FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Sistemas de Bases de Dados

Relatório - MySQL

Realizado pelo Grupo 06:
João Pedro Lucas de Almeida Fernandes, 41909
Nuno Henrique Sarmento Fernandes Melo, 41968
Sofia Dias Seixas, 41970

Índice

1. Introdução

1.1. Sobre este Relatório

Este relatório foi desenvolvido no âmbito da cadeira de *Sistemas de Bases de Dados*, como parte do trabalho final. O objetivo do trabalho era a análise teórica de um *Sistema de Gestão de Bases de Dados*, e a posterior comparação deste ao conhecido *SGBD* Oracle 11g.
O relatório abrange os tópicos discutidos nas aulas da cadeira e pretende descrever, com algum detalhe, o *SGBD* estudado. Foi escolhido o *SGBD* MySQL como objeto de estudo devido à sua popularidade e forte utilização em ambientes empresariais e pessoais.
O relatório está organizado de uma forma similar à ordem cronológica pela qual os tópicos foram abordados na cadeira.

1.2. História e Aplicabilidade do MySQL

O *MySQL* é um sistema de gestão de base de dados relacional, atualmente mantido e desenvolvido pela Oracle. Foi criado pela empresa Sueca *MySQL AB*, em 1995, com um propósito original de ser utilizado para uso pessoal.
É um sistema amplamente utilizado em contexto de *web development*, e *data centers*. Tem também uma *API* muito completa, algo que o torna num alvo atraente para integração em outras linguagens e sistemas.
Foi estudado o *storage engine* InnoDB do MySQL, apesar de o MySQL ter operado por muitos anos sob um *engine* com maior número de limitações, e com menor escalabilidade, denominado *MyISAM*. O InnoDB suporta transações *ACID*, *foreign keys*, *crash recovery*, divisão de *tablespaces* por ficheiro, entre outros.

2. Armazenamento e File Structure

2.1. Buffer Management

O InnoDB mantém uma zona de armazenamento denominada *buffer pool* para fazer caching de dados e índices em memória. Esta *buffer pool* também faz *caching* de dados alterados por inserções ou alterações, para que escritas a memória persistente seja feita em *batch*, para otimizar performance (*dirty buffers*).
Esta *buffer pool* está estruturada numa única lista ligada de *pages,* ou *blocos*, que são conjuntos de *rows*, ou *registos*.
Esta lista é gerida utilizando uma variação da política de substituição *LRU* (*Least Recently Used*), denominada *midpoint insertion strategy*.
A lista é dividida logicamente em duas sub-listas:

- A sub-lista *new*: corresponde à "primeira metade" da lista, isto é, da *cabeça* da lista até ao *midpoint* denominado ($\frac{5}{8}$ do tamanho da lista, por defeito). É onde estão os *blocos* utilizados mais recentemente.
- A sub-list *old*: corresponde à "segunda metade" da lista, isto é, desde o *midpoint* denominado até à *cauda* da lista ($\frac{3}{8}$ do tamanho da lista, por defeito). É onde estão os *blocos* utilizados menos recentemente.

Quando é lido um novo bloco, o algoritmo opera da seguinte forma:

1. Se não houver espaço para colocar o *bloco* novo na lista, o *bloco* que está na *cauda* da lista é retirado da *buffer pool*;
2. O novo bloco é inserido no *midpoint* da lista, isto é, à *cabeça* da sub-lista *old*.
    a. Se o *bloco* tiver sido lido porque foi pedido por um *query*, o *bloco* é movido para a cabeça da lista.
    b. Se tiver sido lido por efeito de algum *read-ahead* (pré-leitura de *blocos* que provavelmente serão lidos em breve), é mantido no *midpoint*.

Se o *bloco* com os dados desejados já estiver na *buffer pool*, este é simplesmente movido para a *cabeça* da lista.
É fácil ver que este algoritmo produz um efeito de envelhecimento, pois à medida que são lidos *blocos* para a *cabeça* da lista, são "empurrados" outros *blocos* lidos com menos frequência para a cauda da lista.

É possível configurar a *buffer pool* do InnoDB para vários efeitos. Segue-se uma lista de parâmetros configuráveis:

- **innodb_buffer_pool_size**: determina o tamanho da *buffer pool*. Aumentar o tamanho da *buffer pool* melhora substancialmente o desempenho da base de dados.
- **innodb_buffer_pool_instances:** este parâmetro permite dividir a *buffer pool* em várias *regiões*. Cada *região* detém a sua lista *LRU*, e estruturas de dados relacionadas. Serve para reduzir a contenção da estrutura de dados de *blocos*, de forma que transações concorrentes possam aceder a dados na *buffer pool*. Os *blocos* são atribuídos a uma *região* utilizando uma função de *hashing*.
- **innodb_old_blocks_pct:** este parâmetro determina o *midpoint* da lista, ou mais concretamente, o tamanho percentual da sub-lista *old*. O seu valor por defeito é 37 ($\frac{3}{8}$).
- **innodb_old_blocks_time:** este parâmetro determina o tempo mínimo durante o qual um *bloco* deve ser *old*, antes de poder voltar a ser considerado para a sub-lista *new*. O seu valor por defeito é 0.
  É utilizado para prevenir que um ocasional *table scan* encha a sub-lista *new* com dados que serão lidos apenas uma vez, ou com muita frequência durante um curto espaço de tempo e depois não serão lidos novamente.
  - Pode ser configurado em *runtime*, útil para *full table scans*, por exemplo:

```
SET GLOBAL innodb_old_blocks_time = 1000;
... executar queries sob tabelas inteiras ...
SET GLOBAL innodb_old_blocks_time = 0;
```

Para sistemas *64-bit* com grandes quantidades de memória disponível, é muito eficiente balançar os valores de *innodb_buffer_pool_size* e *innodb_buffer_pool_instances*, de forma a criar várias *regiões* de tamanho substancial que funcionam eficientemente em concorrência.

2.2. Sistema de Ficheiros

O InnoDB guarda os dados, *schema* e *índices* de tabelas em ficheiros denominados *tablespaces*. Estes ficheiros são capazes de guardar dados de uma, ou várias tabelas, assim como os índices a ela(s) associado(s). São guardados em ficheiros com formato **.ibd**.

O InnoDB opera em dois modos possíveis: um que utiliza um único *tablespace* denominado *system tablespace*, onde são guardados e geridos *schema data*, *índices* e todos os dados das tabelas, ou num modo que utiliza o sistema de ficheiros do sistema operativo para separar tabelas e *índices* por ficheiros, no seu próprio *tablespace*.
O modo de operação é determinado através da análise do valor atribuído à opção ***innodb_file_per_table***.
Em ambos os modos existe um *system tablespace* (*ibdata file*), pois neste são também mantidos *metadados* relevantes à estrutura das tabelas. Alguns exemplos são:

- ***Data Dictionary***: o dicionário de dados contém metadados sobre a estrutura de objetos como *tabelas*, *índices* e *atributos*.
- ***Undo log***: um registo que mantém informação sobre dados alterados por *transações* ativas. Importante para garantir *consistent reads* a *transações concorrentes*.
- ***Change buffer***: uma estrutura de dados que mantém informações sobre alterações a *páginas* em *índices secundários*, quando a *página* pretendida não se encontra na *buffer pool*.

Historicamente, o InnoDB apenas operava sob o *system tablespace*, no entanto a necessidade de melhorar o performance, e de aproveitar melhor o espaço do disco levou ao desenvolvimento do modo *File-Per-Table* descrito anteriormente.

2.3. Partições

O MySQL tem um mecanismo de *partições*, com particionamento definido pelo utilizador. O utilizador especifica uma *função de particionamento*, que pode ser definida de diferentes formas, utilizada para distribuir os dados de uma tabela pelas diversas *partições* definidas pelo utilizador. Estas funções devem ser *não-aleatórias* e *não-constantes*, de forma a garantir distribuição uniforme. A forma geral da definição destas funções é a seguinte:

```
CREATE TABLE [TABLE-NAME] (
          ... definição da tabela ...
)PARTITION BY FUNCTION-TYPE([table-field]) [(
    PARTITION [partition-1] VALUES [CONDITION],
    ...              ...                  ...,
    PARTITION [partition-n] VALUES [CONDITION]
)];
```

Existem diversos tipos de *funções de particionamento* que podem ser definidas.

- ***Particionamento por Range***: em particionamento por *intervalo* são associados intervalos de valores a *partições*, e a análise do valor parâmetro determina a partição onde deve ser guardado o valor. Um exemplo de uso pode ser um *particionamento* de uma tabela por um campo de data de check-in num hotel, por década.

```
CREATE TABLE checkin (
      ... CAMPOS ...,
      checkin_date DATE NOT NULL
) PARTITION BY RANGE(YEAR(checkin_date)) (
      PARTITION p1 VALUES LESS THAN (2000),
      PARTITION p2 VALUES LESS THAN (2010),
      PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

- ***Particionamento por Lista***: em particionamento por *lista* é associado a uma partição um sub-conjunto de um conjunto de valores discretos. Podiamos querer particionar a tabela anterior por “meses com o mesmo número de dias”, em vez de por década.

```
CREATE TABLE checkin ( ... CAMPOS ...,
      checkin_date DATE NOT NULL
) PARTITION BY RANGE(MONTH(checkin_date)) (
      PARTITION p31 VALUES IN (1, 3, 5, 7, 8, 10, 12),
      PARTITION p30 VALUES IN (4, 6, 9, 11),
      PARTITION pOther VALUES IN (2)
);
```

- ***Particionamento por Hashing***: em particionamento por *hash*, é escolhida pelo utilizador uma função de distribuição válida em MySQL. A *hash* em si é calculada pelo MySQL, no entanto o utilizador deve especificar um campo, ou expressão inteira não-negativa, sob o qual deve ser calculada a *hash*. Deve ser também especificado pelo utilizador o número de partições pretendido (se não for especificado qualquer valor, é utilizado o valor por defeito 1). Utilizando o exemplo anterior:

```
CREATE TABLE checkin (
      ... CAMPOS ...,
      checkin_date DATE NOT NULL
) PARTITION BY HASH(YEAR(checkin_date))
PARTITIONS 10;
```

  - Existe uma variação deste particionamento, denominado *Linear Hashing*. Sintaticamente, a utilização do algoritmo é similar à do *hashing* normal, mas deve ser utilizada a keyword *LINEAR*.

```
CREATE TABLE checkin (
      ... CAMPOS ...,
      checkin_date DATE NOT NULL
) PARTITION BY LINEAR HASH(YEAR(checkin_date))
PARTITIONS 10;
```

    A diferença entre este algoritmo e o anterior é que o anterior utiliza a função *módulo do inteiro* para distribuir a *hash*, enquanto que esta utiliza um algoritmo de potências-de-dois.

- ***Particionamento por KEY***: particionamento por *KEY* é similar ao *particionamento por hashing*, no entanto o utilizador não necessita definir qualquer expressão para a função. *Particionamento por KEY* recebe 0 ou mais colunas, que serão utilizadas para calcular a *hash* final.

```
CREATE TABLE checkin (
      ... CAMPOS ...,
      checkin_date DATE NOT NULL
) PARTITION BY KEY()
PARTITIONS 10;
```

2.4. Organização dos tuplos

Os dados das tabelas do InnoDB estão divididos por blocos. Em cada tabela, os blocos que a constituem estão organizados numa estrutura de dados em árvore, mais especificamente, árvores B+. Estes blocos são acedidos através de um clustered index, organizado de acordo com os atributos da chave primária, numa árvore B+. Ao invés de guardar apontadores para os blocos, as folhas do índice guardam os blocos em si.

Os atributos de tamanho variável, como por exemplo *varchar* ou *blob*, são uma exceção a esta regra. Por não caberem num só bloco, são guardados em blocos alocados separadamente, chamados ***blocos de overflow***.

É possível escolher como são representados estes atributos de tamanho variável através da cláusula ***row_format*** dos comandos *create table* e *alter table*. Por exemplo:

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

Por questões de compatibilidade com bases de dados antigas, o row format por defeito é o ***compact row format***, em que são guardados os primeiros 768 bytes dos atributos de tamanho variável no bloco correspondente da árvore B+, sendo os restantes bytes guardados em blocos de overflow.
Existem, no entanto, dois novos tipos de row format, o ***dynamic*** e o ***compressed***, que fazem parte do ***Barracuda file format***. Quando uma tabela é criada com *row_format=dynamic* ou *row_format=compressed*, os atributos demasiado grandes são guardados completamente em páginas de overflow, e é guardado no respectivo bloco da árvore B+ um apontador de 20 bytes para a página de overflow. A principal diferença entre o *dynamic* e o *compressed row format*, é que o dynamic row format mantém a eficiência de guardar o tuplo inteiro na árvore B+, se este couber todo na página.

Para criar tabelas com *dynamic* ou *compressed row format*, é necessário alterar a opção innodb_file_format para Barracuda:

--innodb_file_format=Barracuda

O file format só se aplica a tabelas que têm o seu próprio *tablespace*, pelo que a opção *innodb_file_per_table* deve estar activa.

2.5. Comparação com o Oracle 11g

Para fazer caching de blocos de dados, o Oracle tem uma zona em memória chamada ***buffer cache*** que, tal como o InnoDB, usa uma lista que utiliza a política de substituição LRU, para aceder aos blocos. A principal diferença entre os dois é que o Oracle guarda na sua lista apontadores para os blocos que se encontram na *buffer cache*, enquanto que o InnoDB guarda os blocos em si.

O sistema de ficheiros do Oracle também guarda os dados em tablespaces, mas ao contrário do InnoDB, pode guardar várias tabelas num tablespace.

O Oracle também tem suporte para particionamento. Os tipos de particionamento do Oracle são os mesmos que os do MySQL, mas o Oracle não suporta particionamento por KEY.

Enquanto que o InnoDB organiza os tuplos em árvores B+, o Oracle organiza os tuplos em heap.

Ao contrário do MySQL, o Oracle suporta *multitable clustering*.

3. Indexação e hashing

3.1. Estruturas de Indexação

Todos os índices do InnoDB são árvores B+ onde os registos são guardados nos blocos das folhas. O tamanho por defeito de cada bloco de um índice são 16 KB, mas podemos especificar outro tamanho, configurando *--innodb_page_size=#k*, com *# = 4* ou *# = 8*, antes de criarmos a instância de uma base de dados. Quando são inseridos novos registos, o InnoDB tenta deixar 1/16 do bloco livre para futuras inserções e alterações.

O InnoDB só permite a criação de índices que usam árvores B+ como estrutura de indexação. No entanto, existem outros *storage engines* que suportam a criação de índices hash, nomeadamente o *memory storage engine*.

3.2. *Create Index*

A sintaxe do MySQL para criar um índice é:

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

- *index_col_name*: os atributos chave do índice.
- *index_type*: estrutura de indexação a utilizar, no caso do InnoDB a unica opção é *btree*.
- index_option, algorithm_option, lock option: parâmetros opcionais.

Um índice ***unique*** cria uma restrição na qual todos os valores no índice devem ser distintos. Se tentarmos adicionar um novo tuplo com um valor chave que já existe, ocorre um erro.

Um índice ***fulltext*** só pode conter atributos do tipo *char*, *varchar* ou *text*. Estes índices facilitam as pesquisas do tipo “Full-Text Search”.

Os índices ***spatial*** são suportados apenas pelo MyISAM storage engine. Se tentarmos criar um índice *spatial* utilizando o InnoDB, ocorre um erro.

3.3. *Clustered Indexes*

Todas as tabelas do InnoDB têm um ***clustered index*** associado, onde os dados dos tuplos estão guardados. Tipicamente, é usada a ***primary key*** (chave primária) das tabelas para criar o índice, mas nem sempre é possível. O InnoDB usa os seguintes critérios para criar o *clustered index* de uma tabela:

- Se a tabela tiver uma chave primária definida, o InnoDB usa-a como chave do *clustered index*.
- Se a tabela não tiver uma chave primária definida, o MySQL procura o primeiro *unique index* (se existir) onde todos os atributos chave são *not null*, e o InnoDB usa-o *como clustered index*.
- Se a tabela não tiver uma chave primária definida, nem um *unique index* adequado, o InnoDB gera um ID interno que será utilizado como chave no *clustered index*.

O clustered index faz com que o acesso aos tuplos seja rápido, visto que a procura no indíce leva directamente ao bloco com todos os dados do tuplo. Se a tabela for grande, a arquitectura do clustered index pode poupar uma operação de I/O comparando com outras organizações de tuplos, que guardam os dados dos tuplos em blocos diferentes daqueles onde se encontram os registo do índice.

3.4. *Secondary Indexes*

Todos os índices à parte do clustered index são chamados ***secondary indexes*** (índices secundários). No InnoDB, cada registo num índice secundário contém os atributos da chave primária do respectivo tuplo, bem como os atributos especificados como chave do índice secundário. O InnoDB usa este valor da chave primária para procurar o tuplo no *clustered index*.

Se a chave primária for muito longa, os índices secundários ocupam mais espaço, pelo que é vantajoso ter chaves primárias pequenas.

3.5. *Adaptive Hash Indexes*

Tal como já foi dito anteriormente, o InnoDB só permite a criação de índices que usam árvores B+ como estrutura de indexação. No entanto, existe uma funcionalidade chamada ***adaptive hash index*** (AHI), em que o InnoDB cria uma espécie de ”base de dados em memória”. Esta funcionalidade pode ser configurada através da opção *--innodb_adaptive_hash_index=#* (*# = on*, *off*).

Os índices em hash são construídos com base num índice em árvore B+ já existente para a mesma tabela. Com base no padrão de pesquisas observado, o MySQL constrói um índice em hash usando um prefixo da chave definida na árvore B+. O prefixo da chave pode ser de qualquer comprimento, e é possível que só alguns valores da árvore B+ apareçam no índice em hash. Os índices em hash são construídos “on demand” para os blocos que são acedidos frequentemente.

Os índices em hash são usados apenas para comparações de igualdade que usam os operadores = ou <>, e nunca para operadores como < ou > que procuram um intervalo de valores.

3.6. Comparação com o Oracle 11g

O MySQL apenas suporta índices persistentes sob Árvores-B+, que são os índices utilizados por defeito pelo Oracle.
No entanto, ao contrário do MySQL, o Oracle suporta índices *bitmap*, que tornam *queries* que fazem contagens, ou que fazem cálculos sobre colunas limitadas a valores discretos, muito mais eficientes. O Oracle também não suporta *hash indexes*, tal como o MySQL, no entanto o MySQL fornece o mecanismo de *Adaptive Hash Index* para melhorar substancialmente o custo de execução de alguns *queries*.

4. Processamento e otimização de perguntas

O MySQL produz um conjunto de otimizações sob um *query* antes de o executar.
De uma forma geral, começa por analisar se os dados pertinentes ao *query* solicitado se encontram em *cache* (*buffer pool*), e em caso afirmativo devolve logo o resultado. Se não se encontram em *cache*, começa por desenhar um *plano de execução* do *query*, que envolve um série de otimizações sob o *query*. É possível analisar o *plano de execução* gerado pelo InnoDB através do comando *EXPLAIN*, que tem a forma geral:

```
{EXPLAIN | DESCRIBE | DESC}
    [explain_type]
    {explainable_stmt | FOR CONNECTION connection_id}
```

É útil notar que os comandos *EXPLAIN*, *DESCRIBE* e *DESC* são sinónimos, e podem ser utilizados para o mesmo efeito. Os parâmetros desta operação são os seguintes:

- ***explain_type***: Utilizado para especificar que informação deve ser produzida quando for gerado o plano de execução. Os valores possíveis são:
  - ***EXTENDED***: Adiciona alguma informação extra ao plano de execução, como a percentagem estimada de colunas da tabela que serão afetadas pelo query fornecido. Este parâmetro é obsoleto em versões mais recentes do MySQL, e é *enabled* em todas as chamadas a *EXPLAIN*.
  - ***PARTITIONS***: Adiciona informação relevante ao plano de execução quando o query é efetuado sob tabelas particionadas. Este parâmetro é obsoleto em versões mais recentes do MySQL, e é *enabled* em todas as chamadas a *EXPLAIN*.
  - ***FORMAT***: Esta opção permite especificar o formato em que a informação deve ser apresentada utilizando *FORMAT=format_type*. As opções para *format_type* são:
    - *TRADITIONAL*: O formato que será apresentado por defeito, se esta opção for omitida. Apresenta os dados num formato tabular.
    - *JSON*: Apresenta os dados em *JSON*. Útil se for pretendido o processamento destes dados por aplicações exteriores.
- ***explainable_stmt***: A expressão *SQL* a ser avaliada. Expressões válidas são operações *SELECT*, *DELETE*, *INSERT*, *REPLACE*, *UPDATE*.
  Uma alternativa a este parâmetro é o parâmetro *FOR CONNECTION* ***connection_id***, que não recebe nenhum *statement* a ser avaliado, mas sim avalia o *statement* a ser executado num dado momento na ligação com o id fornecido.

4.1. Otimização de cláusulas *WHERE*

O MySQL otimiza as cláusulas *WHERE* aplicando algumas transformações e substituições às condições estabelecidas na cláusula. Algumas dessas otimizações são:

- Remoção de parêntesis desnecessários.

```
  De: ((a AND b) AND c OR (((a AND b) AND (c AND d))))
Para: (a AND b AND c) OR (a AND b AND c AND d)
```

- Remoção de condições constantes.

```
  De: (B>=5 AND B=5) OR (B=6 AND 5=5) OR (B=7 AND 5=6)
Para:  B=5 OR B=6
```

- Resolução de constantes.

```
  De: (a<b AND b=c) AND a=5
Para:  b>5 AND b=c AND a=5
```

- Expressões constantes utilizadas por *índices* são avaliadas apenas uma vez.
- Deteção prévia de *perguntas SELECT* onde as expressões de condições constantes são impossíveis, caso em que o retorno é feito imediatamente, de um conjunto de registos vazio.
- A cláusula *HAVING* é juntada à cláusula *WHERE* se não forem utilizadas funções de agregamento, ou uma cláusula *GROUP BY*.
- Quando possível, o MySQL lê registos diretamente de um índice sem ter de consultar o ficheiro de dados.

É possível também ativar uma opção *SQL_SMALL_RESULT*, onde o MySQL utiliza uma tabela temporária criada em memória, para reduzir significativamente o tempo de execução do *query*.

4.2. Otimização de pesquisas em Intervalos (*range*)

O MySQL procura por condições no *query* que estabeleçam intervalos de valores sob *colunas* utilizadas na definição de índices, para que possa reduzir substancialmente o número de registos a analisar. Dispõe de uma função *range* para o efeito.

O processamento destas condições depende do tipo de *índice* a ser utilizado, nomeadamente, se é um *índice de uma coluna* ou um *índice multi-coluna*.

4.2.1. Índices de uma coluna

Como o InnoDB utiliza índices em árvores B+, uma condição é considerada um intervalo se for uma comparação entre a chave de um índice e uma constante, utilizando qualquer um dos seguintes operadores: =, <=>, IN(), *IS NULL*, *IS NOT NULL*, >, <, >=, <=, *BETWEEN*, !=, ou <>, assim como o operador *LIKE*, desde que o seu argumento seja uma constante, e não comece com um caractér *wildcard*.
O MySQL tenta extrair *condições de intervalo* da cláusula *WHERE*, por cada um dos índices existentes.
O processo de extração funciona da seguinte forma:

1. São substituídas por *TRUE* as condições que não podem ser usadas para construir *condições de intervalo*.
2. São juntas condições que são sempre *TRUE* ou *FALSE*.
3. São unidas condições que produzem intervalos sobrepostos.
4. E finalmente, são removidas condições que produzem intervalos vazios.

Um exemplo de execução deste processo, executado sob o seguinte query, onde *key* representa uma coluna que é chave num índice, e *nonkey* é um atributo sem índice:

```
SELECT * FROM t1 WHERE
  (key < 'abc' AND (key LIKE 'abcde%' OR key LIKE '%b')) OR
  (key < 'bar' AND nonkey = 4) OR
  (key < 'uux' AND key > 'z');
```

1. São substituídos por *TRUE* as condições:
    a. *nonkey = 4*: Como *nonkey* não é chave de um índice, não produzirá um intervalo relevante.
    b. *key LIKE '%b'*: Como o argumento do *LIKE* começa com a *wildcard* %, também não poderá ser considerado.

```
(key < 'abc' AND (key LIKE 'abcde%' OR TRUE)) OR
(key < 'bar' AND TRUE) OR
(key < 'uux' AND key > 'z')
```

2. São juntas as seguintes condições:

a. *(key LIKE 'abcde%' OR TRUE)*: Pois será sempre *TRUE*.
b. *(key < 'uux' AND key > 'z')*: Pois será sempre *FALSE*.

```
(key < 'abc' AND TRUE) OR (key < 'bar' AND TRUE) OR (FALSE)
Que é equivalente a:
(key < 'abc') OR (key < 'bar')
```

3. São unidas operações que produzem intervalos sobrepostos:

```
(key < 'bar')
```

O resultado deste algoritmo é um número reduzido de *registos* a analisar pelo algoritmo de pesquisa.

4.2.2. Índices Multi-Coluna

O algoritmo utilizado com *índices multi-coluna* é similar ao descrito anteriormente, no entanto é mais restritivo pois deve respeitar a estrutura do índice.
Os intervalos válidos para esta extração são encontrados da seguinte forma, onde a nossa chave de índice de exemplo é *key=(key_part1*, *key_part2*, *key_part3, …, key_partN)*:

1. *key_part1* for comparada com uma constante utilizando qualquer um dos seguintes operadores: =, <=>, IN(), *IS NULL*, *IS NOT NULL*, >, <, >=, <=, *BETWEEN*, !=, ou <>, assim como o operador *LIKE*, desde que o seu argumento seja uma constante, e não comece com um caractér *wildcard*.
2. O otimizador tenta (de forma sequencial) utilizar mais *key parts*, desde que a sua operação de comparação seja =, <=>, ou *IS NULL*.
    a. Se encontrar uma *key part* que tenha como operação de comparação >, <, >=, <=, !=, <>, *BETWEEN*, ou *LIKE*, esta é utilizada mas o otimizador não utiliza mais *key parts*.

Um exemplo disto é a seguinte cláusula, sob uma chave *key=(key_part1, key_part2, key_part3)*:

```
key_part1 = 'foo' AND key_part2 >= 10 AND key_part3 > 10
```

Que produzirá o seguinte intervalo, onde *key_part3* já não é considerada pois *key_part2* faz uma comparação com um operador >=:

```
('foo',10,-inf) < (key_part1,key_part2,key_part3) < ('foo',+inf,+inf)
```

4.2.3. Otimização Index Merge

O método *Index Merge* é utilizado para juntar os resultados de várias pesquisas *range* (como as definidas anteriormente) num único resultado.
Implementa três possíveis algoritmos de união:

- ***Index Merge Intersection***: Utilizado quando uma cláusula *WHERE* foi convertida para várias *condições de intervalos* combinadas com *AND*.
- ***Index Merge Union/Sort-Union***: Utilizados quando uma cláusula *WHERE* foi convertida para várias *condições de intervalos* combinadas com *OR*.

4.3. *Joins*

As operações de *junção* em MySQL são executadas utilizando um algoritmo de *nested-loop* (ou a variação *block nested-loop*).
A execução do *nested-loop* é a conhecida implementação de ciclos dentro de ciclos.
Por exemplo, numa base de dados com tabelas *t1*, *t2* e *t3*, um join entre elas gerava um *nested-loop* como o seguinte:

```
for each row in t1
  for each row in t2
    for each row in t3
      if row satisfies join conditions: send to client
```

A variante *block nested-loop* é similar, mas faz caching the alguns valores num *buffer* dedicado denominado *join_buffer*, cujo valor pode ser configurado através da variável de sistema *join_buffer_size*, de forma a produzir resultados mais rapidamente.

4.3.1. Otimização do *LEFT JOIN* e *RIGHT JOIN*

Vamos analisar o processo de otimização do *LEFT JOIN*. A implementação do *RIGHT JOIN* é análoga à do *LEFT JOIN*, mas com os papéis de cada tabela invertidos.
Analisemos o seguinte *query* genérico:

```
A LEFT JOIN B join_condition
```

É processado da seguinte forma:

1. A tabela *B* é dependente da tabela *A*, e de todas as tabelas de que *A* depende.
2. A tabela *A* é dependente de todas as tabelas envolvidas no *JOIN*, excepto *B*.
3. A condição *join_condition* é utilizada para decidir que registos devem ser seleccionadas de *B* (a cláusula *WHERE* é, para já, ignorada).
4. São efetuadas todas as otimizações de junções habituais, com a condição extra de uma tabela só poder ser lida após todas as tabelas das quais esta depende terem sido lidas.
5. São efetuadas todas as otimizações sob a cláusula *WHERE* (como discutido anteriormente).
6. Se existe um registo em *A* que satisfaz a cláusula *WHERE*, mas não existe um registo em *B* que satisfaça a condição *ON*, é gerado um registo extra em *B* com todas as colunas inicializadas a *NULL*.

4.4. Otimização de Operações de Ordenação (*ORDER BY*)

Em alguns casos o MySQL pode utilizar um *índice* para processar uma cláusula *ORDER BY*, sem qualquer ordenação extra. Mesmo num caso em que a cláusula não contém todo o índice, é ainda possível utilizar o índice desde que todas as partes do índice inutilizadas sejam constantes na cláusula *WHERE*. Exemplos de *queries* com cláusula *ORDER BY* que utilizam índices são:

```
SELECT * FROM t1
  ORDER BY key_part1,key_part2,... ;

SELECT * FROM t1
  WHERE key_part1 = constante
  ORDER BY key_part2;

SELECT * FROM t1
  WHERE key_part1 > constante
  ORDER BY key_part1 ASC;
```

Casos em que não é possível utilizar índices para este efeito são:

- Usa-se o *ORDER BY* em chaves diferentes.
- Mistura-se ordenação ascendente (*ASC*) com descendente (*DESC*).
- A chave utilizada para obter registos na cláusula *WHERE* não é a mesma que a chave utilizada na cláusula *ORDER BY*.

Quando não é possível utilizar um índice, é utilizado um algoritmo de *filesort*. O MySQL dispõe de dois algoritmos de *filesort*, onde um deles (o original) apenas utiliza as colunas presentes na cláusula *ORDER BY*, e o outro (a versão modificada) utiliza todas as colunas envolvidas no *query*. O otimizador utiliza, normalmente, a versão modificada do algoritmo, com a exceção de quando estão envolvidas colunas de tipo *BLOB* ou *TEXT*, altura em que utiliza o algoritmo original.
É possível configurar o tamanho do *buffer* utilizado pelos algoritmos através da variável de sistema *sort_buffer_size*. É recomendada, quando possível, a definição de um *buffer* grande, pois o tamanho deste tem um impacto significativo no tempo de execução das *operações de ordenação*.

4.5. Otimização de Operações de Agregação (*GROUP BY*)

A forma mais simples e comum de satisfazer uma cláusula *GROUP BY* é fazer uma pesquisa em toda a tabela e criar uma tabela temporária onde todos os registos de cada grupo são consecutivos, e depois utilizar esta tabela temporária para descobrir esses grupos e aplicar funções de agregação sobre estes.
O MySQL é capaz, em alguns casos, de fazer muito melhor que isso, até evitando a criação de tabelas temporários utilizando o acesso por índice.
Existem duas formas de executar um *GROUP BY* através de acesso a índices: *Loose Index Scan* e *Tight Index Scan*.

4.5.1. Loose Index Scan

O método mais eficiente de pesquisa para processar uma operação de agregação é obter diretamente as colunas de agrupamento. Com este algoritmo, o MySQL utiliza índices com chaves ordenadas (como as Árvores-B+) para encontrar as colunas necessárias sem que tenha de percorrer todo o índice. Este método não pode ser utilizado com um *GROUP BY* quando uma das seguintes condições se verifica:

- São utilizadas outras funções de agregação para além de *MIN()* ou *MAX()*.
- Quando as colunas da cláusula *GROUP BY* não estão pela mesma ordem que no índice (mais à esquerda).

- O *query* utiliza uma parte da chave numa fase que é processada depois do *GROUP BY*, e para a qual não há qualquer condição de igualdade com uma constante.

O algoritmo pode, no entanto, ser utilizado com outras funções de agregação quando não está definida uma cláusula *GROUP BY* e *DISTINCT*.

4.5.2. Tight Index Scan

Este algoritmo é utilizado como alternativa à *Loose Index Scan*, quando não é possível utilizá-la. Esta técnica de pesquisa pode ser uma *full index scan* ou uma *range index scan*. Se houverem condições na cláusula *WHERE* que operam sobre intervalos, este método executa uma *range index scan*, e portanto apenas avalia uma fração dos conteúdos do índice, caso contrário executa um *full index scan*.

4.6. Otimização de *Sub-queries*

O MySQL suporta otimização de *sub-queries* por *materialização*. São geradas tabelas intermédias, ou temporárias, por cada *query* intermédio. Estas tabelas são mantidas em memória sempre que possível, sendo apenas escritas para memória persistente quando não houver espaço suficiente em memória de acesso rápido.
Estas tabelas são indexadas com índices *hash unique*, de forma a tornar os acessos rápidos e eficientes, e não permitir valores duplicados, com o objetivo de gerar tabelas de tamanho reduzido. Para que esta otimização seja usada, a flag *materialization* da variável de sistema *optimizer_switch* deve ser ativada:

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

4.7. Estimativas de Custo de Perguntas

As estimativas de custo podem, na maioria dos casos, ser obtidas através da contagem de *disk seeks*. Para tabelas de tamanho reduzido, normalmente é possível obter o *tuplo* pretendido em apenas um *disk seek*, pois é provável que o índice esteja em *cache*.
Nos casos em que a tabela é grande, com a pesquisa feita em índices implementados sob Árvores-B+, é possível estimar com a seguinte fórmula o número de *disk seeks* necessários para encontrar os *tuplos* pretendidos:

$$\frac{log(\text{número de tuplos})}{log(\frac{\text{tamanho de um bloco do índice}}{3} \times \frac{2}{\text{tamanho de um valor chave + tamanha do ponteiro de dados}})} + 1$$

Por exemplo, um bloco de índice habitual em MySQL tem 1024 bytes de tamanho, e um ponteiro de dados tem um tamanho habitual de 4 bytes. Se uma tabela tiver 500.000 *tuplos*, e cada valor chave tem 3 bytes de tamanho, a fórmula indica:

$$\frac{log(500.000)}{log(\frac{1024}{3} \times \frac{2}{3+4})} + 1 = 4$$

Ou seja, serão necessários aproximadamente 4 *disk seeks* para obter os tuplos pretendidos.

4.8. Comparação com o Oracle 11g

O Oracle é mais completo que o MySQL, pois apresenta uma maior variedade de opções no que toca à otimização de *queries*, pois permite o uso de *pipelining* e execução paralela de perguntas, para além da *Materialização* que o MySQL já fornece. O *pipelining* e a execução paralela de perguntas torna o processamento de *sub-queries* mais eficiente que no MySQL, pois permite a execução e cálculo concorrente dos resultados dos *queries*, resultando num processamento mais rápido.
O processamento de expressões menos complexas do Oracle 11g é muito similar ao método utilizado pelo MySQL, pelo que o Oracle não tem uma vantagem discutível sob o MySQL.

5. Gestão de transações e controlo de concorrência

Neste capítulo iremos analisar a definição de transações, assim como alguns dos métodos utilizados pelo MySQL para garantir as propriedades *ACID* (Atomicidade, Consistência, Isolamento e Durabilidade).

5.1. Definição de transações

No InnoDB, todas as ações dos utilizadores ocorrem dentro de transações. Se o modo *autocommit* estiver ativo, cada expressão SQL forma uma transação por si só.
Por defeito, o MySQL começa a sessão de cada conexão com o *autocommit* ativado. Assim, o MySQL faz *commit* no fim de cada expressão SQL, a não ser que a expressão tenha retornado um erro.

Numa sessão com *autocommit* ativado, podemos definir transações de múltiplas expressões da seguinte maneira:

- ***start transaction***: começa uma nova transação.
- ***commit***: faz commit da transação corrente, tornando as mudanças permanentes.
- ***rollback***: faz roll back da transação corrente, cancelando as mudanças.

Exemplo:

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

Quando fazemos *start transaction*, o *autocommit* permanece desativado até terminarmos a transação com *commit* ou *rollback*.

Podemos desativar o modo autocommit explicitamente usando a expressão:

```
SET autocommit=0;
```

Quando desativamos o modo *autocommit*, a sessão tem sempre uma transação “aberta”. É necessário fazer *commit* ou *rollback* para que a transação corrente termine e uma nova transação comece.

5.2. Atomicidade

A *atomicidade* em bases de dados é uma propriedade que garante que, numa transação, ou são efetuadas todas as modificações, ou não é efetuada nenhuma.
Esta propriedade é mantida utilizando um mecanismo de *ROLLBACK* do sistema.
Quando uma transação é abortada, devido a um erro ou falha no sistema, ou se é chamado um *ROLLBACK* explicitamente na transação, o InnoDB executa o *rollback* assim que possível utilizando um *log* denominado *redo log*, onde estão registadas as operações executadas.
O *rollback* anula todas as instruções efetuadas pertinentes à transação que falhou, de forma a recuperar um estado da base de dados *consistente*.

5.2.1. *Savepoints*

O MySQL fornece um mecanismo de *savepoints*, que permite o controlo explícito da transação por parte do utilizador. Funciona como um sistema tradicional de *savepoints*, onde o utilizador pode definir ao longo de uma transação *savepoints* identificados, e que permite em qualquer momento fazer um *rollback* parcial da transação até ao *savepoint* indicado.
É declarado um *savepoint* com o comando:

```
SAVEPOINT identifier
```

E efetuado um *rollback* até ele usando o comando:

```
ROLLBACK TO SAVEPOINT identifier
```

É ainda possível remover um *savepoint* previamente definido (sem que haja qualquer *rollback*), utilizando o comando:

```
RELEASE SAVEPOINT identifier
```

5.3. Consistência

A *consistência* em bases de dados é a propriedade que garante que, após a execução de uma sequência de operações, a base de dados se mantém consistente. Esta propriedade está relacionada com a *atomicidade* e o *isolamento*.

Um dos mecanismos utilizados para manter a tabela consistente é a definição e verificação de *integrity constraints*. O MySQL verifica estas *constraints* de forma a garantir a *consistência* da base de dados sempre que são efetuadas alterações sob a base de dados.
É possível definir:

- *Constraints de integridade referencial*: A integridade referencial garante que uma base de dados se mantém consistente no que toca a tabelas que dependem dos valores de outras, isto é, garante a integridade das *foreign keys*, ou *chaves estrangeiras*.
    - São definidas durante a criação da tabela, ou através da operação:

```
ALTER TABLE table
ADD FOREIGN KEY (table_field) REFERENCES other_table(field);
```

- *Constraints de verificação de condições*: Estas *constraints* definidas pelo utilizador verificam condições necessárias à consistência da base de dados.
    - São definidas durante a criação das tabelas, ou através da operação:

```
ALTER TABLE table
ADD CHECK predicate
```

5.3.1. Verificação de Consistência de Tabelas

O MySQL fornece um comando que, uma vez executado, verifica se a base de dados se encontra num estado consistente analisando se *constraints* são cumpridas, problemas de referências de tabelas, etc.
O comando *CHECK TABLE* tem a seguinte sintaxe:

```
CHECK TABLE tbl_name [, tbl_name] ... [option] ...
option = {FOR UPGRADE | QUICK | FAST | MEDIUM | EXTENDED | CHANGED
```

O parâmetro *option* permite parametrizar o quão extensa/detalhada deve ser a verificação feita à tabela.

5.4. Isolamento

O *isolamento* em bases de dados é a propriedade que garante que a execução de transações concorrentes são independentes, e que não interferem umas com as outras, garantindo *consistência*.
O protocolo de isolamento utilizado pelo MySQL suporta *Multi-Versioning* (para *leituras consistentes*) e mecanismos de *locks* (para que as modificações à base de dados sejam *atómicas*). São suportados vários níveis de *isolamento* e *granularidade*.

5.4.1. Níveis de Isolamento

Em ambientes concorrentes, é necessário garantir o *isolamento* de transações concorrentes, no entanto é também desejável que as transações possam aceder a dados de forma *concorrente* e *consistente* para otimizar o uso dos recursos do sistema. O MySQL oferece diversos *níveis de isolamento*, que isolam as transações de formas diferentes, e com garantias diferentes.
A configuração do *nível de isolamento* utilizado é feita através do comando:

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

Com a opção *global*, o efeito da alteração é persistente para todas as sessões futuras efetuadas na base de dados, enquanto que a opção *session* apenas altera o *nível de isolamento* na sessão que executou o comando. É também especificado o *nível de isolamento* desejado. Por defeito é utilizado o nível *repeatable read*.
Os *níveis de isolamento* fornecidos pelo MySQL são: *repeatable read*, *read committed*, *read uncommited* e *serializable*, níveis estes que têm uma implementação idêntica no MySQL à estudada nas aulas.

5.4.2. Row-Level Locking

Um dos mecanismos de *locking* que o InnoDB implementa é o *row-level locking*.
Esta estratégia é mais eficiente que um *table lock* tradicional, que bloqueia toda a tabela, em vez de bloquear apenas os tuplos que serão utilizados pelo *query*. Este mecanismo funciona como o *row-level locking* estudado nas aulas, isto é, existem dois tipos de *locks*:

- *shared lock*: Esta *lock* atribui a uma transação direitos de leitura sob um *tuplo*. Só é atribuída quando nenhuma outra transação detém uma *exclusive lock*.
- *exclusive lock*: Esta *lock* atribui a uma transação direitos de leitura e escrita/remoção sob um tuplo. Só é atribuída quando nenhuma outra transação detém uma *shared lock* ou *exclusive lock*.

	shared lock	*exclusive lock*
shared lock	true	false
exclusive lock	false	false

Matriz de lock-compatibility para row-level locking

Existem ainda outros tipos de *record-level locks*, relacionados com este tipo de *locks*.

- ***Record Lock***: Bloqueia um *tuplo de índice*. Se uma tabela não tiver índices, o InnoDB gera um *clustered index* invisível, e faz *lock* a esse índice.
- ***Gap Lock***: Bloqueia um intervalo de *tuplos de índices*. Utilizado para sincronizar inserções em intervalos similares.
- ***Next-Key Lock***: Este mecanismo de *locking* é uma combinação dos dois anteriores. Previne a existência de *phantom rows*.

5.4.3. Table Locking

É possível, apesar de não recomendável, usar uma *lock* sob uma tabela inteira explicitamente através do comando:

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

Para remover a *lock* sobre a tabela utiliza-se:

```
UNLOCK TABLES
```

5.4.4. Multiple Granularity Locking

Para além das *locks* apresentadas anteriormente, o InnoDB suporta ainda *multiple granularity locking*, ou *locking por níveis de granularidade*. Este mecanismo permite a co-existência de *locks* sob *tuplos* (*row-level locking*) e *locks* sob tabelas (*table locks*).
Para o efeito, são introduzidas novas *locks*, denominadas *intention locks*. Estas *locks* servem para indicar que tipo de *lock* uma transação virá a precisar sob um *tuplo*, algures na transação.
Estas *intention locks* são:

- *intention shared*: A transação pretende pedir uma *shared lock* sob tuplos individuais de uma tabela.
- *intention exclusive*: A transação pretende pedir uma *exclusive lock* sob tuplos individuais de uma tabela.

As *intention locks* devem ser adquiridas antes de se poder adquirir uma *lock* concreta.
Por exemplo, um *query* da forma:

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

Deverá obter uma *intention shared lock*, antes de poder obter a *shared lock*, que lhe dará o direito de ler os tuplos desejados. A matriz de *lock compatibility* seguinte representa a compatibilidade das *locks* envolvidas nestes *níveis de granularidade*.

	exclusive	*int. exclusive*	*shared*	*int. shared*
exclusive	false	false	false	false

int. exclusive	false	true	false	true
shared	false	false	true	true
int. shared	false	true	true	true

Matriz de lock-compatibility para multiple granularity

5.4.5. Deadlocks

O InnoDB deteta *deadlocks* em transações automaticamente, e faz *rollback* das transações necessárias à quebra do *deadlock*. O protocolo tenta escolher para *rollback* transações com poucas operações de modificação à base de dados, de forma a reduzir o tempo e custo de execução do *rollback*.
Pode ser analisada a causa do último *deadlock* através do comando:

```
SHOW ENGINE INNODB STATUS
```

O InnoDB não é capaz de detetar *deadlocks* quando *autocommit=1*, o que leva ao sistema funcionar com um mecanismo de prevenção de *deadlocks por timeout*. Isto quer dizer que uma transação que fique bloqueada por mais do que um determinado tempo definido é *rolled back*. Este valor pode ser configurado através da variável de sistema *innodb_lock_wait_timeout*.

5.5. Durabilidade

A *durabilidade* em bases de dados é a propriedade que garante que as modificações efetuada por uma transação *committed* são persistentes, ou permanentes.

5.5.1. DoubleWrite Buffer

O InnoDB utiliza uma técnica de *file flushing* chamada *Doublewrite*. Antes de os dados serem escritos para um ficheiro de dados, o InnoDB escreve-os primeiro para uma zona contígua de memória denominada *Doublewrite buffer*. Só após uma escrita completa dos dados for feita para este *buffer*, é que os dados são escritos para os ficheiros de dados. Se houver um *crash de*

sistema a meio de uma escrita, é sempre possível obter uma cópia saudável dos dados a partir dos dados ainda mantidos no *doublewrite buffer*.
É possível desativar este mecanismo através da variável de sistem *innodb_doublewrite=0*.

5.6. Comparação com o Oracle 11g

O Oracle 11g implementa as transações de forma muito parecida ao MySQL.
São *ACID-compliant*, pois respeitam as propriedades de atomicidade, consistência, isolamento e durabilidade. Fazem deteção automática de *deadlocks*, fornecem os mesmo *níveis de isolamento*, têm mecanismos de *locks* muito parecidos (*row-level locking, table-level locking, multiple granularity locking*), permitem a definição de *savepoints* e permitem a definição de *integrity constraints*.
É de notar, no entanto, que ao contrário do MySQL, o Oracle 11g tem suporte a *nested transactions*.

6. Suporte para bases de dados distribuídas

O MySQL suporta bases de dados distribuídas, sendo que a melhor tecnologia que fornece para o efeito é o *MySQL Cluster*. É importante notar que esta funcionalidade do MySQL não utiliza o *storage engine InnoDB*, mas sim um *engine* próprio denominado *NBD* (***N**etwork **D**ata**B**ase*) que funciona em memória.
Foi desenvolvido com as propriedades *ACID* em mente, assim como a escalabilidade e uma disponibilidade elevada (99.9%).
É desenhado para não ter um *single point of failure*, pois é implementado sob uma arquitetura *shared-nothing*, algo que também permite que o sistema funcione com hardware barato.

6.1. Arquitetura do MySQL Cluster

O MySQL Cluster é organizado por *nodes*. Um *node* é um processo a correr num computador, ou *cluster host*. Um *host* pode executar vários *nodes*, mas é ideal uma configuração onde vários *nodes* são distribuídos por diversos *hosts*.
Existem três tipos de *nodes* num *cluster*, e é condição necessária que haja pelo menos um de cada um destes *nodes* para criar um *cluster*. Estes são:

- ***Management Node*:** A função deste *node* é gerir os outros nós do MySQL Cluster, executando funções como fornecer dados de configuração, iniciar e para *nodes*, efetuar *backups*, etc. Como este nó gere ficheiros de configuração de outros *nodes*, deve ser inicializado primeiro um destes nós antes de qualquer outro.
- ***Data Node*:** A função deste *node* é guardar *cluster data*, isto é, dados mantidos em *cluster*. Existem tantos *nodes* destes quanto réplicas, para fornecer redundância e tolerância de perda de dados.
- ***SQL Node*:** A função deste *node* é aceder à *cluster data*. É um servidor de SQL normal.

O MySQL Cluster é desenhado com a intenção de os *data nodes* serem homogéneos.

6.2. Replicação e Fragmentação de Dados

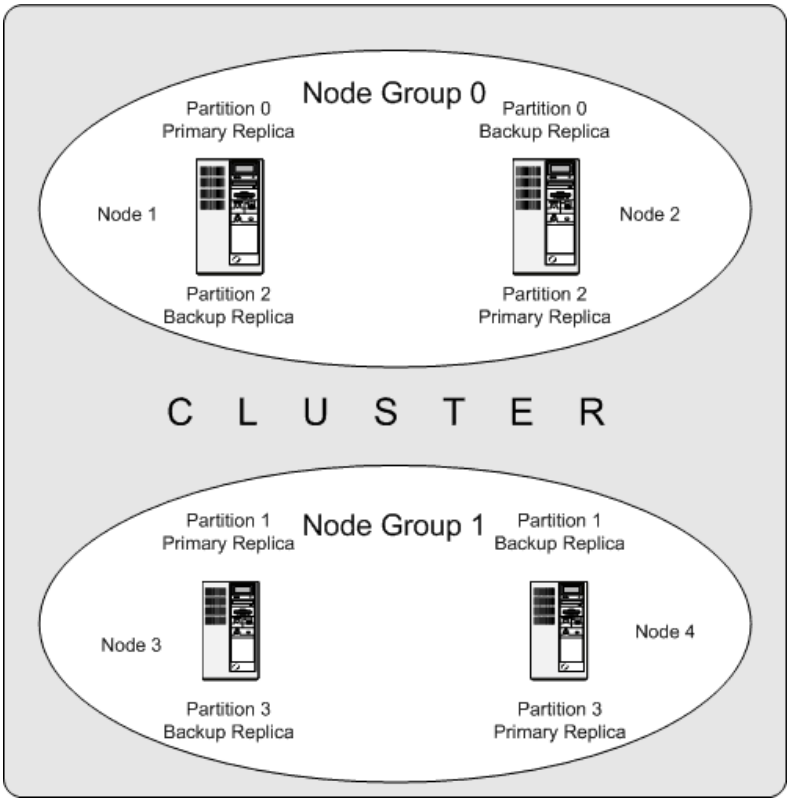
O MySQL Cluster é limitado a um *data node* por *host*. Isto significa que a replicação e fragmentação de dados não pode ser feita num único *host*, mas sim num conjunto deles.
Para este efeito foram criados *node groups*, que são conjuntos de um ou mais *data nodes* que guardam partições ou conjuntos de réplicas relacionadas entre si.

O número de *node groups* num MySQL Cluster não é diretamente configurável, mas sim o produto da seguinte fórmula:

```
[número_de_grou_nodes] = número_de_data_nodes / NumeroDeRéplicas
```

Portanto, um MySQL Cluster com 4 *data nodes,* tem 2 *node groups* se *NumeroDeRéplicas=2*, cenário onde há redundância de dados.
É condição necessária que todos os *node groups* de um MySQL Cluster tenham o mesmo número de *data nodes*.
Uma *partition*, ou *partição*, é uma porção de todos os dados guardados no *cluster*. Existem tantas partições como *nodes* num *cluster*, pois cada *data node* é responsável por manter pelo menos uma cópia de todas as partições pela qual é responsável.
Uma *replica*, ou *réplica*, é uma cópia de uma partição. Cada *node* de um *node group* guarda uma réplica. O número de réplicas é igual ao número de *nodes* por *node group*.

O seguinte diagrama ilustra um MySQL Cluster com 4 *data nodes*, organizados em 2 *node groups* (portanto, *NumeroDeRéplicas=2*). Os *nodes* 1 e 2 pertencem ao *node group* 0, e os *nodes* 3 e 4 pertencem ao *node group 1*. Apenas são apresentados *data nodes*.



(Documentação do MySQL Cluster: http://dev.mysql.com/doc/refman/5.6/en/mysql-cluster-nodes-groups.html)

É possível observar na figura que:

- A *partition 0* é uma *primary replica* (cópia principal de uma partição), e que uma *backup replica* (cópia de backup da partição) é mantida no *node 2*.
- A *partition 1* é guardada no outro *node group* (grupo 1), e que a sua *primary replica* está guardada no *data node 3*, e a *backup replica* correspondente no *data node 4*.
- As partições *partition 2* e *partition 3* são guardadas de forma similar, mas com as funções de cada *data node* invertidas.

Para garantir que as réplicas são consistentes, as escritas às partições são feitas usando um protocolo de *two phase commit*. O *tradeoff* deste protocolo é o seguinte: um número elevado de réplicas produz um sistema com maior tolerância a falhas, no entanto produz também um maior número de escritas necessárias pois será necessário que todas as réplicas representem a mesma informação. Garante também atomicidade em transações globais pelo uso deste algoritmo.

6.3. Transparência em Queries Distribuídos

Os acessos aos dados são todos feitos através dos *SQL Nodes*, que têm a função de interpretar as transações e comunicá-las aos *management nodes*. Os *Management Nodes* têm então a função de distribuir os pedidos pelos *node groups* relevantes.
O cliente não sabe com que réplicas ou partições está a interagir, nem como, pois nenhum desses *nodes* são visíveis a ele, ou a qualquer interação exterior ao MySQL Cluster.
É então garantida a transparência de fragmentação, replicação e localização.

6.4. Algumas notas sobre o MySQL Cluster

A documentação fornecida pela Oracle pertinente ao MySQL Cluster é muito incompleta, portanto foi-nos difícil determinar alguns aspetos relevantes ao estudo desta tecnologia.
No entanto, gostariamos de realçar, de forma sumária, alguma informação que consideramos relevante, mas que não conseguimos explorar com o detalhe desejado.

O MySQL Cluster não implementa locks distribuídos
Num sistema de bases de dados distribuídos é comum a dificuldade em implementar *locks distribuídos*. É fácil ver porque é que a existência de tais *locks* seria uma mais valia enorme para o sistema, pois os *node groups* são perfeitos para acesso paralelo a dados de um *node group*, no

entanto seria necessário um protocolo de *exclusão mútua* para que isso fosse implementado de forma fiável.

O MySQL Cluster não suporta *foreign keys*, *savepoints* nem *savepoint rollbacks*
Está a ser testada, no entanto, a implementação de *foreign keys* para versões futuras do MySQL Cluster.

O MySQL Cluster só fornece um nível de isolamento
O *NDB* apenas fornece o nível *read_committed*. Para além disso, não suporta *rollbacks parciais*, sendo que num caso em que uma transação falha é imediatamente feito um *rollback total*, e a transação terá de ser executada totalmente.

7. Bibliografia

Documentação de referência Oficial do MySQL, versão 5.7.
http://dev.mysql.com/doc/refman/5.7/en/

Artigo na Wikipédia onde é descrito o MySQL Cluster:
http://en.wikipedia.org/wiki/MySQL_Cluster

Artigo na Wikipédia sobre o MySQL:
http://en.wikipedia.org/wiki/MySQL

Database System Concepts - Sixth Edition, por Abraham Silberschatz, Henry F. Korth, e S. Sudarshan.