problema de ler as entradas duas vezes. Uma vez durante a avaliação da cláusula WHERE e outra depois de ordenar os pares de valores.

Sendo assim usado o algoritmo modificado quando possível para evitar a leitura duplicada.

O algoritmo modificado funciona da seguinte maneira:

1. Ler as entradas que correspondem à cláusula WHERE.

2. Para cada entrada, guardar um tuplo de valores que consistem na chave de ordenação e as colunas referenciadas pela query.

3. Quando o sort buffer ficar cheio, ordena-se os tuplos pela chave de ordenação na memória e escreve-se o resultado para um ficheiro temporário

4. Depois de se fazer merge-sorting ao ficheiro temporário, obtem-se as entradas por ordem mas lê-se as colunas necessárias diretamente dos tuplos ordenados ao contrário de aceder à tabela uma segunda vez.

4.1.4 Agrupamento – GROUP BY

A maneira mais comum de realizar um GROUP BY é percorrer toda a tabela e criar uma nova tabela temporária onde todas as entradas de cada grupo sejam consecutivas. Usando então esta tabela temporária para aplicar funções de agregação e descobrir grupos. O MySQL através de índices consegue realizar esta tarefa muito melhor.

Existem duas maneiras de executar um GROUP BY através do uso de índices. Através do Loose Index Scan e do Tight Index Scan. No primeiro método a operação de agrupamento é aplicada com todos os predicados do intervalo. O segundo método efetua uma pesquisa de intervalo e agrupa os tuplos resultantes.

4.1.4.1 Loose Index Scan

A forma mais eficiente de processar o agrupamento é quando um índice é usado para obter diretamente as colunas de agrupamento. O MySQL faz uso de índices com chaves ordenadas, como B-Tree, para encontrar as colunas necessárias sem ter de percorrer todas as chaves do índice, usando assim apenas uma fração das chaves do índice.

Contudo, para se efetuar este tipo de agrupamento é necessário reunir as seguintes condições:

- A pergunta é feita sobre uma única tabela

- O GROUP BY refere apenas a colunas que foram um prefixo à esquerda do índice. Isto é se uma tabela tem um índice em (c1, c2, c3), este loose índex scan so é possível para uma query de GROUP BY c1, c2. Não sendo possível para GROUP BY c2, c3, pois (c2, c3) não é um prefixo mais à esquerda do índice (c1, c2, c3).

- Se existirem, as únicas funções de agregação são MIN() e MAX() e todas elas têm de referir à mesma coluna. Essa coluna tem de pertencer ao índice.

4.1.4.2 Tight Index Scan

O Tight Index Scan é usado quando não se pode usar o Loose Index Scan porque não se consegue cumprir as suas condições de funcionamento. Esta procura pode ser um full index scan ou range index scan, conforme a pergunta. A operação de agrupamento é feita apenas após de todas as chaves que satisfaçam a condição terem sido encontradas.

4.1.5 DISTINCT

Em termos gerais, um DISTINCT pode ser considerado um caso especial do GROUP BY. Um exemplo desta afirmação é o seguinte:

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

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

Ou seja, devido a esta equivalência de operações, as otimizações efetuadas ao GROUP BY podem ser aplicadas ao DISTINCT.

O MySQL tem o comando row_count disponível para parar de procurar entradas quando encontrar um número de entradas únicas igual a row_count.

4.2 Otimização de Subqueries

O otimizador de queries do MySQL tem estratégias diferentes para lidar com a avaliação de subqueries.

Temos dois conjuntos de escolhas de estratégias a usar conforme o tipo de subquery realizada:

- Para subqueries com *IN* ou *=ANY:*
  - Semi-Join
  - Materialização
  - Estratégia de EXISTS
- Para subqueries com NOT IN ou <>ALL:
  - Materialização
  - Estratégia de EXISTS

4.3.1 Otimização de subqueries com Transformações de Semi-Join

Desde da versão 5.6.5 que o MySQL usa estratégias de semi-join para melhorar a performance da execução de subqueries.

Para uma subquery ser tratada como um semi-join é necessário para o MySQL que essa subquery satisfaça os seguintes critérios:

1. Tem de ser uma IN ou =ANY subquery que aparece nos níveis superiores da cláusula WHERE ou ON, podendo ser um termo e dentro de uma expressão AND.

```
SELECT ...
FROM ot1, ...
WHERE (oe1, ...) IN (SELECT ie1, ... FROM it1, ... WHERE ...);
```

2. Tem de ter um único SELECT sem UNION.

3. Não pode conter uma cláusula GROUP BY e HAVING ou funções agregadas.

4. Não pode conter um ORDER BY com LIMIT.

5. A soma das tabelas exteriores com interiores tem de ser inferior ao número máximo de tabelas permitidas num join.

Se estes critérios se verificarem então a subquery é convertida num semi-join e após uma avaliação de custo executa uma das seguintes estratégias:

1. Table Pullout: Converter uma subquery num join, ou executar a query como se fosse um inner join entre as tabelas da subquery e as tabelas exteriores.

2. Duplicate Weedout: Executar o semi-join como se fosse um join e remover os resultados duplicados com auxilio de uma tabela temporária

3. FirstMatch: Quando se procura as tabelas interiores e há múltiplas instancias de um conjunto de combinações de entradas, obtem-se o primeiro conjunto de valores ao contrário de se obter todos os grupos. Eliminando-se a produção de resultados desnecessários.

4. LooseScan: Pesquisar numa tabela da subquery com auxílio de um índice que permite que um único valor seja escolhido de cada grupo de valores da subquery.

5. Materialização: Materialização da subquery numa tabela temporária com um índice e usar essa tabela temporária para efetuar o join.

Cada uma destas diferentes estratégias à exceção do Duplicate Weedout, pode ser ativada ou desativada pelo uso da variável *optimizer_switch.* A flag *semijoin* determina se é usado semi-joins ou não. Se essa flag tiver definida como *on* então podemos usar as flags *firstmatch*, *loosescan* e *materialization* para permitir um controlo mais refinado sobre as estratégias a usar.

Para expressões complexas, como:

```
SELECT * FROM t1
WHERE t1.a IN (SELECT t2.b FROM t2 WHERE where_condition);
```

O otimizador pode usar materialização para otimizar as expressões. As tabelas intermédias são guardadas em memória sempre que poderem ser lá armazenadas pelo seu tamanho.

A materialização é usada porque caso contrário o otimizador iria reescrever uma subquery não relacionada como se tivesse relacionada. Com a materialização são criadas tabelas temporárias para evitar reescrever subqueries, sendo desta forma possível executar a subquery uma única vez ao contrário de uma vez por cada entrada da query exterior.

A primeira vez que o MySQL precisa de um determinado resultado de uma subquery, o MySQL materializa esse resultado para uma tabela temporária e cada vez que precisar desse resultado o MySQL irá novamente a essa tabela temporária. Esta tabela está indexada com um índice por hash e desta forma as procuras são rápidas e eficazes.

4.3 Mecanismos de Otimização

Para ajudar a otimizar perguntas temos vários mecanismos disponíveis como por exemplo o Benchmarking, o Analyse Table e o Explain Table.

4.3.1 Benchmarking

Para avaliar a velocidade de expressões ou funções especificas podemos invocar o BENCHMARK(). A sintaxe é:

$$BENCHMARK(loopCount, expression)$$

O resultado é sempre 0 mas em conjunto mostra uma linha com a informação de quanto tempo demorou a executar esta expressão.

4.3.2 Analyse Table

O ANALYSE TABLE analiza e guarda a distribuição de chaves para uma tabela. Durante a análise a tabela está com um read lock para tanto o InnoDB ou MyISAM.

```
ANALYZE [NO_WRITE_TO_BINLOG | LOCAL] TABLE
    tbl_name [, tbl_name] ...
```

O MySQL usa a distribuição de chaves para decidir em que ordem as tabelas são unidas quando se efetua uma união. Além disso, também serve para decidir que tipo de índice se pode usar para uma tabela específica numa pergunta.

Pode-se ver a distribuição de chaves que foi guardada através do SHOW INDEX:

```
SHOW {INDEX | INDEXES | KEYS}
    {FROM | IN} tbl_name
    [{FROM | IN} db_name]
    [WHERE expr]
```

4.3.2 Explain Table

Permite obter informação sobre como o MySQL executa uma expressão. Com o MySQL pode-se ver estas informações para o SELECT, DELETE, INSERT, REPLACE e UPDATE.

Com a ajuda do EXPLAIN pode-se ver onde é que se poderia adicionar índices para que as expressões corram mais depressa.

```
{EXPLAIN | DESCRIBE | DESC}
    tbl_name [col_name | wild]
```

4.4 Estimativas

As estimativas feitas são normalmente baseadas no número de disk seeks efetuados. Em tabelas muito grandes, recorre-se ao uso de índices B-Tree como auxílio nas estimativas, podendo-se achar o número de seeks necessários para encontrar uma dada entrada recorrendo-se a esta fórmula:

$$\frac{\log(rowCount)}{\log\left(\frac{indexBlockLength}{3} \times \frac{2}{indexLength + dataPointerLength}\right)} + 1$$

A estimativa pode não ser calcula de modo linear, isto porque o MySQL faz caching de tabelas. Isto é, se uma tabela for pequena e por sua vez possível de se manter em cache então a performance não aumenta de forma logarítmica com o número de disk seeks. Acontecendo apenas se a tabela não caber na cache. Para evitar isto pode-se aumentar a cache para as keys à medida que os dados a base de dados cresce. Para tabelas MyISAM o tamanho da cache é controlado pela variável key_buffer_size.

4.5 Comparações com o Oracle

Para otimização de perguntas o MySQL perde a nível de versatilidade por não incluir Pipelining e processamento paralelo como o Oracle. Para perguntas complexas o MySQL irá revelar-se mais lesnto a processá-las por materialização do que o Oracle por processamento em paralelo.

O Oracle tem também um Index Fast Full Scan que não existe no MySQL o que pode afetar a performance nos algoritmos de seleção.

Capítulo 5

Gestão de Transações e Controlo de Concorrência

O MySQL suporta a gestão de transações desde a versão 3.23 com a introdução do mecanismo de armazenamento InnoDB.

5.1 Transações

As transações em MySQL são definidas através da forma:

```
START TRANSACTION;
SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
UPDATE table2 SET summary=@A WHERE type=1;
COMMIT;
```

Por defeito o MySQL usa o modo *"autocommit"* ,ou seja, a cada operação SQL executada efetua *commit*, fazendo assim update na tabela. Guardando em disco para manter a modificação permanente, **impossibilitando assim um *"roll back"*** (este modo é desactivado aquando o uso do código acima referido).

A transação só termina com o executar do comando *commit*, explícito pelo utilizador, nessa altura verifica-se a consistência. Logo para modificar o momento de verificação basta colocar o modo *autocommit=0* e invocar o commit quando desejado.

Em MySQL não são possíveis *"nested transactions"*.

5.1.1 Propriedades ACID

O mecanismo de armazenamento InnoDB verifica as propriedades ACID, se for utilizado *engines* diferentes em tabelas distintas, apenas aquelas em que se usa o InnoDB, serão garantidas as propriedades.

5.1.1.1 Isolamento e Locks

O *Engine* InnoDB utiliza protocolo de *locks* a nível de linhas e suporta READ-UNCOMMITTED, READ-COMMITTED, REPEATABLE-READ (escolhido por defeito), ou SERIALIZABLE, em que o último permite um nível de isolamento de maior grau. O READ-UNCOMMITTED ou READ-COMMITTED são os mais relaxados a nível de consistência mas com um *overhead* menor.

O REPEATABLE-READ e READ-COMMITTED utilizam Consistent Nonlocking Reads, ou seja utilizam um protocolo multi-versão para gerar uma *"snapshot"* da base dados naquela altura, tendo assim acesso às modificações executadas até ao momento, não tendo em conta as transações *uncommited* evitando assim a necessidade fazer *locking nos reads*. O refrescamento da *snapshot* é obtido aquando o commit da transação.

Pode-se também mudar o foco do nível de isolamento podendo ser:

- **GLOBAL** - Define o *scope* para a transação atual e para todas as próximas
- **SESSION -** Define o *scope* para a transação atual

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

O MySQL utiliza *locking* de tabelas para **MyISAM**, **MEMORY** e **MERGE,***locking* de paginas para **BDB** e *locking* de registros para **InnoDB**.

A definição granularidade é estipulada pela seleção do Engine.

5.1.1.2 Tratamento de Deadlocks

O Engines MyISAM, MEMORY e MERGE evitam os *deadlocks* pedindo sempre todos os *locks* no inicio da pergunta e bloqueando as tabelas por essa mesma ordem adicionando esses pedidos numa fila de espera. Este método reduz a concorrência.

O InnoDB permite múltipla granularidade, sendo que para obter um *lock* no registo tem reservar a tabela assinalando assim uma intenção de *lock* nos tuplos.

	X	*IX*	*S*	*IS*
X	Conflito	Conflito	Conflito	Conflito
IX	Conflito	Compatível	Conflito	Compatível
S	Conflito	Conflito	Compatível	Compatível
IS	Conflito	Compatível	Compatível	Compatível

Para definir um lock de intenção de leitura:

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

Para definir um lock de intenção de escrita:

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

O InnoDB ao utilizar *locking* a nível de registo permite uma maior concorrência.

Definir a granularidade para tabelas:

```
LOCK TABLES
    tbl_name [[AS] alias] lock_type
    [, tbl_name [[AS] alias] lock_type] ...

lock_type:
    READ [LOCAL]
  | [LOW_PRIORITY] WRITE

UNLOCK TABLES
```

O InnoDB deteta o *deadlock* automaticamente, fazendo *rollback* de uma ou mais transações. É escolhida a transação mais pequena para fazer *rollback,* sendo que o tamanho da transação é determinado pela quantidade de inserções, atualizações e remoções de tuplos.

5.1.1.3 Atomicidade, Consistência e Durabilidade

O Engine InnoDB inclui aspectos que promovem atomicidade como:

- Função de Autocommit (commit a cada ação executada);
- Funça de ROLLBACK (faz rollback à transação corrente);

A nível de consistência e durabilidade temos caracteristicas como:

- **doublewrite buffer** - Antes de escrever páginas para os ficheiros de dados, o InnoDB primeiro escreve-os para uma área contigua denominada doublewrite buffer. Apenas após a escrita neste buffer é que escreve para os ficheiros de dados. Caso haja um erro, a recuperação é feita através do acesso a esta estrutura.
- **crash recovery** - Modificações que foram commited antes do problema e não foram escritos para os ficheiros de dados são reconstruidos apartir do doublewrite buffer.

É possivel a utilização de ***Savepoints*** com InnoDB da seguinte forma:

```
SAVEPOINT identifier
ROLLBACK [WORK] TO [SAVEPOINT] identifier
RELEASE SAVEPOINT identifier
```

5.2 Comparação com o Oracle

5.2.1 Gestão das Transações

Neste sistema de base de dados o as transações acabam com um *Commit* explicito ou implicito, implicito no caso de a execução das operações tiverem sido bem sucedidas;o *Rollback* leva à interrupção da transação, o mesmo ocorre no MySQL.

O MySQL suporta também a operação de *Savepoint*. Esta operação demonstra a mesma facilidade no Oracle com no MySQL a nível de sintaxe.

5.2.3 Consistência e Locks

Em ambos os sistemas é utilizado o isolamento por *snapshot e a* consistência é verificada no fim da execução de cada operação, visto ser feito commit após comando.

Estes SBD também partilham o mesmo nível de isolamento por defeito, *Read Commited*, a diferença reside na utilização do mecanismo de snapshot por parte do MySQL neste nível, ao contrário do Oracle.

Os modos de lock no MySQL diferem do Oracle na medida em que o primeiro utiliza uma lógica de "intenção de lock", ou seja, antes de se efecutar o pedido de lock do registo, há ainda um pedido de lock à tabela demonstrando assim intenção de efectuar lock ao contéudo da mesma.
Enquanto que no oracle apenas temos os locks (shared e exclusive) a nível de registo.

Capítulo 6

Suporte Para Bases de Dados Distribuídas

6.1 MySQL Cluster

O MySQL Cluster é uma tecnologia que permite o *clustering* de dados fragmentados in-memory sobre um sistema shared-nothing.

Utiliza um processo denominado de *sharding* para o particionamento dos dados.

Esta tecnologia é construída sobre um mecanismo de armazenamento, o NDB, que distribui os dados por vários servidores e permite o acesso aos mesmos como se tratasse de um sistema centralizado. O NDBCluster é um mecanismo transacional que armazena dados em memória.
Esta tecnologia utiliza um conjunto de *hosts* em que cada um executa um conjunto de processos (Nodes).

6.1.1 Node

O NDBCluster utiliza o conceito de Node, que pode ser estendido em três tipos:

- **SQL Node:** providencia o acesso aos dados, através do parsing de queries até ao retorno dos dados pelo mecanismo de armazenamento do *Cluster*.

- **Storage Node/Data Node:** Processo que armazena os dados. Contém um *Transaction Coordinator* que permite a comunicação com o SQL Node. Cada Data Node deve estar situado em computadores distintos, ou seja, cada *host* possui um *node*.

- **Management Node:** Responsável por especificar informações acerca do Cluster. Quando o SQL e o *storage node* arrancam, comunicam com o *management node* para receber informação. É também responsável por colecionar informações temporárias dos nós e escrevê-los num sistema central para ficheiros log. Normalmente este node está relacionado apenas com um servidor.

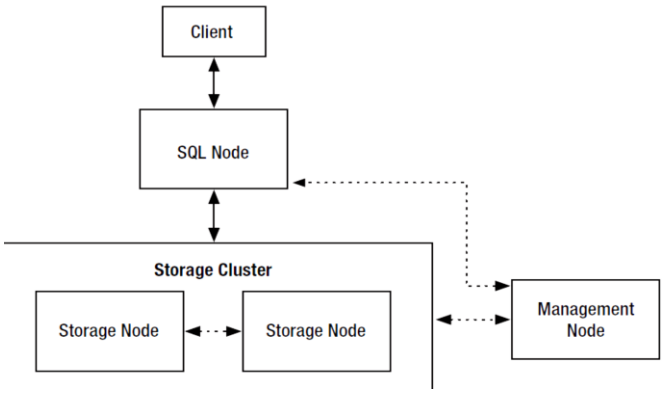

Figura 6.1.1 - Exemplo de arranjo simples dos *nodes*

6.2 Replicação e fragmentação dos dados

O MySQL Cluster cria automaticamente *node groups* que consistem no agrupamento de um ou mais *data nodes*. Cada *node group* tem assim um conjunto de partições e réplicas dos mesmos. As partições são porções de dados guardados em cada *data node*, tipicamente existem tantas partições quanto a quantidade de *data nodes*. Cada *node* tem uma réplica de uma partição de outro *host*.
O número de *node groups* não é diretamente configurável, sendo obtida através da seguinte fórmula:

```
[number_of_node_groups] = number_of_data_nodes / NoOfReplicas
```

Sendo que o número de *data nodes* tem de ser um múltiplo do número de réplicas.

A variável NoOfReplicas é configurável e pode ser acedida no ficheiro de configurações. Por instância, se for especificado que o número de réplicas é 4 e existirem 4 *data nodes*, então haverá 1 *node group*. Se existir mais que um grupo então cada um terá a mesma quantidade de *data nodes.*

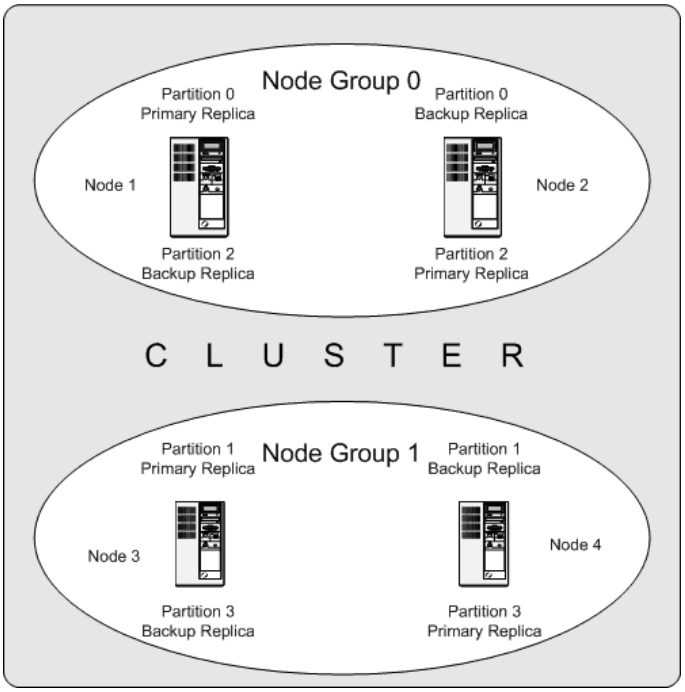

Figura 6.2 - Organização dos Data Nodes e partições no Cluster

No caso acima, pode-se reparar que cada *data node* contém uma partição/réplica primária e adicionalmente uma réplica de *backup* de uma partição de outro *data node* distinto porém do mesmo grupo.
No mesmo exemplo pode-se concluir que enquanto cada *node group* tenha pelo menos um *data node* operacional, todos os dados estão disponíveis para serem acedidos.

6.2.1 Memória

Um dos fatores importantes a ter em conta, é a quantidade de memória que é necessária por cada *data node* para processos de armazenamento pelo NDB. Tipicamente cada *data node* precisa de guardar os dados duas vezes (partição + réplica).
É possível calcular a quantidade de memória, em MB, necessária através da seguinte fórmula:

$$\frac{MB\ total\ de\ dados * número\ de\ réplicas * 1.1}{número\ de\ nodes}$$

6.2.2 Configuração do Management Node

Neste processo reside todas as configurações do MySQL Cluster, sem a especificação das definições deste ficheiro, os restantes *nodes* falhavam. O ficheiro de configuração é do tipo .ini e deverá conter as seguintes entradas:

```
[NDBD DEFAULT]
NoOfReplicas=2
```

```
DataMemory=80M
IndexMemory=52M

[TCP DEFAULT]

[NDB_MGMD]
hostname=<ip of your management node>
datadir=/var/lib/mysql

[NDBD]
hostname=<ip of first storage node>
datadir=/var/lib/mysql

[NDBD]
hostname=<ip of second storage node>
datadir=/var/lib/mysql

[MYSQLD]
hostname=<ip of SQL node>

[MYSQLD]
hostname=<ip of SQL node>
```

Neste exemplo pode-se reparar que existe 5 secções. As duas primeiras dizem respeito à configurações gerais, nomeadamente a primeira tem definições sobre a configuração dos *node groups* e de utilização de memória. As restantes estão relacionadas com processos específicos.
Depois das configurações do management node corre-se o seguinte comando para a execução do processo:

```
Shell> <Path>/ndb_mgmd -f <Path>/config.ini
```

6.2.3 Configuração do Data Node

Depois da configuração e execução do *management node* efetua-se o mesmo procedimento para o *storage node.* O ficheiro de configuração é o my.cnf e deverá conter para além de outras definições as seguintes entradas:

```
[MYSQL_CLUSTER]
ndb-connectstring=<ip of management node>
```

Neste caso especifica-se o endereço IP do computador que corre o *management node*.
Para executar o *data node* executa-se o seguinte comando:

```
shell> <Path>/ndbd -initial
```

Se o *management node* estiver indisponível é lançado uma mensagem de erro.

6.2.4 Configuração do SQL Node

Com o *management* e o *data node* disponíveis e em execução configura-se o SQL *node* para permitir a interação com o mecanismo de armazenamento. O ficheiro de configuração é o my.cnf e deverá conter para além de outras definições as seguintes entradas:

```
[MYSQLD]
ndbcluster
ndb-connectstring=<ip of management node>[:<port>]
```

A opção ndbcluster indica a este servidor que o mecanismo de armazenamento é o NDB. A especificação da porta não é obrigatória.
Para executar o SQL *node* executa-se o seguinte comando:

```
shell> <Path> mysqld_safe &
```

Se o *management node* estiver indisponível é lançado uma mensagem de erro.

Para visualização do estado dos vários processos corre-se o seguinte comando no *host* do management node:

```
shell> <Path> ndb_mgm
ndb_mgm> show
```

6.3 Transações Globais

No MySQL as transações globais utilizam o mecanismo de *2-Phase-Commit*, em que numa primeira fase os ramos envolvidos são preparados e avisados para estarem prontos para executar o *commit*. Na fase seguinte se todos os ramos indicaram que conseguem fazer *commit* então todos fazem *commit*, no entanto, caso um ou mais destes não estejam preparados, todos fazem *rollback*.

Por vezes estas transações também utilizam o *1-Phase-Commit* quando apenas existe um ramo envolvido e não vários.

Os comandos associados às transações globais são os seguintes:

```
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 [CONVERT XID]
```

em que **xid** é o identificador da transação.

Os sistema de base dados utiliza a **replicação síncrona** para garantir que os dados são escritos em todos os nós. Os locks globais são geridos através de um esquema de servidor **Principal** e **Secundário** (Cópia primária) em que as modificações feitas no primeiro são registadas num *Log* por ordem de serialização e seguidamente replicados para os Secundários.

Este sistema distribuído **apenas** suporta *Read Commited* como nível de isolamento e apenas executa *Rollbacks* completos em caso de erro. Não havendo suporte de niveis fracos de consistência.

6.4 BACKUP

É possível fazer backup do sistema utilizando snapshots (aquando o começo ou o fim do backup). Esta operação é executada através da seguinte sintaxe:

```
START BACKUP [backup_id] [wait_option] [snapshot_option]
wait_option:
WAIT {STARTED | COMPLETED} | NOWAIT
snapshot_option:
SNAPSHOTSTART | SNAPSHOTEND
```

6.5 MySQL Replication

É um protocolo de replicação homogénea de bases de dados sobre uma modelo de comunicação *Master-Slave*. Permite replicação de dados entre mecanismos de armazenamento distintos. É utilizado um *log* para verificação periódica de alterações feitas pelo *master*, as atualizações necessárias são guardas em *logs* dos *slaves* e são então executadas nos mesmos.

6.5.1 Configurações

É necessário um conjunto de configurações, nomeadamente:

- Ativação do *log* do servidor mestre (Binary Log) e atribuição de um ID único:

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

- Atribuição de um ID único no slave:

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

- Criação de uma conta para o *slave*, com o objetivo de obter as atualizações periodicamente do *Binary Log* do servidor mestre*:*

```
mysql> GRANT REPLICATION SLAVE ON *.* TO 'replicate'@'%' IDENTIFIED
BY 'r3p1!c8';
```

- É efetuado um *snapshot* de leitura, dos dados no servidor mestre, para o servidor escravo de modo a que este tenha o estado da base de dados do mestre na altura em que se corre a replicação;
- Registro dos dados do mestre no servido escravo:

```
mysql> CHANGE MASTER TO
MASTER_HOST='master.example.com',
MASTER_USER='replicate',
MASTER_PASSWORD='r3p1!c8',
MASTER_LOG_FILE='master-bin.000002',
MASTER_LOG_POS=6016;
```

- O servidor *slave* está então pronto para correr o processo de replicação:

```
mysql> START SLAVE;
```

6.6 MySQL Cluster replication

A replicação “normal” (*non-clustered*) envolve um servidor “mestre” e um servidor “escravo”. O servidor mestre é a fonte dos dados a serem replicados e o escravo o recipiente dos mesmos. No MySQL *cluster* a replicação entre *clusters* é semelhante (assíncrona), porém pode ser mais complexa, visto que os computadores estão organizados em *clusters*, o processo de configuração é diferente.

O servidor mestre e escravo, são referidos como se fossem *replication nodes* com um canal de replicação próprio para o fluxo dos dados.

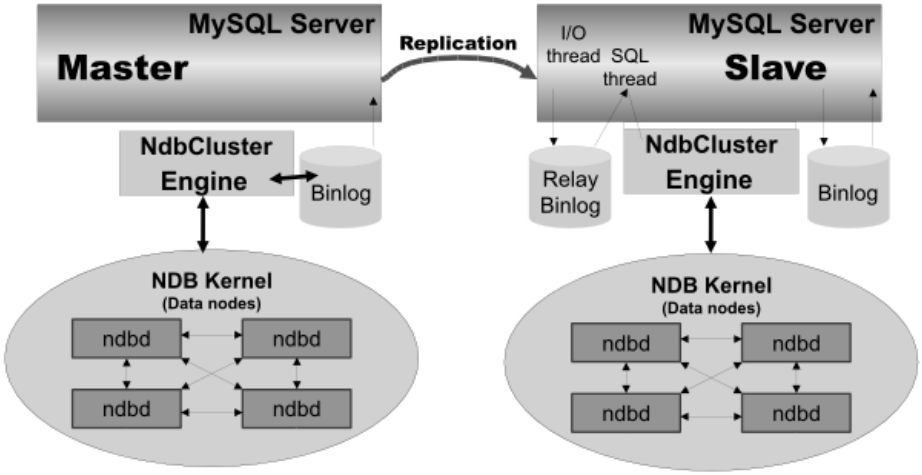

Figura 6.6 – Processo de replicação entre *clusters*

6.7 Comparação com o Oracle

- Ambos os SGBD'S (MySQL com a tecnologia Cluster), permitem a fragmentação dos dados;
- O Oracle permite a replicação das bases de dados de forma síncrona enquanto no MySQL Cluster com o seu protocolo de replicação, o processo é efetuado de forma assíncrona;
- Adicionalmente, visto que o Oracle permite a replicação síncrona, ao contrário do MySQL Cluster os estados entre os servidores estão sempre consistentes.
- Porém, os servidores no MySQL são mais independentes devido à comunicação assíncrona e o uso de repositórios *log*;
- O Oracle também faz uso de *snapshots* para processos de replicação;

Capítulo 7

Outras Características do Sistema

7.1 XML

Com o MySQL podemos tanto importar, exportar e manipular ficheiros XML.

7.1.1 Exportar

Podemos exportar tabelas para um ficheiro .xml recorrendo à seguinte sintaxe:

```
mysql --xml -e 'SELECT * FROM mytable' > file.xml
```

Por exemplo, para uma base de dados com uma tabela chamda “colors” podemos efectuar

```
mysql --xml -e 'SELECT * FROM myDb.colors' > /colors.xml
```

O que resultaria no seguinte ficheiro .xml:

```
<resultset statement="SELECT * FROM mydb.Colors”>
      <row>
            <field name="Id">1</field>
            <field name="Color">Blue</field
      </row>
      <row>
            <field name="Id">2</field>
            <field name="Color">Red</field
      </row>
</resultset>
```

7.1.2 Importar

Para importar dados de um determinado ficheiro .xml para uma tabela podemos usar a expresão *LOAD XML*:

```
LOAD XML [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE 'file_name'
   [REPLACE | IGNORE]
   INTO TABLE [db_name.]tbl_name
   [PARTITION (partition_name,...)]
   [CHARACTER SET charset_name]
   [ROWS IDENTIFIED BY '<tagname>']
   [IGNORE number {LINES | ROWS}]
```

```
   [(column_or_user_var,...)]
   [SET col_name = expr,...]
```

Para carregar o ficheiro .xml do exemplo anterior bastava executar o seguinte:

```
LOAD XML /colors.xml INTO TABLE Colors;
```

7.1.3 Manipulação - XPath

O MySQL dispõe de duas funções que fornecem capacidades básicas de XPath de manipulação de ficheiros XML.

7.1.3.1 Extract Value

A função Extract Value é usada para obter nós de um ficheiro XML:

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

Indica-se o fragmento XML e a expressão XPath que deve ser validada.

7.1.3.2 Update Value

A função Update Value atualiza um ficheiro XML:

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

O primeiro argumento especifica o fragmento XML, o segundo a expressao XPath que atua como um filtro dos nós a serem alterados, e por fim o terceiro argumento indica o código XML a ser inserido no ficheiro XML.

7.2 MySQL Connector/ODBC

O MySQL Connector/ODBC é um nome para uma familia de drivers que fornecem acesso a uma base de dados MySQL de acordo com a API standard Open Database Connectivity, fornecendo flexibiidade em disponibilizar uma interface nativa à base de dados independentemente do sistema operativo utilizado (Windows, Linux, Mac OS, Unix).

7.3 Stored Procedures

Uma stored procedure é uma forma de encapsular tarefas repetitivas, permitindo por exemplo declarações de variáveis entre outras técnicas de programação.

Recorrendo a estas técnicas podemos partilhar lógica com outras aplicações, assegurando que o acesso aos dados e a sua manipulação é coerente entre aplicações.

Pode-se também isolar os utilizadores das tabelas de dados, dando assim acesso às stored procedures que manipulam dados mas não acedem diretamente às tabelas. Sendo assim também um mecanismo de segurança, pois só se pode aceder a dados usando stored procedures definidas.

Pode-se criar uma stored procedure em MySQL da seguinte forma:

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

Onde routine_body representa comandos de SQL válidos a serem guardados.

7.4 Triggers

Um trigger é um tipo especial de uma stored procedure uma vez que não é chamado diretamente como uma stored procedure, mas sim automaticamente quando existe algum evento.

A sintaxe para criar um novo trigger é a seguinte:

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

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

Cada trigger tem de ter um nome único num determinado esquema de base de dados.

7.5 Segurança

O MySQL usa um tipo de segurança baseada em Listas de Controlo de Acessos para todas as conexões, perguntas e outras operações que os utilizadores possam efetuar. Também existe suporte para conexões encriptadas com SSL entre clientes e servidores MySQL.

As funções principais deste sistema é autentificar um utilizador que se conecta a um dado host e associar esse utilizador a um conjunto de privilegios na base de dados como capacidade de fazer SELECT, INSERT, UPDATE ou DELETE.

Define-se e cria-se privilégios recorrendo a CREATE USER, GRANT e REVOKE.

Por exemplo, para criar um utilizador usa-se esta sintaxe:

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

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

7.6 Dados Suportados

O MySQL suporta três grandes categorias de dados como os tipos númericos, data e tempo, e string.

7.6.1 Tipos Númericos

O MySQL suporta todos os tipos standard SQL númericos como:

- **Exatos:** INTEGER, SMALLINT, DECIMAL, NUMERIC.
- **Aproximados:** FLOAT, REAL, DOUBLE.
- **Valores bit:** BIT

7.6.2 Tipos Data e Tempo

Suporta os tipos DATE, TIME, DATETIME, TIMESTAMP e YEAR.

7.6.3 Tipos String

Os tipos de string suportados são o CHAR, VARCHAR, BINARY, VARBINARY, BLOB, TEXT, ENUM e SET.