Comparação de Sistemas de Bases de Dados

Sistemas de Bases de Dados

Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa

MySQL

1 de Junho de 2014

G08
Alexandre Martins Garcia Nº 41952
Miguel José Avó Aires Teixeira Nº 41724
Ricardo Jorge Marchão Carrapiço Nº 419288

Conteúdo

Introdução

Este relatório foi desenvolvido no âmbito da disciplina de Sistemas de Bases de Dados e tem como objetivo analisar um sistema de gestão de base dados relacional. Este relatório em particular incide sobre o MySQL. O MySQL é escrito em C/C++, utiliza a linguagem SQL (Structured Query Language), é open source (sob a licença GNU General Public License) e é atualmente um dos sistemas de bases de dados mais utilizados no mundo [1].

Foi desenvolvido por David Axmark, Allan Larsson e Michael "Monty"Widenius, inicialmente para uso pessoal e para substituir o sistema mSQL, baseado na linguagem de baixo nível ISAM, que os criadores acharam muito lento e inflexível. Os criadores desenvolveram uma nova interface SQL enquanto mantiveram a mesma API que o mSQL, o que permitiu uma fácil transição de um sistema para outro. A primeira versão foi lançada a 23 de Maio de 1995. A 16 de Janeiro de 2008 a empresa MySQL AB, detentora dos direitos do sistema, foi adquirida pela Sun Microsystems. No dia 20 de Abril de 2009 a Oracle adquiriu a Sun e todos os seus produtos pelo que atualmente o MySQL faz parte do portfólio de produtos Oracle.

É um sistema de bases de dados que teve um crescimento muito rápido devido à sua integração com diversas linguagens de programação, como PHP, Java, C, C++, C#, Python, entre outras. Os seus pontos fortes são a portabilidade, compatibilidade e o facto de ser open source. Utiliza um sistema de storage engines que possibilita a escolha do sistema mais efetivo para cada tabela. Existem vários engines disponíveis, sendo os mais usados: o InnoDB, o engine por defeito do MySQL, este suporta transações, locks ao nível dos tuplos e chaves estrangeiras; o MyISAM que prima pela simplicidade e rapidez; o Memory que mantém a tabela apenas em memória e nunca usa o disco. Este sistema de bases de dados é atualmente usado em produtos de alto nível como a Wikipedia, Facebook ou o YouTube [9].

No capítulo 2 é referido ao armazenamento em disco e a estrutura dos ficheiros. Como o sistema organiza os tuplos das tabelas, gere os blocos em memória e o sistema de partições.

O capítulo 3 incide sobre os ficheiros de indexação para acelerar as queries e mostra as possibilidades de indexação disponibilizadas pelo MySQL.

O capítulo 4 explica os mecanismos de processamento de otimização de perguntas deste sistema de bases de dados.

No capítulo 5 é apresentado o sistema de transações e concorrência das mesmas.

A distribuição da base de dados em vários servidores é abordado no capítulo 6 e no capítulo 7 são apresentadas algumas caraterísticas adicionais deste sistema que não se enquadram nas outras secções. No final de cada capítulo é feita uma comparação do MySQL com o sistema de bases de dados estudado nas aulas, o Oracle.

Capítulo 2

Armazenamento e file structure

2.1 Filesystem

Até à versão MySQL 5.6.7, todas as tabelas e indexes no InnoDB eram guardados num conjunto reduzido de ficheiros, chamado de **system tablespace**.

Estes ficheiros, além de conterem informações de metadata relacionadas com as tabelas armazenadas, continham também o undo log, o doublewrite buffer, etc. Isto faria com que os ficheiros se tornassem demasiados grandes, se a base de dados fosse de grandes dimensões. Esta solução provocava problemas de armazenamento dos dados se um grande volume de dados temporários fosse criado e, logo de seguida, eliminado.

Nas versões seguintes, o default passou a ser o modo **file-per-table**, em que cada tabela e respetivos índices são armazenados num único ficheiro em separado de outras tabelas. Este modo facilita a utilização das funcionalidades fornecidas pelo formato de ficheiro Barracuda, como a compressão de tabelas.

A partir da versão 5.6, é ainda permitido, através do comando **innodb undo tablespaces**, dividir o undo log em vários ficheiros tablespace.

O MySQL baseia-se no file system do sistema operativo para aceder a todos os ficheiros.

2.2 Buffer management

O InnoDB mantém em memória uma área onde faz cache de dados e indexes na chamada *buffer pool*. Esta é utilizada para que o acesso aos dados mais frequentes seja feito de forma mais rápida e eficiente. A buffer pool mantém uma lista de pages, usando uma variante do algoritmo **LRU** (least recently used). Quando é necessário adicionar um novo bloco à pool, é removido o bloco menos utilizado e o novo bloco é adicionado no meio da lista.
A lista de blocos em memória pode ser tratada como duas sub-listas:

- No topo da lista são mantidos os blocos acedidos mais recentemente;
- O final da lista, mantém os blocos que foram acedidos há mais tempo.

Assim, os dados que são pouco utilizados são removidos da cache. Os que têm mais probabilidade a sair da lista são os que estão há mais tempo nesta.

Os blocos resultantes de uma query de leitura sobre a base de dados são copiados para a pool, onde são inseridos na sub-lista dos blocos mais recentes.

2.3 Organização dos tuplos

Uma *page* representa uma única transferência de dados entre o disco e o buffer pool, esta pode conter várias rows da tabela, dependendo do tamanho de cada uma.

Quando uma tabela é criada, é criado também um *B-tree clustered index* sobre a tabela, sendo que tanto os secondary indexes como os próprios dados estão organizados segundo esta estrutura. Assim, cada page associada a uma tabela está organizada numa B-tree index (discutida no ponto 3) que armazena todas as colunas de cada row. Colunas de tamanho variável, grandes demais para caberem numa B-tree page, são armazenadas noutra zona do disco que contém todas as overflow pages, sendo que os valores destas colunas são repartidos em várias overflow pages. Cada coluna contém uma lista ligada de uma ou mais overflow pages.

O MySQL não permite multitable clustering file organization.

Apesar disso, o **InnoDB** permite guardar tuplos em quatro formatos: **dynamic**, **compressed**, **compact** e **redundant**. Apesar de permitir estes quatro formatos, é necessário ter em conta o formato do ficheiro utilizado na tabela do respetivo tuplo.

O **InnoDB** armazena os ficheiros nos seguintes formatos: **Antelope** e **Barracuda**.

- O Antelope suporta os formatos redundant e compact, mas não os formatos mais recentes dynamic e compressed;
- O Barracuda suporta apenas os formatos dynamic e compressed. Este formato suporta compressão de dados de uma tabela.

Atualmente, o system tablespace utilizado pelo InnoDB é guardado no formato Antelope, sendo que para utilizar o Barracuda é necessário especificar numa configuração, fazendo com que novas tabelas sejam colocadas no seu próprio tablespace (separado do system tablespace).

Quando uma tabela é criada utilizando o formato **dynamic** ou **compressed**, colunas com valores grandes (maiores que o tamanho máximo de uma page) são guardadas por completo numa overflow page, sendo que o clustered índex record contém apenas um apontador para esta. Se um tuplo for muito grande, o InnoDB coloca as colunas que contém os valores mais elevados numa overflow page até o tuplo caber por completo numa page.

O formato **dynamic** guarda o tuplo num index node se este lá couber por completo, mas evita que se preencham nós B-tree com dados de colunas maiores. Caso um valor do tuplo não caiba por completo, guarda todo o tuplo numa overflow page. Similarmente ao formato dynamic, o **compressed** também coloca um tuplo grande numa overflow page, mas tem considerações adicionais, já que comprime os dados de modo a ter tamanhos de pages mais pequenos.

Nos formatos **compact** e **redundant**, o InnoDB guarda os primeiros 768 bytes de colunas de tamanho variável num nó da B-tree index, os restantes são guardados numa overflow page. Assim, é permitido no MySQL utilizar registos de tamanho variável.

Apesar de se poder guardar dados num índice até 768 bytes, sem ser necessário criar uma overflow page, isto pode causar uma perda de eficiência já que, no caso de se guardarem vários dados blob, começa-se a ter outros dados na árvore e não valores de chaves, e caso estes sejam em grande número, o índice começa a tornar-se menos eficiente do que se os valores das colunas estivessem guardados numa overflow page.

Por defeito, o InnoDB utiliza o formato de ficheiros Antelope e o seu formato compact, fazendo com que se consiga reduzir o espaço ocupado pelos tuplos.

2.4 Partições

O particionamento em bases de dados permite que se distribuam tabelas em parcelas pelo filesystem, sendo que cada parcela de uma tabela pode ser guardada numa localização diferente (cada parte é tratada como uma tabela individual). O utilizador pode definir regras para dividir os dados através de uma função de particionamento.

Apesar de existirem dois tipos de particionamento, nomeadamente **particionamento horizontal**, onde diferentes linhas da tabela são divididas por diferentes partições físicas e **particionamento vertical**, na qual diferentes colunas são colocadas em diferentes partições, o MySQL apenas suporta particionamento horizontal.

No MySQL, podem ser definidos os seguintes tipos de particionamento: **Range**, **List**, **Hash**, **Key**, **Range Columns** e **List Columns**.

De seguida é descrito cada um dos tipos de particionamento indicados. Em todos os exemplos pressupõe-se a existência de uma tabela que contenha no seu esquema uma coluna com o nome *store_id*:

- **RANGE** – cada partição contém tuplos nos quais os valores da expressão de particionamento pertencem a um determinado intervalo. Isto é indicado através do operador VALUES LESS THAN, em que todos os valores que sejam inferiores ao valor indicado a seguir a este operador, são colocados na respetiva partição.

  De modo a evitar que haja um erro no caso de ser introduzido um tuplo cujo valor da coluna na qual está a ser feita a partição não esteja presente no conjunto de regras, deve utilizar-se o valor MAXVALUE.

  Este tipo de particionamento pode ser definido através de

```
PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

- **LIST** - é semelhante ao RANGE, mas a partição é definida com base nos valores pertencentes a listas. Assim, um tuplo é colocado numa partição se pertencer a uma

certa lista de valores, indicada através do operador IN tal como se exemplifica de seguida.

```
PARTITION BY LIST(store_id) (
    PARTITION pNorth VALUES IN (3,5,6,9,17),
    PARTITION pEast VALUES IN (1,2,10,11,19,20),
    PARTITION pWest VALUES IN (4,12,13,14,18),
    PARTITION pCentral VALUES IN (7,8,15,16)
);
```

No caso de ser introduzido um tuplo que tenha um valor de store_id não presente nos valores anteriores, o sistema devolve um erro.

- **RANGE COLUMNS** - semelhante ao tipo RANGE, tem como diferenças a definição dos ranges das partições, neste caso são permitidos vários valores para várias colunas. Note-se que neste tipo de particionamento também se pode ter valores de colunas que não sejam inteiros.

  Um tuplo é colocado numa determinada partição se os valores deste forem iguais aos valores da lista presente nas regras definidas. Supondo a existência de uma tabela que tenha no seu esquema colunas com os nomes a, d, c, as seguintes expressões exemplificam a definição deste tipo de particionamento:

```
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** - esta é um variante do particionamento LIST que permite o uso de valores de colunas de outros tipos diferentes de inteiros;

- **HASH** - o particionamento por hash permite uma distribuição uniforme por várias partições. Enquanto no particionamento por range ou list tem que definir-se em que partições serão colocados os tuplos que estão dentro de um certo intervalo ou que tenham um certo valor, no hash apenas se tem que indicar a coluna a ser utilizada para calcular o valor de hash (desde que seja um inteiro) ou uma expressão que também retorne um número inteiro.

  Além disso, tem de se indicar o número de partições a criar, ficando o sistema encarregue da partição dos tuplos. Esta operação é executada calculando o resto da divisão inteira do número de tuplos pelo número de partições.

  Um exemplo de criação de um particionamento por hash é

```
PARTITION BY HASH(store_id) PARTITIONS 4;
```

  O MySQL também suporta um outro tipo de HASH chamado de **LINEAR HASH**. Em vez de se utilizar uma função que aplica o módulo sobre um dado valor, é utilizada uma função que utiliza um powers-of-two algorithm, como descrito em [4].

  A criação do particionamento é muito semelhante à partição por HASH, acrescenta-se apenas a palavra LINEAR:

```
PARTITION BY LINEAR HASH (store_id) PARTITIONS 4
```

A vantagem deste particionamento linear hash é que adicionar, remover, fazer merge e separar partições é muito mais rápido do que no hash, o que pode ser benéfico quando se lida com tabelas com grande quantidade de dados. A desvantagem é que as partições podem estar distribuídas de forma menos uniforme.

- **KEY** - é semelhante ao particionamento por hash, mas em vez de ser o utilizador a especificar a função de hash, esta é da responsabilidade do sistema.

  A criação de um particionamento deste tipo é feito da seguinte forma (supondo a existência de uma tabela k1 com os atributos id e name, sendo id chave primária (**k1(id, name)**):

```
PARTITION BY LINEAR KEY() PARTITIONS 2;
```

  Note-se que não é necessário indicar os nomes das colunas, pois será utilizada a primary key da tabela (caso não exista é utilizada uma unique key).
  Caso o utilizador deseje, pode indicar quais as colunas a utilizar neste particionamento, desde que estas sejam primary key ou tenham a restrição not null.

  Também se pode particionar uma tabela utilizando **LINEAR KEY**, sendo que este terá o mesmo efeito que o linear tem no particionamento do tipo hash, será utilizado o powers-of-two algorithm em vez do resto de divisão.
  Para criar um linear key o utilizador necessita apenas de acrescentar a palavra linear antes do key existente na expressão anterior:

```
PARTITION BY LINEAR KEY() PARTITIONS 2;
```

2.5 Comparação com o Oracle 11g

Ao criar-se uma tabela no Oracle, esta terá os seus tuplos organizados num **heap** (heap-organized table). Também é possível organizar os tuplos de uma tabela num índice (index-organized table).

O buffer management do Oracle também utiliza o algoritmo **LRU**.

No Oracle, os dados estão armazenados em tablespaces. Um tablespace pode ser partilhado por várias tabelas e estes por sua vez estão armazenados em datafiles, sendo que um tablespace pode estar dividido em dois datafiles. Automaticamente é criado o SYSTEM tablespace, que contém a maior parte da informação sobre a estrutura e conteúdos da base de dados.

No Oracle é permitido multitable clustering, ao contrário do MySQL.

Também é permitido no Oracle particionamento de tabelas, sendo que os métodos disponíveis são **Range**, **List**, **Hash** e **Composite**.

Capítulo 3

Indexação e Hashing

3.1 Indexes

Cada tabela tem um ficheiro de clustered index onde é guardada a informação dos tuplos. Normalmente este index está ordenado pela chave primária da tabela.

Assim, é aconselhado que se defina uma chave primária para cada tabela para ser usada no index. Mesmo que logicamente não haja uma, deve-se introduzir uma coluna que faça uso do mecanismo auto-increment para estabelecer ID's para os tuplos.

Caso não seja definida uma chave primária, o MySQL procura o primeiro UNIQUE index em que não haja colunas que suportem valores nulos e usa como clustered index.

Se não houver chave primária ou colunas que possam ser usadas como chave primária, é internamente adicionada uma coluna que contém ID's geridos pelo sistema de base de dados para cada tuplo, sendo depois criado um clustered index sobre esta coluna.

Existe também suporte para ficheiros de index secundários. Estes sob o engine InnoDB utilizam a chave primária junto com as colunas especificadas para o index secundário. A chave primária é usada para o sistema procurar depois o tuplo correspondente no clustered index.

O MySQL permite a criação de índices em múltiplos atributos (multiple-column indexes), sendo que um índice deste tipo suporta até 16 atributos. Estes índices podem ser utilizados pelo MySQL para queries que testam todas as colunas no índice ou mesmo apenas algumas.

Um índice em múltiplos atributos pode ser criado da seguinte forma:

```
CREATE TABLE test (
      id          INT NOT NULL,
      last_name   CHAR(30) NOT NULL,
      first_name  CHAR(30) NOT NULL,
     PRIMARY KEY (id),
      INDEX name (last_name,first_name)
);
```

Neste caso, o índice é criado sobre os atributos *last_name* e *first_name*.

O MySQL suporta quatro tipos de indexes: O tipo **KEY** (ou INDEX) refere-se a um index non-UNIQUE. Valores não distintos são permitidos, significando que no index pode haver tuplos que tenham todas as colunas idênticas.

O tipo **UNIQUE** especifica que todos os tuplos no index têm que ser distintos (suporta colunas com valores nulos). Além de melhorar a performance das pesquisas, também podem ser usados para impôr restrições de integridade, não permitindo que existam tuplos iguais.

O tipo **PRIMARY** é semelhante ao UNIQUE, mas não suporta colunas com valores nulos e apenas pode haver um index deste tipo por tabela. No caso do InnoDB é um clustered index, ou seja, os tuplos estão guardados no disco pela ordem do index. Por fim há um tipo de index diferente dos anteriores chamado **FULLTEXT**. Este tem como objectivo a pesquisa de texto e não serve um propósito geral como os anteriores.

Os indexes **KEY**, **UNIQUE**, **PRIMARY** e **FULLTEXT** são guardados em árvores B+ sem qualquer opção de escolha. O MySQL tem suporte para tipos de dados espaciais (GEOMETRY, POINT, LINESTRING e POLYGON) e neste caso os tipos de index utilizados são árvores R. Tabelas que usem o engine MEMORY, isto é, tabelas que fiquem apenas em memória, suportam por defeito hash indexes (por razões históricas suportam também árvores B+).

As tabelas a funcionar em InnoDB suportam ainda um outro tipo de index chamado **Adaptive Hash Index** (AHI), que é construído a partir de um index secundário já existente (que usa uma árvore B+). Este index tem como objectivo acelerar as pesquisas que usem os operadores = e IN usando um hash index que reside em memória. O index é gerido pelo sistema de base de dados e pode ser parcial apenas para as chaves mais usadas.

Para criar um index sob o engine InnoDB ou MyISAM a sintaxe é a seguinte:

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

index_col_name:
    col_name [(length)] [ASC | DESC]

index_type:
    USING {BTREE | HASH}

index_option:
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'

algorithm_option:
    ALGORITHM [=] {DEFAULT|INPLACE|COPY}

lock_option:
    LOCK [=] {DEFAULT|NONE|SHARED|EXCLUSIVE}
```

Para criar, por exemplo, um index usando árvore B+ sobre um atributo id na tabela lookup:

```
CREATE INDEX id_index ON lookup (id) USING BTREE;
```

Ou no caso de uma tabela que esteja a usar o engine MEMORY, pode-se usar tanto árvores B+ como *hash indexes*. Para o *hash index* seria:

```
CREATE INDEX id_index ON lookup (id) USING HASH;
```

No caso deste engine, é utilizado dynamic hashing nas tabelas.

3.2 Comparação com o Oracle 11g

Tal como no MySQL, o Oracle por defeito cria um *index* em árvore B+ para cada chave primária. Cria também um index em árvore B+ para cada UNIQUE existente, o MySQL não. O Oracle suporta *clustered indexes* mas por defeito a organização dos ficheiros é feita em *heap*.

Suporta múltiplos indexes com múltiplos atributos e também acrescenta uma nova coluna virtual que tem como objectivo tornar todos os tuplos únicos.

Ao contrário do MySQL, o Oracle implementa *bitmap indexes*.

Em relação ao hashing, ambos os sistemas não implementam indexes baseados em *hashing* (com excepção do MEMORY engine do MySQL como foi referido anteriormente). O Oracle suporta ainda organização estática dos ficheiros baseada em hashing para as partições.

Capítulo 4

Processamento e otimização de perguntas

Ao executar um query que seleciona várias rows, o MySQL analisa este query de forma a verificar se podem ser feitas otimizações para que o seu processamento seja o mais rápido e eficiente possível.

O query optimizer do MySQL tira proveito da existência de indexes, mas também pode utilizar outra informação disponível. Começa por utilizar os indexes mais restritivos de modo a eliminar o mais rápido possível o maior número de rows.

Na terminologia do optimizador MySQL, cada query é um conjunto de joins. Assim, podem utilizar-se as mesmas estruturas e algoritmos que são usados nos joins.

O optimizador seleciona o método de acesso aos registos e coloca as tabelas numa ordem que julga ser a ideal para minimizar o custo. Assim, a otimização de um query pode ser dividida em duas partes: para uma dada ordenação de um join, encontrar os melhores caminhos de acesso a cada tabela; depois, basta encontrar a melhor ordem para fazer o join num curto período de tempo.

O melhor caminho de acesso a cada tabela define se o optimizador vai ler de uma chave, fazer um scan à tabela ou fazer um scan à chave. Para encontrar a melhor ordem para fazer o join, pode executar-se uma das seguintes opções: uma procura exaustiva ou uma procura *greedy*.

A procura exaustiva examina todas as combinações possíveis de tabelas e procura o melhor plano. O problema desta procura é que pode demorar muito tempo.

A procura greedy, primeiro tenta todas as combinações possíveis das *optimizer_search_depth* (uma variável com um valor inteiro) tabelas das n tabelas presentes no query e encontra a melhor. Apesar da procura *greedy* não encontrar sempre o melhor plano, tem uma complexidade controlada logo, em termos de performance, esta é superior à procura exaustiva.

Após ter sido determinada a melhor ordem de joins, o optimizador começa a executar o join, sendo que este é feito através de uma sequência de nested loops. A combinação é depois comparada com o conteúdo da cláusula **WHERE** e são retornados os tuplos que a satisfazem.

De seguida são apresentadas algumas otimizações feitas pelo MySQL sobre várias cláusulas.

4.1 Otimização de WHERE

- **Remoção de parênteses desnecessários**
  ((a AND b) AND c OR (((a AND b) AND (c AND d))))
  é transformado em
  (a AND b AND c) OR (a AND b AND c AND d);

- **Troca de atributos por constantes**
  (a<b AND b=c) AND a=5
  é transformado em
  b>5 AND b=c AND a=5;

- **Remoção de condições constantes**
  (B>=5 AND B=5) OR (B=6 AND 5=5) OR (B=7 AND 5=6)
  é transformado em
  B=5 OR B=6;

- Expressões constantes usadas por índices são avaliadas apenas uma vez;

- Deteção inicial de expressões inválidas;

- Cláusulas que contenham **HAVING** são agrupadas com o **WHERE**, se não existir um **GROUP BY** ou funções de agregação;

- Quando é feito um join, para cada tabela presente neste, é primeiro executado um **WHERE** de modo a conseguir-se uma avaliação final mais rápida, já que nesta fase são removidas várias rows;

- Se todas as colunas de uma tabela apenas contiverem valores numéricos, estes são lidos diretamente da *index tree*.

  Se uma query contiver a cláusula **ORDER BY**, os tuplos são obtidos utilizando o índice, não sendo necessário um passo adicional para ordenação.

4.2 Otimização de RANGE

Quando é utilizado um range numa cláusula **WHERE**, é utilizado um índex para obter a parte dos tuplos da tabela que estão contidos no intervalo indicado. Condições com range podem ser especificadas, no caso de índices **BTREE** e **HASH**, através de uma comparação entre uma chave com um valor constante utilizando os operadores =, <=>, IN(), IS_NULL ou IS_NOT_NULL. Para além destes, nos índices BTREE são ainda considerados os operadores >, <, >=, <=, BETWEEN, !=, <> ou LIKE (no caso do LIKE deve ser comparado com uma string que não inicie com uma wildcard).

4.3 Otimização de ORDER BY

Como indicado anteriormente, o MySQL pode usar um índice para responder a uma query que utilize um **ORDER BY** sem necessitar de ordenar a resposta. No entanto, por vezes não é possível o MySQL utilizar índices para resolver o ORDER BY. Neste caso, o MySQL

recorre ao algoritmo *filesort*, na qual é feito um quicksort em partes dos dados que cabem em memória, utilizando no fim uma variante do *mergesort* para juntar estes pedaços [12].

Por defeito, o MySQL ordena todos os queries que contenham *GROUP BY col1, col2* da mesma forma que se tivesse sido especificado *ORDER BY col1, col2*.

No caso de se desejar evitar o overhead de ordenação, é possível evitar esta funcionalidade, bastando para tal fazer ORDER BY NULL após o GROUP BY.

O MySQL tem dois algoritmos *filesort* para ordenar os resultados obtidos: o algoritmo **filesort original**, na qual são utilizadas apenas as colunas presentes no ORDER BY e o algoritmo **modificado** que além de utilizar as colunas do ORDER BY, também usa as restantes colunas referidas no query. O algoritmo *filesort* a utilizar numa determinada query é determinado pelo optimizador. A descrição dos dois algoritmos pode ser consultada em [5].

Um problema com o algoritmo **filesort original** é que cada tuplo é lido duas vezes, sendo que a primeira vez é durante a avaliação da cláusula WHERE e a segunda durante a ordenação dos valores.

O algoritmo **filesort modificado** tem uma otimização para evitar ler cada tuplo duas vezes. Em vez de guardar a *row ID*, guarda as colunas referenciadas pelo query. Utilizando o algoritmo **filesort modificado**, os tuplos são maiores que os tuplos no **filesort original**, levando a que o algoritmo filesort modificado seja mais lento devido a extra I/O, já que estes tuplos não cabem no *sort buffer*.

Para evitar este problema, o optimizador usa o algoritmo filesort modificado apenas se o tamanho total das colunas extra no tuplo ordenado não exceder o valor definido na variável de sistema *max_length_for_sort_data*.

4.4 Otimização de JOINs

No caso de uma operação de junção como A LEFT JOIN B, o MySQL executa-a da seguinte forma:

- Todas as otimizações de join standard são feitas, exceto se uma tabela depender de todas as outras. Neste caso, esta é lida no final. No caso de haver uma dependência circular, é retornado um erro;
- Se houver uma row em A que satisfaça uma cláusula WHERE, mas não houver uma row em B que satisfaça a condição ON, é gerada uma row em B com todas as colunas a NULL.

Quando é feito um JOIN, o optimizador determina a ordem na qual as tabelas devem ser processadas. Ao introduzir-se **LEFT JOIN** ajuda o optimizador a executar a otimização mais rapidamente, já que existem poucas permutações de tabelas a considerar.

Os joins no MySQL são executados utilizando o algoritmo **nested-loop join** ou **block nested-loop join**.

No momento de parsing, os queries que contenham operações right outer join são convertidos em queries equivalentes que contêm apenas operações left join.

A conversão é feita seguindo a seguinte regra:

(T1, ...) RIGHT JOIN (T2,...) ON P(T1,...,T2,...)
é convertido para
(T2, ...) LEFT JOIN (T1,...) ON P(T1,...,T2,...)

Quando o optimizador avalia um plano que contenha operações outer join, tem apenas em consideração os planos onde, para cada operação, as outer tables são acedidas antes das inner tables, sendo que apenas estes planos permitem a execução de queries com operações outer join utilizando o nested loop.

4.5 Otimização de GROUP BY

O MySQL consegue evitar, através da utilização de indexes, que se crie uma tabela temporária onde todas as rows de cada grupo estão consecutivas, para depois se poder descobrir cada grupo.

Para tal, existem dois algoritmos: **Loose Index Scan** e **Tight Index Scan**.

4.5.1 Loose Index Scan

Quando um **GROUP BY** é utilizado, a forma mais eficiente de o processar é quando um index contém todas as colunas presentes neste.

Como, no caso das *Btree index*, as chaves estão ordenadas, consegue-se fazer um *lookup* nos grupos sem se necessitar que todas as chaves do index satisfaçam todas as condições da cláusula *WHERE*.

Caso não exista uma cláusula *WHERE*, são lidos tantos números de chaves como o número de grupos.

Para ser possível aplicar este algoritmo, têm que verificar-se as seguintes condições:

- A query tem de ser feita sobre uma tabela;
- Os nomes presentes no GROUP BY devem estar pela mesma ordem da parte mais à esquerda do índice, ou seja, no caso de se fazer *GROUP BY c1, c2*, o índice deve estar sobre *(c1, c2, c3)*. Caso seja feito *GROUP BY c2, c3*, já não é aplicável;
- As funções de agregação usadas pelo SELECT devem ser MIN e MAX, e a respetiva coluna deve estar presente tanto no índice como no GROUP BY;
- Para as colunas presentes no índice, o valor completo da coluna deve ser indexado e não apenas uma parte. Por exemplo, se for definido *c1 VARCHAR(20), INDEX(c1(10))*, não vai ser possível utilizar este índice neste algoritmo.

4.5.2 Tight Index Scan

Se as condições do **loose index scan** não se verificarem, ainda é possível evitar a criação de tabelas temporárias. Caso existam condições com range na cláusula WHERE, são lidas apenas as chaves que satisfaçam estas condições. Se não existirem estas condições, é feito um *index scan*.

Para se conseguir utilizar este algoritmo, basta que exista uma condição de igualdade com uma constante em todas as colunas que venham antes ou estejam num espaçamento no GROUP BY, como se exemplifica de seguida:

Assumindo a existência de um índice definido em **(c1, c2, c3)**,

```
SELECT c1, c2, c3 FROM t1 WHERE c2 = 'a' GROUP BY c1, c3;
```

Neste caso existe um espaçamento no GROUP BY, já que a coluna *c2* não aparece neste. No entanto, como existe uma condição de igualdade com uma constante na outra coluna, satisfaz a condição para se poder aplicar o algoritmo.

O outro caso será:

```
SELECT c1, c2, c3 FROM t1 WHERE c1 = 'a' GROUP BY c2, c3;
```

Aqui, o GROUP BY apenas é aplicado em *c2, c3*, mas como a coluna *c1* vem antes de *c2, c3*, e existe uma condição de igualdade com uma constante nesta coluna, também se verifica que satisfaz a condição do algoritmo.

4.6 Otimização DISTINCT

Muitas vezes o DISTINCT pode ser considerado como um caso especial do GROUP BY. Sendo assim, as otimizações que se aplicam no GROUP BY também podem ser aplicadas no DISTINCT.

4.7 Otimização de subqueries

Para subqueries **IN** ou **NOT IN**, o otimizador pode utilizar *materialization*.

Utilizando *subquery materialization*, o otimizador consegue um processamento mais eficiente. Com isto, consegue-se que a execução de um query seja mais rápida ao guardar o resultado do subquery numa tabela temporária (normalmente em memória).

Caso a tabela se torne muito grande, esta é colocada em disco.

Para que se consiga utilizar esta otimização, a *flag materialization* deve ser colocada com o valor on.

4.8 Otimização LIMIT

Por vezes, o MySQL otimiza um query que contenha um LIMIT da seguinte forma:

- Se forem selecionados apenas alguns tuplos, são utilizados indexes em vez de *full table scan*;
- Caso seja utilizado um ORDER BY junto com um LIMIT, a ordenação termina assim que é encontrado o número de tuplos existentes no LIMIT.

  Neste caso, pode ser feita uma ordenação com um índice (caso exista) ou com *filesort*, onde todos os tuplos que satisfazem o query sem LIMIT são selecionados, e onde a maior parte é ordenada antes de se encontrar o número de tuplos existentes no LIMIT;
- Utilizando um DISTINCT, o MySQL pára a execução do comando assim que encontra o número de tuplos existentes no LIMIT que sejam unique;
- Se for utilizado LIMIT 0, é automaticamente retornado um conjunto vazio;

Apesar de existirem estas otimizações, muitas vezes é preferível fazer um *full table scan* nas seguintes circunstâncias:

- A tabela é muito pequena (menos de 10 tuplos e cada tuplo não tem muitas colunas);
- Não existem restrições nas cláusulas ON ou WHERE aplicáveis às colunas indexadas.

4.9 Comando EXPLAIN

É possível no MySQL verificar como é que uma expressão está a ser executada.

As expressões permitidas são SELECT, DELETE, INSERT, REPLACE e UPDATE.

A informação mostrada na execução deste comando é obtida do optimizador e mostra como é que a expressão seria executada. Para cada tabela usada numa expressão SELECT, é retornada uma row com informação sobre esta.

A ordem pela qual aparecem as tabelas no output é a ordem de processamento das tabelas.

A sintaxe para executar este comando é *EXPLAIN query*.

4.10 Estimativas

Para cada tabela e index InnoDB, existem estimativas. Os valores principais destas são a cardinalidade (número de valores distintos) e o número total de rows.

O cálculo destas é baseado no número de disk seeks necessários.

A fórmula para este cálculo é:

$$\frac{log(row\_count)}{log(\frac{index\_block\_length}{3} \times \frac{2}{index\_length + data\_pointer\_length})} + 1 \tag{4.1}$$

De modo a evitar que os valores destas estimativas sejam recalculados frequentemente, pode ativar-se as *persistent statistics*, onde os valores são armazenados nas *system tables* InnoDB e são atualizadas apenas quando é executado o comando *ANALYZE TABLE*.

A tabela *STATISTICS* presente no *INFORMATION_SCHEMA* (meta-dados da base de dados), disponibiliza informação sobre os índices das tabelas.

4.11 Performance Schema

O *MySQL Performance Schema* é uma funcionalidade para monitorizar a execução do servidor. Este tem as seguintes caraterísticas:

- Disponibiliza uma forma de verificar a execução interna do servidor em tempo de execução;
- Monitoriza eventos do servidor, tais como chamadas a funções, uma parte de uma expressão SQL (e.g. parsing ou sorting) ou mesmo uma expressão completa ou grupo de expressões.

Este foi implementado com vista a disponibilizar acesso a informação sobre o servidor tendo pouco impacto na sua performance.

4.12 Comparação com o Oracle 11g

O optimizador do Oracle também faz transformações nos queries de modo a obter uma expressão SQL mais eficiente. Além disto, guarda estimativas de um plano de execução.

Os tipos de access path disponíveis no Oracle são:

- Full table scans;
- Rowid scans;
- Index scans;
- Cluster access;
- Hash access.

Quanto aos joins, pode utilizar um dos seguintes métodos:

- Nested Loop Joins;
- Hash Joins;
- Sort Merge Joins.

Permite também a utilização de *pipelining*, sendo mais eficiente já que, ao contrário do MySQL que utiliza *materialization*, permite a execução paralela de várias subárvores do plano de execução em vez de esperar que uma subárvore do plano termine para poder iniciar outra que dependa desta.

O Oracle, tal como o MySQL, disponibiliza um comando para analisar o plano de execução de uma expressão (*EXPLAIN PLAN*).

Capítulo 5

Gestão de transações e controlo de concorrência

Uma transação é uma unidade de um programa que acede e/ou atualiza vários data items numa base de dados. Para executar uma transação basta inserir os seguintes comandos:

- Começar por colocar, antes das expressões SQL, os comandos **START TRANSACTION** ou **BEGIN** para especificar o início da transação. No comando **START TRANSACTION** é ainda possível definir os seguintes modificadores:
  - **WITH CONSISTENT SNAPSHOT** permite que todas as leituras efetuadas na transação sejam consistentes, ou seja, o seu resultado se mantenha o mesmo até ao final da mesma. Isto só é possível se o nível de isolamento atual o permitir. No InnoDB este comando é executado por defeito e só é válido quando usado nos níveis de isolamento *READ COMMITTED* e *REPEATABLE READ*, caso contrário, é ignorado;
  - **READ WRITE** ou **READ ONLY** permite ou proíbe alterações às tabelas acedidas na transação, respetivamente. Apesar disso, a restrição *READ ONLY* não coloca qualquer restrição a tabelas temporárias, criadas no decorrer da transação.
- Depois das expressões SQL, para fazer commit, basta adicionar o comando **COMMIT**. Para cancelar as alterações feitas à base de dados, adiciona-se o comando **ROLLBACK**.

  Por defeito o MySQL tem ativado o modo de *autocommit*. Isto significa que, após qualquer alteração à base de dados, as alterações são transferidas para disco de modo a torná-las permanentes. Para desativar este modo, utiliza-se o comando **START TRANSACTION** em vez do comando **BEGIN** (que ativa este modo por defeito). Pode também desativar-se explicitamente o *autocommit* executando o seguinte comando: *SET autocommit=0;*

5.1 Nested transactions

O MySQL não permite definir explicitamente nested transactions, mas é possível simular este comportamento utilizando o mecanismo de savepoints. Para tal, basta utilizar o comando SAVEPOINT identifier na transação.

Mais tarde, para fazer rollback para um ponto intermédio da transação, basta executar o comando **ROLLBACK TO [SAVEPOINT] identifier**.
Desta forma, operações anteriores ao savepoint são mantidas e a transação não é terminada.

5.2 Long-duration transactions

Para permitir transações de longa duração é importante que o tamanho dos *log files* tenham um tamanho adequado para permitir registar todas as operações decorridas na transação. O tamanho máximo dos *log files* é 512GB e o valor por defeito é 48 MB.

Os *logs* são mantidos temporariamente em memória no *buffer pool* para reduzir as operações I/O com o disco rígido. O tamanho reservado no *buffer*, por defeito, é de 8MB, quanto maior for este valor menos operações de I/O terão de ser executadas até ao *commit*.

5.3 Modelo ACID

O modelo ACID é um conjunto de princípios, que salientam aspetos de fiabilidade, importantes a seguir em aplicações criticas como é o caso da gestão de uma base de dados.

5.3.1 Atomicidade

A atomicidade é um mecanismo que garante que, numa transação, todos os comandos são executados com sucesso ou nenhum é executado, garantindo a consistência dos dados. O InnoDB utiliza *logs* e um mecanismo apelidado de *fuzzy checkpoints* para recuperação de erros que possam ocorrer durante a transação.

Os dados são transferidos do buffer pool para disco em pequenas quantidades evitando a interrupção do processamento de comandos SQL que estejam a ser executados. Durante a recuperação, o InnoDB procura pela última *checkpoint label* nos ficheiros de log. Todos os comandos antes da label foram corretamente processados e transferidos para disco com sucesso. Os comandos depois da *label* são executados novamente de modo a garantir a atomicidade da transação.

Além disso, pode ser forçada a recuperação manual da base de dados através do mecanismo *point-in-time recovery*. Este permite recuperar/restaurar os dados de um determinado ponto no passado. Este tem dois modos de execução: através de **event time** ou **event positions**. Primeiro deverá ativar-se os *binary logs* com o seguinte parâmetro:

```
--log-bin[=base\_name]}
```

Um exemplo de uma recuperação com *point-in-time recovery* utilizando **event times**, que restaura a base de dados para a versão do dia 2005-04-20 às 9:59:59 horas:

```
mysqlbinlog --stop-datetime="2005-04-20 9:59:59" \
/var/log/mysql/bin.123456 | mysql -u root -p
```

Utilizando **event positions**:

```
mysqlbinlog --stop-position=368312 /var/log/mysql/bin.123456 \
         | mysql -u root -p
```

5.3.2 Consistência

A execução de uma transação deve preservar o estado (consistente) da base de dados.

Para tal, o InnoDB verifica as restrições de integridade linha a linha no decorrer da transação nos comandos *insert*, *delete* ou *update*. Ao fazer esta verificação, é feito um *shared row-level lock* sobre os registos. Não é possível, por enquanto, adiar a verificação

da consistência de uma transação para o final da mesma, tornando impossível certas operações.

A verificação da consistência de tabelas na base de dados pode ser verificada executando o seguinte comando:

```
CHECK TABLE tbl\_name [, tbl\_name] ... [option] ...

option = {FOR UPGRADE | QUICK | FAST | MEDIUM | EXTENDED | CHANGED}
```

Se for encontrado algum erro na consistência durante a execução do comando o servidor é desligado de forma a evitar a propagação do erro.

5.3.3 Isolamento

O protocolo de isolamento utilizado no InnoDB combina as melhores propriedades do protocolo de *multi-versão* com o *two-phase locking*. Para garantir escritas atómica faz locks a cada tuplo (*row-level locking*, por defeito) e para garantir leituras consistentes utiliza multi-versão.

O InnoDB tem diferentes tipos de locks aplicado ao row-level locking: **record locks**, **gap locks** e **next-key locks**.

- **Record locks** - lock ao index do tuplo/record.
  No caso da tabela não ter qualquer index, é criado um clustered index oculto e utilizado para fazer lock sobre o tuplo em questão;

- **Gap locks** - este lock é feito sobre o espaço entre index records, ou sobre o espaço anterior/seguinte ao primeiro/último index lock. Com este tipo de locks, enquanto uma session detém um lock sobre um index record, nenhuma outra session pode inserir/modificar um tuplo que esteja no ”espaço”bloqueado;

- **Next-key locks** - este lock combina o record lock (no index record) com o gap lock (no espaço anterior ao index record). Ao procurar pelo tuplo no index, aplica locks (shared ou exclusive) aos percorrer os tuplos. Deste modo, nenhuma inserção pode ocorrer até que o lock sobre o index record seja libertado.
  Este é o modo ativado por defeito no InnoDB.

Para modificar características associadas à transação basta executar o comando:

```
SET [GLOBAL | SESSION] TRANSACTION transaction_characteristic
```

Ao utilizar-se a palavra-chave *GLOBAL*, o comando será aplicado a todas as sessões posteriores à atual. Pelo contrário, ao utilizar-se palavra-chave *SESSION*, o comando aplicar-se-á apenas à sessão atual.

Se não se colocar qualquer uma das alternativas, o comando será válido até ao final da próxima transação na sessão atual.

Esta última hipótese só será permitida se não houver qualquer transação ativa no momento da execução do comando. Na *transaction_characteristic* é possível definir, além da característica *READ_ONLY* ou *READ_WRITE*, referidas anteriormente, o nível de isolamento da transação.

O InnoDB permite definir quatro níveis de isolamento: **Serializable**, **Repeatable Read**, **Read Committed** e **Read Uncommitted**.

5.3.3.1 Repeatable Read

Este é o nível de isolamento por defeito no InnoDB. Permite que todas as leituras na transação sejam consistentes, ou seja, o seu resultado seja o mesmo em qualquer ponto da transação independentemente de possíveis alterações causadas por outras transações.

Este modo estabelece uma snapshot na primeira leitura da transação e todas as leituras posteriores utilizam esta para garantir uma leitura consistente. Evita-se o uso de locks nas operações realizadas na transação, o que reduziria a concorrência.

Em operações de leituras bloqueantes (SELECT com cláusula FOR UPDATE ou LOCK IN SHARE MODE, descritas posteriormente)/update/delete, o lock depende que query é executada. Se for com uma única condição de procura, apenas o index record é bloqueado. Caso contrário, se for com uma condição de procura variável, são utilizados gap locks ou next-key locks para prevenir inserções, de outras sessions, sobre os espaçamentos na condição de procura.

5.3.3.2 Read Committed

Numa transação, cada leitura atualiza e lê a sua própria versão.

No caso de leituras bloqueantes (SELECT com cláusula FOR UPDATE ou LOCK IN SHARE MODE, descritas posteriormente)/update/delete, o lock é aplicado apenas sobre o index record. Assim, outra sessão pode inserir tuplos livremente, já que estes não se encontram bloqueados.

5.3.3.3 Read Uncommitted

As *queries* de seleção são executadas de forma não bloqueante, mas é possível que seja processada uma versão mais antiga de um tuplo. Ao utilizar este nível de isolamento, as leituras não são consistentes.

5.3.3.4 Serializable

Este nível de isolamento funciona de forma semelhante ao repeatable read. Mas, no caso do mecanismo de autocommit estar desativado, as queries de seleção são convertidas para SELECT ... LOCK IN SHARE MODE. Se a opção estiver ativa, a operação de seleção é, ela mesma, uma transação.

5.3.4 Durabilidade

Qualquer operação realizada no decorrer de uma transação persiste na base de dados, mesmo que hajam falhas no sistema ou cortes de energia. O InnoDB garante esta condição ao escrever os dados, de forma temporária, no *doublewrite buffer*, que oferece redundância na proteção dos dados contra possíveis falhas.

Desta forma, antes de escrever as *pages* para os respetivos *data files*, estas são escritas para numa zona de memória contígua. Após o *flush* dos dados para o *doublewrite buffer*, é que os dados são escritos de forma permanente no data file. Se ocorrer uma falha no

sistema operativo na escrita no ficheiro, pode aceder-se a uma cópia dos dados no buffer para uma nova tentativa na escrita no ficheiro.

5.4 Concorrência

O InnoDB implementa *row-level locking* e, se executado o comando **LOCK TABLES** utiliza *table locking*. Uma transação pode adquirir *shared locks* ou *exclusive locks*.

- **Shared lock** permite que a transação leia o tuplo;
- **Exclusive lock** permite que a transação que detém o lock modifique ou apague o tuplo.

Se uma transação T1 detém um *exclusive lock* sobre um tuplo, e se uma outra transação T2 solicita um lock (de qualquer tipo), esta transação terá de aguardar pelo final da transação T1 para poder aceder ao tuplo.

O InnoDB suporta ainda *multiple granularity locks* que permite a coexistência de locks sobre tuplos e locks sobre tabelas. Para tal, é definido um outro tipo de locks denominado *intention locks*. Estes são locks sobre tabelas que indicam qual o tipo de lock (shared ou exclusive) que a transação irá utilizar nos tuplos da respetiva tabela.

- **Intention shared (IS)** são utilizados quando a transação pretende adquirir shared locks sobre os tuplos da tabela. Este pode ser adquirido por várias transações;
- **Intention exclusive (IX)** são utilizados quando a transação pretende adquirir exclusive locks sobre os tuplos da tabela. Este bloqueará todas as transações que pretendam adquirir um lock sobre os tuplos na tabela.

Antes de uma transação adquirir um *shared lock/exclusive lock* sobre um tuplo de uma tabela, deve obter um *intention shared/intention exclusive lock* ou um lock de maior granularidade sobre a respetiva tabela.

Para adquirir um intention shared lock terá de se executar o seguinte comando:

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

Se se quiser obter um *intention exclusive lock* sobre a tabela em questão executa-se o seguinte comando:

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

As regras de obtenção de locks podem ser resumidas na tabela 5.1.

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

Tabela 5.1: Regras de obtenção de locks

Um lock é obtido por uma transação se este não entrar em conflito com nenhum dos outros locks. Se não for possível a transação obter um determinado lock, esta irá aguardar até que o lock seja libertado pela transação que o detém.

Se se tentar obter um lock que entre em conflito com outro lock e que provocaria um deadlock é lançado um erro.

Ao serem detetados deadlocks, é executado o rollback sobre as transações mais curtas (que inseriram/apagaram/atualizaram menores tuplos) e os locks detidos por estas libertados.

5.5 Comparação com o Oracle 11g

Todas as transações do Oracle estão conforme as propriedades básicas de uma transação (ACID).

Tal como o MySQL também é utilizado *two-phase commit* no Oracle.

Na granularidade da concorrência e nos níveis de isolamento, o Oracle é como o MySQL já que na granularidade suporta *row-locks* e *table-locks* e nos níveis de isolamento suporta *Repeatable Read*, *Read Uncommited*, *Read Commited* e *Serializable*.

O Oracle também deteta automaticamente situações de *deadlock* e resolve-as fazendo rollback de um dos statements que provoca deadlock.

Capítulo 6

Suporte para Bases de Dados Distribuídas

O InnoDB não suporta bases de dados distribuídas. Apesar disso, o *engine* permite a replicação homogénea de bases de dados e a replicação de bases de dados que corram sobre diferentes engines (por exemplo, sobre InnoDB e MyISAM).

6.1 Master-Slave

O mecanismo suportado pelo InnoDB, *Master-Slave*, permite a replicação de dados com um baixo grau de consistência (sem garantia de serialização). Neste mecanismo, um dos nós é *master* onde todos os *updates* são efetuados, e os restantes nós são *slaves*, para onde são propagados os dados do primeiro nó.

Para criar um novo nó *slave*, terá de fazer-se uma cópia, do *tablespace* e dos *ficheiros de log*, do nó master para o novo nó.
Para tal, deverá executar-se o seguinte comando:

```
mysqldump --all-databases --master-data > dbdump.db
```

Durante o período de cópia, o nó master deverá estar bloqueado.

A replicação é baseada no uso de *binary logs* onde são armazenadas todas as queries SQL que modificam dados na base de dados. O *binary log* só é atualizado quando uma transação termina com sucesso. Caso isso se verifique, o log é enviado do nó master para os nós slave para permitir a atualização dos dados replicados.

Existem duas formas de replicação:

- **Statement-based replication** - as queries SQL, depois de enviadas do nó master para os nós slave, são reproduzidas por estes;
- **Row-based replication** - o nó master indica aos nós slave quais os tuplos a adicionar/atualizar na respetiva tabela.

Para utilizar o mecanismo de *Master-Slave replication* deverá ativar-se o *binary logging*. Para tal, basta executar o seguinte comando: *–log-bin[=base_name]*. De seguida, deverá definir-se um ID único para todos os nós: *–server-id=#*.

Para melhorar a consistência da replicação entre os diferentes nós, devem executar-se o seguintes comandos, que garantem que o flush para disco do log buffer só é efetuado depois do commit: *innodb_flush_log_at_trx_commit=1*, *sync_binlog*=1.

6.2 XA Transactions

O InnoDB permite a execução de transações distribuidas utilizando o mecanismo **XA Transactions**. O processo para executar as transações segue o protocolo *two-phase commit*, garantido as propriedades ACID. As aplicações que utilizem este mecanismo, necessitam dos seguintes elementos:

- **Resource manager** - pode haver um ou mais, este possibilita acesso a recursos transacionais de cada servidor que participa na transação;
- **Transaction manager** - existe exactamente um, comunica com os restantes resource managers de modo a coordenar as transações que fazem parte da transação global.

6.2.1 Comandos

Existem os seguintes comandos para fazer uma transação XA:

```
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]
```

O valor *xid* presente nos comandos anteriores é o identificador global da transação.

6.2.2 Estados das Transações

As transações podem estar nos seguintes estados:

- **Active** - Depois de inicializada, através do comando XA START, a transação XA encontra-se neste estado;
- **Idle** - A transação encontra-se em espera. Uma transação está neste estado depois de executado o comando XA END. Para uma transação neste estado, ao executar-se o comando XA PREPARE, esta é colocada no estado prepared;
- **Prepared** - Uma transação neste estado, ao executar-se XA COMMIT é feito o commit desta. Ao executar-se o comando XA ROLLBACK é feito rollback da transação e esta é terminada.

6.3 MySQL Cluster

6.3.1 Visão geral

Existe uma versão adaptada do MySQL, o *MySQL Cluster*, focado em ambientes distribuídos, de alta escalabilidade e redundância. Como storage engine, esta versão utiliza o *NDBCLUSTER*, um storage engine que reside em memória [13].

O MySQL Cluster admite a existência de um agregado de bases de dados num sistema shared-nothing. Este sistema permite que o hardware utilizado seja menos caro/potente e que cada componente deste tenha a sua própria memória e disco, o que faz com que não exista um único ponto de falha.

Esta tecnologia consiste num conjunto de computadores (*hosts*), em que cada um pode correr vários processos (*nodes*) que podem incluir servidores MySQL, dados, servidores de gestão e ainda outros programas especializados de acesso a dados.

A figura 6.1 permite ter uma ideia geral da composição de um MySQL Cluster.

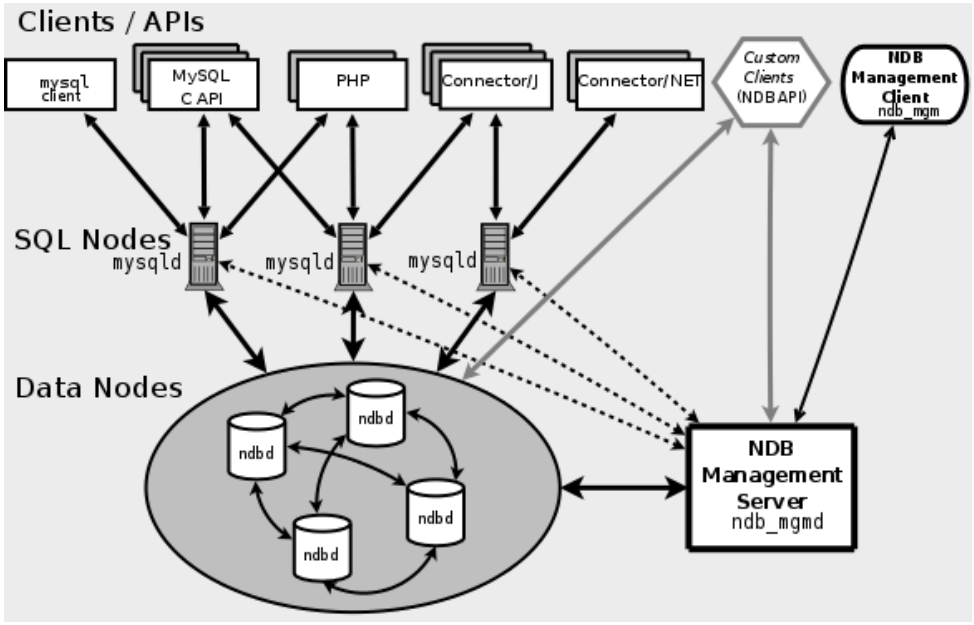

Figura 6.1: Componentes do MySQL Cluster

Os dados presentes nos *Data Nodes* podem ser replicados, permitindo que o cluster consiga gerir falhas de nós individuais sem afectar o restante sistema, sendo que apenas uma pequena parte das transações podem ser abortadas.
Cada nó pode ser terminado ou reiniciado, sendo depois possível que se junte novamente ao cluster.
Existem três tipos de nós num cluster:

- **Management node** - gere os restantes nós no cluster. É o primeiro nó que deve ser iniciado no cluster devido à sua funcionalidade;
- **Data node** - armazena dados do cluster. Existem tantos data nodes quanto o número de réplicas multiplicadas pelo número de fragmentos;

- **SQL node** - acede aos dados do cluster. Tipicamente é um servidor MySQL a correr sobre o storage engine NDBCLUSTER.

Cada réplica pertence a um data node, mas um data node pode ter várias réplicas. Normalmente as tabelas no *NDBCLUSTER* são particionadas automaticamente, mas é possível o utilizador definir as suas regras para particionamento.

Pode-se também introduzir o conceito de **node group**, que consiste num conjunto de um ou mais nós que guardam partições ou réplicas.
Cada nó presente num *node group* mantém uma réplica, sendo que o número de réplicas é igual ao número de nós num *node group*.

Na figura 6.2 pode-se verificar um cluster com quatro data notes organizados em dois node groups.

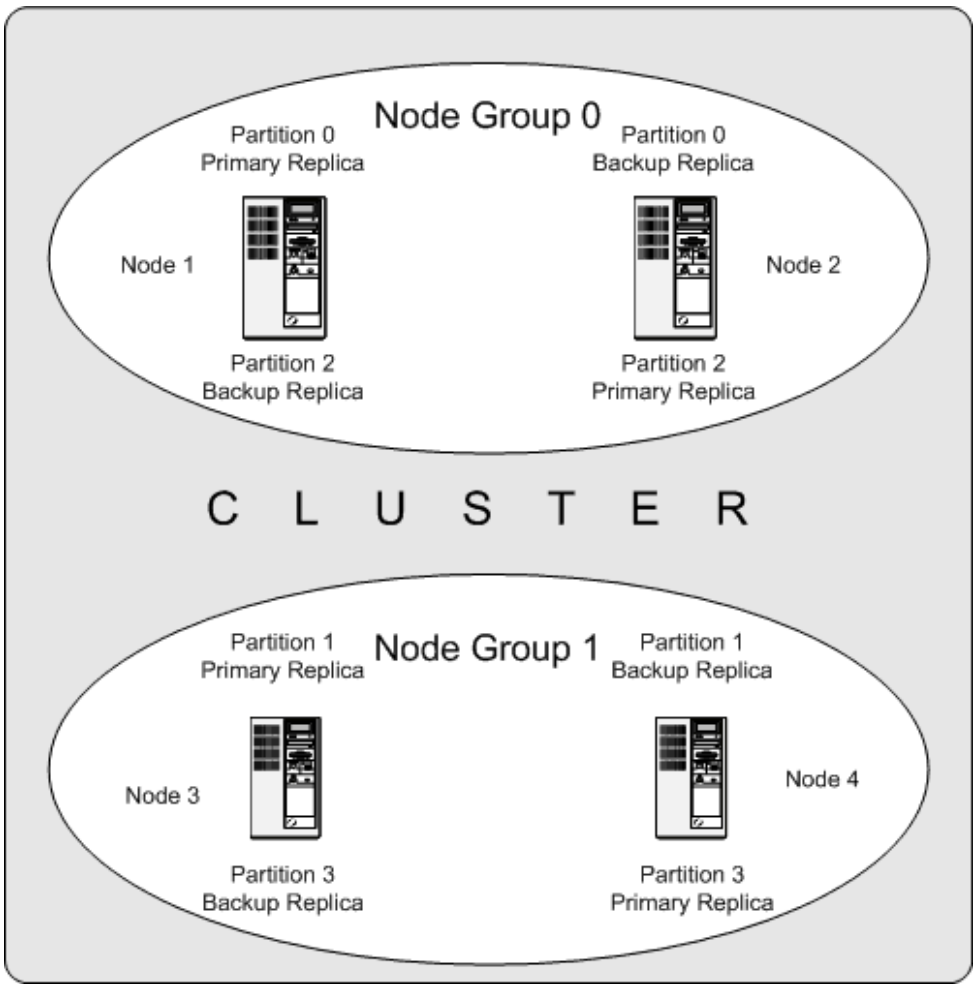

Figura 6.2: Cluster com quatro data nodes organizados em dois node groups

Os dados armazenados neste cluster estão divididos em quatro partições (0, 1, 2, 3), e cada partição está armazenada num mesmo node group.

Com esta estrutura é possível, mesmo que dois nós (de grupos diferentes) falhem, aceder a todos os dados do cluster através deste esquema de cópias. No entanto, se todos os nós de um dos grupos falharem, os restantes não serão suficientes para obter todos os dados no cluster.

6.3.2 Funcionamento

De seguida, será descrito o comportamento de um sistema MySQL Cluster ao serem efetuadas diferentes queries:

- **Query SELECT** - ao fazer-se uma query SELECT num SQL node, são efetuados (simultaneamente) pedidos a todos os Data nodes no cluster. Cada um dos Data nodes, depois de executar a devida operação localmente, retorna os dados para o SQL Node de onde proveio a query inicialmente;
- **Query INSERT/UPDATE** - ao fazer-se uma query INSERT ou UPDATE, o SQL node comunica com o Data node que contém parte dos dados que se pretende manipular e elege-o como Transaction Coordinator (TC) da transação corrente. De seguida, este nó executa a operação de Linear Two-Phase Commit Protocol junto com os restantes Data nodes para garantir a atomicidade da transação [6] [7].

6.3.3 Replicação

A replicação entre MySQL Clusters é assíncrona (guarda as alterações locais numa queue e periodicamente aplica estas alterações nos servidores remotos).

6.3.3.1 Ligação entre bases de dados Master e Slave

Para se poder utilizar a replicação no *MySQL Cluster*, deve ser criada uma conta *slave* no *master cluster* com os privilégios apropriados:

```
GRANT REPLICATION SLAVE ON *.* TO 'slave_user'@'slave_host' \
IDENTIFIED BY 'slave_password';
```

Sendo *slave_user*, *slave_host* e *slave_password* o utilizador, hostname e passwords de um replication slave, respetivamente. No nó slave deve ser indicado qual é o master:

```
CHANGE MASTER TO MASTER_HOST='master_host', \
MASTER_PORT=master_port, MASTER_USER='slave_user', \
MASTER_PASSWORD='slave_password';
```

Sendo *master_host*, *master_port* o hostname e porto onde está presente o replication master e *slave_user*, *slave_password* o utilizador e password a utilizar pelo slave para se ligar ao master.

O *MySQL Cluster* não permite definir locks de forma distribuída. A tabela de locks funciona apenas no nó SQL na qual o lock foi emitido, sendo que, no caso de se emitir um comando *ALTER TABLE*, não existe um lock global sobre os vários servidores.

6.3.4 Limitações da gestão de transações

- O *NDBCLUSTER* suporta apenas transações READ COMMITED;
- Não existem transações parciais nem rollbacks parciais de transações;
- Não se consegue gerir transações grandes;
- Não é possível garantir consistência na função *COUNT()*, sendo que executar esta função leva a que se obtenha valores intermédios se, concorrentemente existe uma transação a fazer modificações na base de dados.

6.4 Comparação com o Oracle 11g

O Oracle permite bases de dados *homogéneas* e *heterogéneas*, e utiliza *two-phase commit*. Permite igualmente transações distribuídas.

O Oracle também utiliza *replicação assíncrona*. A vantagem de ambos utilizarem este tipo de replicação, invés da replicação síncrona, é que no caso de um sistema que utilize replicação síncrona, é necessário garantir que não existem falhas num nó para que a replicação seja executada com sucesso logo, se um nó estiver em baixo, tem de se aguardar que este volte ao ativo para terminar a replicação.

Capítulo 7

Outras características do sistema estudado

Existem outras funcionalidades que não se enquadram na matéria da cadeira de Sistemas de Bases de Dados e como tal não são mencionadas nas secções anteriores. No entanto é importante apresentá-las pois também são importantes para o ecossistema do MySQL.

7.1 Data types

O MySQL suporta diferentes tipos de dados [8] para números:

- INT
- TINYINT
- SMALLINT
- MEDIUMINT
- BIGINT
- FLOAT
- DOUBLE
- DECIMAL

Tipos para datas e horas:

- DATE
- DATETIME
- TIMESTAMP
- TIME
- YEAR

E tipos para strings:

- CHAR
- VARCHAR
- BLOB/TEXT
- TINYBLOB/TINYTEXT

- MEDIUMBLOB/MEDIUMTEXT
- LONGBLOB/LONGTEXT
- ENUM

7.2 Stored programs, views e cursors

Existe suporte para stored programs que incluem: stored procedures, stored functions, triggers e eventos. A linguagem utilizada para estes stored programs é própria do MySQL mas é muito parecida a SQL/PSM. Uma *stored procedure* não tem valor de retorno. Uma *stored function* funciona como uma função pertencente ao sistema, retorna um valor no final da chamada. Os eventos são tarefas que o sistema corre de acordo com um horário.

São suportadas *views* e *updatable views* (views que podem ser usadas nos comandos INSERT, UPDATE e DELETE).

Existe ainda suporte para *cursors* nos stored programs.

7.3 API Middleware

O MySQL tem conectores disponibilizados no próprio website para ODBC, JDBC, .NET, Python, C++ e C. Para PHP a integração é diretamente feita na linguagem através de um biblioteca que acompanha as instalações da linguagem. Existem ainda API's desenvolvidas por terceiros para as linguagens já cobertas e para outras linguagens.

7.4 Segurança

São suportados múltiplos utilizadores com vários níveis de privilégios. O sistema usa *Access Control Lists* (ACL's) para todas as conexões, queries e outras operações que um utilizador tente executar. Permite ainda definir quotas de utilização para cada um destes utilizadores de forma a controlar o uso da base de dados.

Suporta ainda SSL nas conexões de forma a torná-las seguras.

7.5 XML

Não existe suporte para armazenar XML diretamente, no entanto, é possível importar e exportar XML a partir de ficheiros. São disponibilizadas funções para navegar no XML (se este for guardado como texto) usando XPath.

7.6 Semantic Web

Não existe suporte nativo para web semântica, contudo existem extensões desenvolvidas por terceiros que procuram adicionar este tipo de pesquisas às tabelas de MySQL. Este é o caso do *SPARQL*. [10]

7.7 Ferramentas

A acompanhar o sistema de base de dados vem uma ferramenta gráfica chamada *MySQL Workbench*. Esta disponibiliza desenvolvimento SQL com gestão das conexões, editor de SQL e execução de queries; modelação de dados com visualização gráfica do modelo da base de dados; administração do servidor; migração de dados. Vem ainda outras ferramentas de gestão da base de dados como ferramentas de backup, gestão de plugins, configuração do servidor de base de dados, entre outras.

Existem ferramentas desenvolvidas por terceiros bastante utilizadas. Entre estas está o phpmyadmin, uma interface gráfica totalmente desenvolvida em PHP pelo que é acessível sem nada instalado e o Navicat, um programa proprietário de gestão de bases de dados disponível para vários sistemas operativos.

7.8 Cloud

Existe suporte para executar um servidor de MySQL em serviços cloud como o *Amazon EC2* através de *Virtual Machine Image*. A Amazon oferece também um serviço chamado *Amazon Relational Database Service* que suporta MySQL na cloud. Existem outras empresas que oferecem serviços semelhantes.

7.9 Comparação com o Oracle 11g

Assim como o MySQL, o Oracle tem suporte para stored programs, views e cursors. Estes não usam a mesma linguagem que o sistema em estudo, usam PL/SQL.

Ao contrário do MySQL, o sistema de bases de dados Oracle oferece suporte nativo para todo o standard XML e Web Semântica através do grafo semântico RDF.

Na parte da segurança tem os mesmos mecanismos do MySQL mas como é um sistema bem mais complexo e robusto oferece muitas outras facetas como firewall e outros mecanismos de proteção.

O Oracle suporta ainda o mapeamento objecto-relacional, funcionalidade não oferecida pelo sistema MySQL.

Bibliografia

[1] DB-Engines Ranking - popularity ranking of database management systems (http://db-engines.com/en/ranking).

[2] MySQL :: MySQL 5.7 Reference Manual :: 14 The InnoDB Storage Engine (http://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html).

[3] MySQL :: MySQL 5.7 Reference Manual :: 15.2 The MyISAM Storage Engine (http://dev.mysql.com/doc/refman/5.7/en/myisam-storage-engine.html).

[4] MySQL :: MySQL 5.7 Reference Manual :: 18.2.4.1 LINEAR HASH Partitioning (http://dev.mysql.com/doc/refman/5.7/en/partitioning-linear-hash.html).

[5] MySQL :: MySQL 5.7 Reference Manual :: 8.3.1.11 ORDER BY Optimization (http://dev.mysql.com/doc/refman/5.7/en/order-by-optimization.html).

[6] MySQL :: MySQL Internals Manual :: 6.2 Current Situation (http://dev.mysql.com/doc/internals/en/transactions-current-situation.html).

[7] MySQL Cluster Transaction Data Handling - kjalleda (https://sites.google.com/site/kjalleda/mysqlclustertransactiondatahandling).

[8] MySQL Data Types (http://www.tutorialspoint.com/mysql/mysql-data-types.htm).

[9] MySQL (http://en.wikipedia.org/wiki/MySQL).

[10] Notes on Adding SPARQL to MySQL (http://www.w3.org/2005/05/22-SPARQL-MySQL).

[11] Oracle Database Concepts (http://docs.oracle.com/cd/E11882_01/server.112/e25789/toc.htm).

[12] Sergey Petrunia's blog ■ How MySQL executes ORDER BY (http://s.petrunia.net/blog/?p=24).

[13] What Is MySQL Cluster? - O'Reilly Answers (http://answers.oreilly.com/topic/1862-what-is-mysql-cluster).