Sistemas de Bases de Dados 13/14

Análise do Sistema
MySQL 5.7.4 (InnoDB)

Docente:
José Alferes

Autores:
Bruno Palma 42003
Bruno Farto 41854
Pedro Fernandes 42359

Grupo 29

Index

1 Resumo 3
2 Armazenamento e Estrutura de Ficheiros 4
2.1 Buffer Pool 4
2.1.1 Configuração da buffer pool: 5
2.2 Tablespace 6
2.2.1 Tipos de ficheiro suportados: 8
2.3 Modo File-Per-Table 8
2.4 Clustered Index 9
2.4.1 Particularidades do Clustered Index 9
2.5 Partições 9
2.6 Comparações com o Oracle 13
3 Indexação e Hashing 14
3.1 Modo de Funcionamento 14
3.2 Tipos de Índices Suportados 15
3.2.1 B-tree 16
3.2.2 Hashing 17
3.3 Comandos Para Índices 18
3.4 Comparação com o Oracle 19
4 Processamento e Otimização de Perguntas 20
4.1 Processamento e otimização de perguntas 20
4.1.1 Explain 21
4.1.2 Analyze 22
4.1.3 Optimização da cláusula where 22
4.1.4 Optimização da cláusula join 23
4.1.5 Nested-loop join 24
4.1.6 Block nested-loop join 24
4.1.7 Optimização da cláusula order by 25
4.1.8 Optimização da cláusula group by 26
4.1.9 Loose index scan 26
4.1.10 Tight index scan 28

4.2 Comparação com o Oracle 30
5 Gestão de transações e controlo de concorrência 31
5.1 Propriedades ACID 31
5.1.1 Atomicidade (A) 31
5.1.2 Consistência (C) 32
5.1.3 Isolamento (I) 33
5.1.4 Durabilidade (D) 36
5.2 Comparação com o Oracle 38
6 Suporte Para Bases De Dados Distribuídas 39
6.1 Homogéneas 39
6.1.1 Replicação 39
6.2 Heterogéneas 41
6.2.1 Fragmentação 41
6.3 Transações distribuídas (XA) 41
6.4 Outras ferramentas (motores) 43
6.4.1 MySQL Cluster (NBD) 43
6.4.2 Federated 43
6.5 Comparação com o SGBD da Oracle 44
7 Outras características do MySQL 45
7.1 Suporte para a WEB 45
7.1.2 Suporte para xPath 48
7.1.3 Suporte para Semantic Web (RDF/Schema e SPARQL) 49
7.2 Stored Procedures & Functions 50
7.3 Trigger 51
7.4 Conectores e APIs 52
7.5 MySQL Access Privilege System (APS) 52
7.6 Tipos de dados 53
7.7 Ferramentas Workbench 54
8 Conclusão 56
9 Referencias 57

1 Resumo

O principal objetivo deste relatório é analisar um sistema de base de dados que não o Oracle, tendo sido escolhido o sistema de base de dados relacional MySQL 5.7.4 com o engine InnoDB. A escolha recaiu neste SBD devido ao facto de ser o 2º mais usado mundialmente atrás do Oracle e à frente do Microsoft SQL Server, sendo o 1º tendo em conta apenas os open source. A selecção do engine recaiu no InnoDB devido ao facto de ser o engine que o MySQL usa por defeito e também o que engloba uma maior parte da matéria da disciplina de SBD.

2 Armazenamento e Estrutura de Ficheiros

2.1 Buffer Pool

O InnoDb mantém uma área de armazenamento denominada de *buffer pool* para efetuar caching de dados e índices em memória. Esta estrutura é gerida como uma lista recorrendo a uma variante do algoritmo *least recently used* (LRU). Quando é necessário espaço para adicionar um novo bloco à pool, o InnoDB elimina o bloco menos usado recentemente e adiciona o novo bloco para o meio da lista. Esta estratégia de inserção no ponto médio trata a lista como duas sub-listas:

Na cabeça, uma sublista de blocos “recentes” que foram acedidos recentemente.

Na cauda, uma sublista de blocos "antigos" que foram acedidos menos recentemente.

Este algoritmo mantém blocos que são frequentemente usados em *queries* na nova sublista. A sublista dos blocos antigos contém blocos menos utilizados, blocos que são candidatos a serem eliminados.

O algoritmo LRU funciona da seguinte forma:

3/8 da área do buffer são dedicado à sublista para os blocos antigos.

O ponto médio da lista é a fronteira onde a cauda da sublista dos blocos recentes se intersecta com a cabeça da sublista de blocos antigos.

Quando o InnoDB lê um bloco na área de buffer, este é inicialmente inserido no ponto médio, ou seja, a cabeça da sublista dos blocos antigos. Um bloco pode ser lido no sistema porque ele é necessário para a operação especificada pelo usuário como uma consulta em SQL ou como parte de uma operação de leitura antecipada realizada automaticamente pelo InnoDB.

Acedendo um bloco na sublista de blocos antigos torna-o recente movendo-o para a cabeça da área de buffer, ou seja, a cabeça da sublista dos blocos recentes. Se o bloco foi lido por necessidade, o primeiro acesso ocorre imediatamente e o bloco torna-se recente. Se o bloco foi lido devido ao read-ahead, o primeiro acesso não ocorre imediatamente (e pode não ocorrer antes do bloco ser removido).

À medida que a base de dados funciona, blocos na buffer pool que não são acedidos "envelhecem" sendo deslocados para a cauda da lista. Blocos nas sublistas dos blocos antigos e recentes envelhecem enquanto outros blocos se tornam recentes. Blocos na sublista de blocos antigos também envelhecem

à medida que se inserem blocos no ponto médio. Eventualmente, um bloco que permanece não usado durante algum tempo é deslocado a cauda da sub-lista de blocos antigos sendo depois expulso.

Por defeito, os blocos que são lidos por *queries* são movidos imediatamente para a nova sub-lista significando que vão permanecer na buffer pool durante um longo período de tempo. Um scan numa tabela (como a que é realizada por uma declaração SELECT sem a cláusula WHERE) pode trazer uma grande quantidade de dados para a buffer pool e despejar uma quantidade equivalente de dados mais antigos, mesmo que os dados recentes nunca mais voltem a ser usados. Da mesma forma, os blocos que são carregados pelo *read-ahead background thread* e depois acedidos apenas uma vez passa para a cabeça da lista de blocos recentes. Estas situações podem empurrar blocos frequentemente usados para a sub-lista de blocos antigos onde eles se tornam sujeitos a serem expulsos.

2.1.1 Configuração da buffer pool:

Existem diversas variáveis do sistema InnoDB que permitem alterar o tamanho da buffer pool e afinar o algoritmo LRU:

innodb_buffer_pool_size

Especifica o tamanho da buffer pool. Se a buffer pool for de pequenas dimensões e a memória for suficiente, aumentar o tamanho da pool pode aumentar o desempenho ao reduzir o número de acessos de disco I/O necessários á medida que as *queries* acedem tabelas InnoDB.

innodb_buffer_pool_instances

Divide a área de buffer num número de regiões distintas especificado pelo utilizador, cada uma com a sua própria lista LRU e estruturas de dados relacionadas de forma a diminuir a contenção durante operações de leitura e escrita em memória concorrente. Esta opção tem efeito apenas quando se define o innodb_buffer_pool_size para um tamanho de 1 gigabyte ou superior. O tamanho total que for especificado é dividido entre todas as buffer pools. Para uma melhor eficiência, especifica-se uma combinação entre innodb_buffer_pool_instances e innodb_buffer_pool_size de forma a que cada instância buffer pool tenha pelo menos 1 gigabyte.

innodb_old_blocks_pct

Especifica a percentagem aproximada da buffer pool que o InnoDB usa para a sub-lista de blocos antigos. O intervalo de valores está situado entre os 5 e 95. O valor por defeito é 37, ou seja, 3/8 da pool.

innodb_old_blocks_time

Especifica quanto tempo em milisegundos (ms) é que um bloco inserido terá que permanecer na sublista de blocos antigos depois do primeiro acesso antes que possa ser deslocado para a sublista de blocos recentes. O valor definido por defeito é 0: um bloco inserido na sublista de blocos antigos é movido para a sublista de blocos recentes quando o InnoDB expulsou ¼ das páginas dos blocos inseridos da buffer pool independentemente do quão cedo ocorra o acesso depois da inserção. Se o valor for maior que 0, os blocos permanecem na lista de blocos antigos até que ocorra um acesso pelo menos uns quantos milissegundos depois do primeiro acesso. Por exemplo, um valor de 1000 faz com que os blocos permaneçam na sublista de blocos antigos 1 segundo depois do primeiro acesso antes que se tornem elegíveis a serem deslocados para a sublista de blocos recentes.

Para podermos visualizar informações relativamente a estatísticas da *buffer pool*, corremos o seguinte comando:

```
SHOW ENGINE INNODB STATUS
```

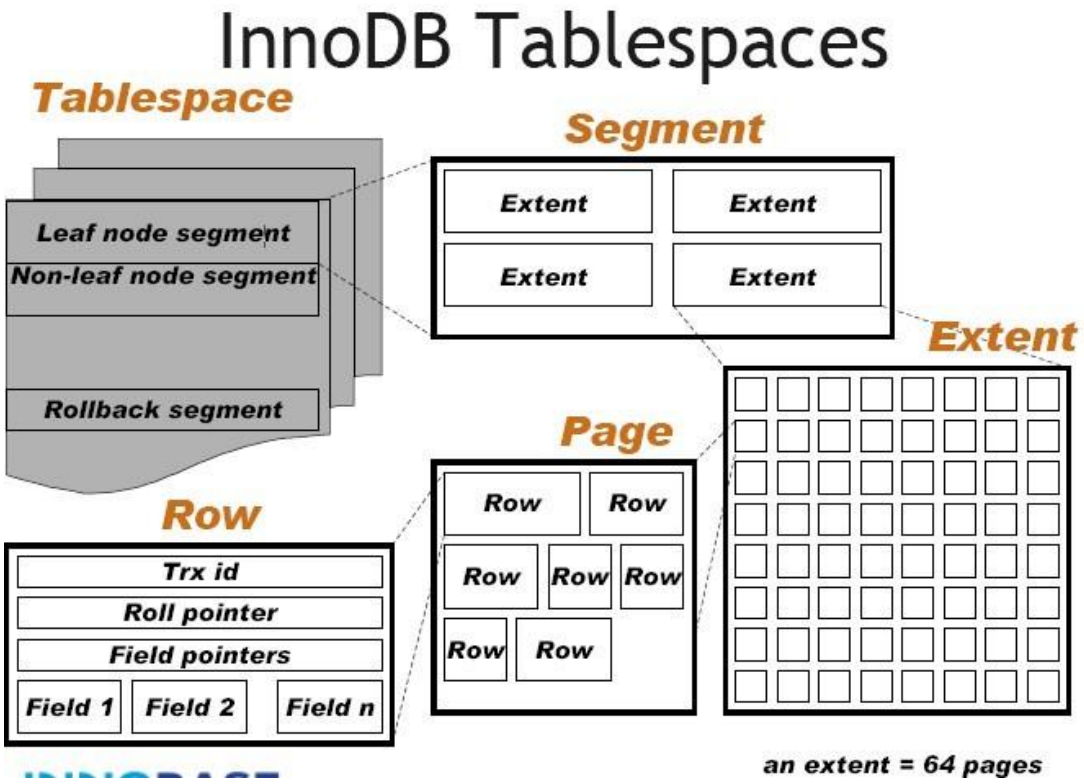
2.2 Tablespace

Um ficheiro de dados consegue manter dados para uma ou mais tabelas InnoDB assim como os índices que estão associados. A *tablespace* deste sistema contém tabelas que formam o dicionário de dados (*data dictionary*) e mantém todas as outras tabelas InnoDB por defeito.

Uma *tablespace* pode ser constituída por múltiplos ficheiros ao nível de sistema operativo (i.e. ibdata1, ibdata2, etc.) mas é na realidade um ficheiro lógico único, ou seja, múltiplos ficheiros físicos são tratados como se fossem fisicamente concatenados.

Cada *tablespace* do InnoDB contém um valor inteiro de 32 bits associado a este conhecido como *space ID* usado para localizar essa *tablespace*. O InnoDB contém um espaço de sistema por defeito com um *space ID* associado que contém o valor 0. Este espaço é usado para efectuar operações especiais agendamento que este motor de armazenamento necessita. Internamente, este ficheiro .ibd é na realidade um espaço funcional que pode conter várias tabelas mas na implementação com o MySQL, apenas irá conter uma tabela única.

Uma *tablespace* é constituída por várias páginas cada uma com um tamanho de 16 kb. Cada página dentro de uma *tablespace* contém um valor inteiro de 32 bits associado conhecido como *offset* que é na realidade o espaço inicial da página.



Estrutura da Tablespace

2.2.1 Tipos de ficheiro suportados:

O InnoDB suporta dois tipos de ficheiro para armazenar *tablespaces*: o Barracuda e o Antelope sendo este o que se encontra configurado por defeito.

O Antelope suporta tuplos do tipo *redundant* e *compact* que permite uma representação mais compacta para nulos e campos com variáveis do tipo *length* comparativamente ao tipo *redundant*. Porém, este formato não suporta tuplos do tipo *dynamic* e *compressed* que se encontram disponíveis no formato de ficheiros Barracuda.

Como referimos anteriormente, o formato Barracuda suporta tuplos do tipo *compressed* que permite ao InnoDB comprimir tabelas e tuplos do tipo *dynamic* que melhora o esquema de armazenamento para BLOB e colunas de texto grandes. Visto que o InnoDB está configurado por defeito com o formato de ficheiros Antelope, para usar o Barracuda, também é necessário permitir a definição *file-per-table* que coloca tabelas recentemente criadas em *tablespaces* distintas da *tablespace* de sistema.

É possível seleccionar o formato do ficheiro a ser usado ao definir a opção `innodb_file_format` antes de criar uma tabela.

2.3 Modo File-Per-Table

Apesar das tabelas e índices serem armazenados numa tablespace por defeito, é possível armazenar tabelas InnoDB e índices em ficheiros separados. Esta funcionalidade é conhecida por modo *file-per-table* (ficheiro por tabela) tendo em conta que cada tabela que é criada quando esta funcionalidade se encontra activada contém o seu próprio ficheiro ".ibd". Cada um deste tipo de ficheiros representa uma tablespace separada. Este modo pode ser controlado através da opção de configuração innodb_file_per_table.

2.4 Clustered Index

Índice específico usado em todas as tabelas do InnoDB que armazena dados dos tuplos. Acedendo a um tuplo através de um índice *clustered* é rápido tendo em conta que cada tuplo de dados encontra-se na mesma página onde é feita a procura através de índice. Se uma tabela for grande, a arquitetura do índice *clustered* irá guardar uma operação I/O em disco com frequência quando comparado com organizações de armazenamento que armazenam tuplos de dados usando uma página diferente do registo do índice.

2.4.1 Particularidades do Clustered Index

Se for definida uma chave primária numa tabela, o InnoDB usa-a como um índice *clustered*.

Se não for definida uma chave primária para uma tabela, o MySQL escolhe o primeiro índice único que contém apenas colunas NOT NULL como chave primária sendo usadas como um índice *clustered*.

Se uma tabela não conter nenhuma chave primária ou um índice único adequado, o InnoDB gera um índice *clustered* escondido numa coluna sintética contendo valores *row ID*. Os tuplos são ordenados pelo ID que o InnoDB assinala aos tuplos numa dada tabela. O *row ID* é um campo de 6 bytes que aumenta monotonicamente à medida que se inserem novos tuplos, sendo que os tuplos ordenados pelo *row ID* encontram-se fisicamente na ordem de inserção.

2.5 Partições

O particionamento permite distribuir porções de tabelas individuais sobre um sistema de ficheiros de acordo com as regras que podem ser estabelecidas. Esta técnica permite melhorar o desempenho de bases de dados.

O MySQL suporta *horizontal partitioning*, permitindo que os tuplos de uma base de dados sejam divididos em conjuntos de dados pequenos distribuindo-os por múltiplas directorias e discos.

Existem diversos métodos que permitem controlar as partições sendo estes 4 os que são suportados pelo MySQL. Antes de analisarmos os métodos suportados pelo MySQL, consideremos a seguinte tabela:

RANGE partitioning - este tipo de particionamento atribui tuplos a partições baseadas em valores da coluna dentro de um determinado intervalo.

Exemplo:

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

LIST partitioning- semelhante ao particionamento por intervalo (*RANGE partitioning*) excepto que a partição é selecionada baseado em colunas que correspondem a uma dentro de um conjunto de valores discretos.

Exemplo:

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

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

HASH partitioning - seleciona a partição baseando-se no valor devolvido por uma expressão definida pelo utilizador, que opera em valores de coluna dentro de tuplos a serem inseridos na tabela. A função pode consistir de uma qualquer expressão válida no MySQL que gere um valor inteiro não-negativo. Existe uma extensão disponível conhecida como LINEAR HASH.

Exemplo:

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

KEY partitioning- semelhante ao particionamento por *HASH* excepto que apenas uma ou duas das colunas a serem avaliadas são fornecidas e o MySQL fornece a sua própria função de *hashing*. Estas colunas podem ter valores para além dos inteiros visto que a função de hashing fornecida pelo MySQL garante um resultado inteiro independentemente do tipo de dados da coluna. Existe uma extensão disponível conhecida como LINEAR KEY.

Exemplo:

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

Também é possível realizar *subpartitioning*, ou seja, particionar partições resultando em tipos de particionamento compostos.

2.6 Comparações com o Oracle

Como no InnoDB, o Oracle tem o seu próprio buffer manager no *System Global Area* com uma lista LRU cujo funcionamento é idêntico á lista do buffer pool. Porém, o Oracle possui mais componentes no SGA comparativamente à buffer pool do InnoDB.

Apesar do Oracle também criar *tablespaces*, por defeito, estes são partilhados por várias tabelas sendo-o por defeito o *system tablespace,* abordagem diferente adoptada pelo MySQL (*tablespace* por tabela). Ao contrário do InnoDB, o Oracle organiza as tabelas por defeito em heap (permitindo também organização por índice como no InnoDB) e permite *multitable clustering*.

Tal como no MySQL, o Oracle permite o particionamento de tabelas e os modos que referimos á exceção do tipo KEY.

3 Indexação e Hashing

Os índices são estruturas de dados usadas em sistemas de bases de dados que permitem melhorar o desempenho de consultas sobre tabelas o que permite uma consulta mais rápida de tuplos sem ter que percorrer uma tabela inteira.

3.1 Modo de Funcionamento

Conforme referimos anteriormente, os índices são utilizados para encontrar tuplos de uma coluna de uma forma mais rápida. Sem a utilização destes, o MySQL começa a ler da primeira linha percorrendo a tabela inteira até encontrar os tuplos desejados. Quanto maior a tabela, maior será o custo.

Se a tabela possui um índice para as colunas em questão, o MySQL pode rapidamente determinar a posição dos dados desejados sem ter de olhar para todos. Para exemplificar, se uma tabela possuir 1000 registos, a leitura vai ser 100 vezes mais rápida do que a leitura sequencial.

A maioria dos tipos de índice suportados pelo MySQL são armazenados em B-trees à excepção dos índices de tipos de dados SPATIAL que usam R-trees.

O tipo de índice mais comum envolve uma única coluna, armazenando cópias dos valores dessa coluna em uma estrutura de dados permitindo pesquisas mais rápidas para as linhas com os valores da coluna correspondentes. A estrutura de dados da B-tree permite que o índice encontre rapidamente um valor específico dentro de um conjunto de valores ou de um intervalo de valores, correspondendo aos operadores de comparação =,>, ≤, BETWEEN e IN entre outros numa cláusula WHERE.

Também é possível criar índices para mais do que uma coluna. Um índice pode conter até 16 colunas. Para certos tipos de dados, é possível indexar um prefixo da coluna. Índices com múltiplas colunas podem ser usados para consultas que testam todas as colunas no índice ou consultas em que só se testa apenas a primeira coluna, as duas primeiras colunas, as três primeiras colunas e assim adiante. Se especificarmos as colunas na ordem correcta ao definirmos o índice, um único índice composto pode acelerar vários tipos de consultas na mesma tabela.

Se um índice de colunas múltiplas existir em col1 e col2, os registros apropriados podem ser recuperados diretamente. Se índices separados para

únicas colunas existirem em col1 e col2, o optimizador tenta usar a técnica de otimização Index Merge ou tenta encontrar o índice mais restritivo decidindo qual índice exclui mais linhas e usará este índice para recuperar os registros.

Se a tabela conter um índice de múltiplas colunas, qualquer prefixo mais à esquerda do índice pode ser usado pelo mecanismo de otimização para encontrar registos. Por exemplo, se tivermos um índice composto por 3 colunas denominadas de col1, col2 e col3, é possível fazer procuras indexadas em (col1), (col1, col2) e (col1, col2, col3).

O número máximo de índices por tabelas e o comprimento máximo do índice é definido pelo mecanismo de armazenamento. Todos os motores de armazenamento suportam pelo menos 16 índices por tabela e um índice de comprimento total de pelo menos 256 bytes. A maioria dos motores de armazenamento têm limites superiores. No caso do InnoDB, uma chave de índice para um índice numa coluna única pode ter um comprimento até 767 bytes.

3.2 Tipos de Índices Suportados

Os tipos de indexação suportados pelo MySQL são B-Tree e Hashing. Porém, nem todos os motores de armazenamento suportam ambos os tipos de indexação referidos.

Storage Engine	Permissible Index Types
MyISAM	BTREE
InnoDB	BTREE
MEMORY/HEAP	HASH, BTREE
NDB	BTREE, HASH

Entraremos em detalhe com as B-Tree que são suportadas pelo motor de armazenamento InnoDB e um mecanismo usado por este conhecido como *Adaptive Index Hashing* que exploraremos em detalhe.

3.2.1 B-tree

Este tipo de índice é usado por defeito pela maioria dos motores de armazenamento no MySQL. A ideia geral é que todos os valores são armazenados em ordem, e cada página da folha é à mesma distância da raiz.

Os índices B-tree aceleram o acesso a dados visto que o seu mecanismo de armazenamento não torne necessário percorrer a tabela inteira para encontrar os dados desejados. Em vez disso, ele começa no nó raiz. Os *slots* no nó raiz armazenam *pointers* para os nós filho e o mecanismo de armazenamento segue estes *pointers*. Ele encontra o *pointer* do lado direito analisando os valores em cada nó que definem os limites superior e inferior dos valores nos nós filho. Eventualmente, o mecanismo de armazenamento determina que o valor pretendido não existe ou alcança a página de uma folha com sucesso. Páginas de folha são especiais, porque eles têm ponteiros para os dados indexados ao invés de ponteiros para outras páginas.

Queries que beneficiam deste tipo de indexação incluem correspondências com valores inteiros, o prefixo mais á esquerda, prefixo de uma coluna, um intervalo de valores e correspondência com uma parte exacta e correspondência com um intervalo de valores na outra.

Porém, existem algumas limitações neste tipo de índices. Uma B-tree não vai ajudar se a pesquisa não começa a partir das colunas mais á esquerda na tabela de índices existentes, se alguma das colunas no índice fôr ignorada e se alguma condição fôr usada depois de uma condição envolvendo margens de valores.

Um índice de B-tree pode ser usado para comparações de colunas em expressões que usam os operadores = ,>,> =, < e <= entre outros. O índice também pode ser usado para comparações do tipo LIKE se o argumento usado fôr uma string constante que não começa com um caractere *wildcard*.

Uma pesquisa em colunas utilizando IS NULL emprega índices se o col_name for indexado.

Qualquer índice que não cobre todos os níveis na cláusula WHERE não é usado para optimizar a consulta. Por outras palavras, para ser possível utilizar um índice, um prefixo deste deve ser utilizado em todo o grupo.

Às vezes o MySQL não utiliza um índice, mesmo que se encontre disponível. Pode acontecer quando o optimizador estima que o uso do índice necessita que o MySQL aceda a uma grande percentagem das linhas na tabela. Neste caso, uma varredura da tabela é provavelmente muito mais rápido porque exige menos procura. No entanto, se uma consulta utilizar LIMIT para recuperar apenas alguns dos registos, o MySQL utiliza um índice de qualquer

maneira porque pode encontrar muito mais rapidamente as poucas linhas para devolver no resultado.

3.2.2 Hashing

Apesar do InnoDB não suportar este tipo de índices, esta lacuna é colmatada com a presença de um mecanismo que monitoriza buscas indexadas feitas aos índices definidos para uma tabela sendo conhecido como *Adaptive Hash Index*. Se o InnoDB notar que as consultas poderiam beneficiar de um *hash índex*, este é gerado automaticamente.

Este tipo de índices são sempre construídos com base num índice B-tree existente numa tabela. O InnoDB pode construir um índice hash num prefixo de qualquer tamanho da chave definida pela B-tree, dependendo do padrão de pesquisas que este observa para o índice B-tree. Um índice hash pode ser parcial: não é necessário que todo o índice B-tree seja armazenado no *buffer*. Índices hash são criados sobre a pedido das páginas do índice que são acedidas com frequência.

De certa forma, o InnoDB adapta-se através do mecanismo *Adaptive Hash Index* para memória principal de grandes dimensões, exibindo uma arquitetura que se aproxima da arquitetura das bases de dados em memória principal.

Eis algumas das características deste tipo de índices:

Estes índices são usados apenas para comparações de igualdade que usam os operadores de comparação = ou <=> (mas são muito rápidos). Não são usados para os operadores de comparação como < que encontram um intervalo de valores.

O optimizador não pode utilizar um índice hash para acelerar operações do tipo ORDER BY. (Este tipo de índice não pode ser utilizado para pesquisar a próxima entrada por ordem).

O MySQL não consegue determinar aproximadamente quantas linhas existem entre dois valores (isto é usado pelo *range optimizer* de forma a decidir qual índice usar).

Apenas se podem usar chaves inteiras para procurar uma linha. (Com um índice B-tree, qualquer prefixo mais à esquerda da chave pode ser usada para encontrar linhas.)

3.3 Comandos Para Índices

Para criar um índice, é executado o seguinte comando:

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

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

index_type:
    USING {BTREE | HASH}
```

Como se pode constatar, o MySQL suporta 3 tipos de índices diferentes.

Um índice do tipo UNIQUE cria uma restrição de tal forma que todos os valores no índice sejam distintos. Um erro ocorre se tentar adicionar uma nova linha com um valor de chave que corresponda a uma linha existente. Para todos os motores de armazenamento, um índice UNIQUE permite que vários valores NULL para colunas que podem conter NULL. Se um valor de prefixo fôr especificado a partir de uma coluna num índice deste tipo, os valores dessa coluna deverão ser exclusivos dentro do prefixo.

A indexação acontece sempre sobre toda a coluna. Prefixos de colunas de indexação não são suportados.

E apesar de permitir tipos de dados SPATIAL, não suporta índices deste tipo.

Para remover um índice anteriormente criado, é executado o seguinte comando:

```
DROP INDEX index_name ON tbl_name
```

Para consultar os índices existentes, executa-se o seguinte comando:

```
SHOW INDEX FROM tbl_name
```

O MySQL inclui comandos que ajudam a analisar potenciais problemas de indexação. O comando EXPLAIN demonstra quais os índices (caso existam) seriam utilizados ao executar uma determinada *query.* Simplesmente coloca-se a cláusula EXPLAIN no início de uma *query* pretendida conforme se segue:

```
EXPLAIN SELECT *** FROM tabela WHERE cond
```

Durante a inicialização do servidor, pode-se ativar o mecanismo *Adaptive Hash Index* através do seguinte comando:

```
innodb_adaptive_hash_index
```

Para desligar o mecanismo *Adaptive Hash Index*, digita-se o seguinte comando também durante o startup do servidor:

```
--skip-innodb_adaptive_hash_index
```

3.4 Comparação com o Oracle

Tal como no Oracle, o MySQL utiliza B-trees nos índices por defeito. Como os índices Bitmap não são suportados pelo MySQL, o Oracle apresenta uma vantagem neste campo: ao suportar este tipo de índices, pesquisas que envolvam contagem de tuplos, acesso a tuplos com valores discretos com poucas entradas na tabela ou ainda junções de tuplos de diferentes tabelas envolvendo tuplos com valores "raros" serão mais eficientes ao utilizar este mecanismo comparativamente ao MySQL. Relativamente aos índices por hashing, tal como o Oracle, o InnoDB não suporta este tipo de índice. Porém, esta lacuna é colmatada com a presença do mecanismo de *Adaptive Index Hashing* conforme foi abordado.

4 Processamento e Otimização de Perguntas

Para entender o funcionamento de um sistema de gestão de bases de dados é importante perceber como esta faz o processamento e otimização de perguntas. Esta secção é dedicada a explicitar estes processos para as várias operações no MySQL (InnoDB).

4.1 Processamento e otimização de perguntas

O processamento das queries em MySQL segue os passos abaixo, começando pelo parser Byson/Yacc, seguindo uma abordagem top-down e fazendo uso duma máquina de estados finita, onde as queries são traduzidas, decompostas e validadas sintaticamente. No passo seguinte é efetuada a tradução para álgebra relacional onde se valida semanticamente a query, continuando depois para o passo de otimização. No optimizador, este tenta em primeiro lugar reduzir o número de tuplos com que vai trabalhar, depois o número de atributos desse duplos e finalmente avalia as condições join (select/project/join). Tendo em conta as estatísticas disponíveis (persistentes desde a versão 5.6 – até então apenas em memória), o plano de execução é então selecionado com base nos dados disponíveis e opções selecionadas, sendo implementado de seguida.

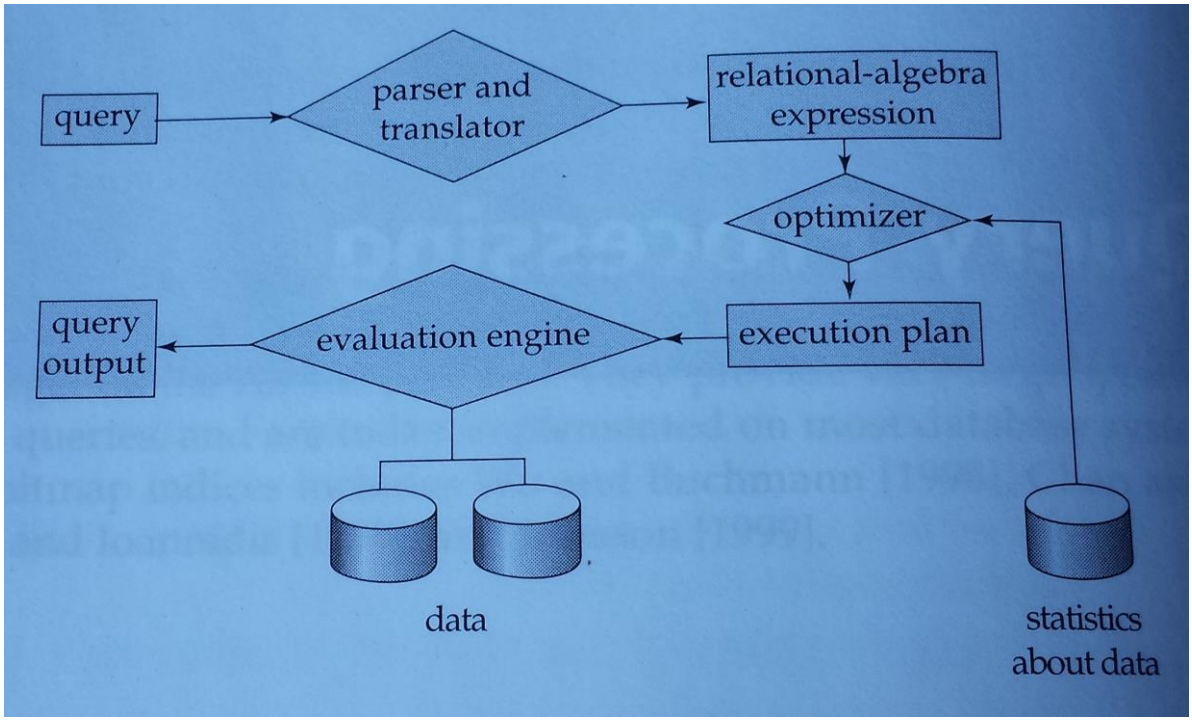


Ilustração 1: Ciclo de vida de uma query

4.1.1 Explain

Fazendo uso do explain, pode-se verificar o plano de execução das queries.

```
1 •  explain extended select * from country, organization, city
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	organization	NULL	ALL	NULL	NULL	NULL	NULL	153	100.00	NULL
1	SIMPLE	country	NULL	ALL	NULL	NULL	NULL	NULL	238	100.00	Using join buffer (Block Nested Loop)
1	SIMPLE	city	NULL	ALL	NULL	NULL	NULL	NULL	3...	100.00	Using join buffer (Block Nested Loop)

Caso se execute o comando show warnings a seguir ao explain extended, tem-se acesso a informação extra de reescrita ou otimização da query.

4.1.2 Analyze

Utilizado o analyze table forçamos o InnoDB a analisar a(s) tabelas(s) nomeadas de modo a atualizar a cardinalidade dos índices nas respetivas tabelas, fazendo 8 tentativas random em cada um dos índices. Durante o analyze a(s) tabela(s) respetivas ficam read locked. Esta distribuição de cardinalidade é usada para definir a ordem de junções e para decidir qual índice usar (se algum). Se a(s) tabela(s) não tiverem sido alteradas a operação não é efectuada.

4.1.3 Optimização da cláusula where

Algumas das optimizações possíveis para a cláusula where estão aqui apresentadas. Como a evolução do optimizador é constante, nem todas estão aqui presentes. São utilizadas de igual modo para o select, delete e update.

– Remoção de parênteses não necessários
(a AND b) AND c OR (((a AND b) AND (c AND d))))
-> (a AND b AND c) OR (a AND b AND c AND d)
– Substituição das variáveis pelas constantes respectivas
(a<b AND b=c) AND a=5
-> 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)
-> B=5 OR B=6
– Expressões constantes usadas por índices são avaliadas apenas 1 vez.
– Detecção prévia de expressões constantes inválidas – detecta rapidamente expressões impossíveis e não retorna tuplos.
– O conteúdo da cláusula having é fundido com o da cláusula where caso não se usa group by ou funções de agregação (count(), min(), etc…).
– Para cada tabela num join é simplificada a cláusula where para ser avaliada mais rapidamente e saltar o maior número de tuplos possíveis.
– Todas as tabelas constantes são lidas antes das outras tabelas na query. Uma tabela constante é qualquer das seguintes:
Uma tabela vazia ou tabela com 1 linha

Uma tabela que é usada numa cláusula where com a primary key ou indíce unique, onde todas as partes do indíce são comparadas a expressões constantes e definidas como not null.

Todas as tabelas seguintes são usadas como tabelas constantes:

SELECT * FROM tabela WHERE ***chave_primaria***=1;

SELECT * FROM tabela1,tabela2

WHERE tabela1.***chave_primaria***=1 AND tabela2.***chave_primaria***=tabela1.id;

– A melhor combinação para junção de tabelas é encontrada tentando todas as possibilidades. Se todas as colunas num order by e group by forem da mesma tabela, essa tabela é preferida em primeiro lugar na junção.

– Expressões constantes usadas por índices são avaliadas apenas 1 vez.

– Se a opção sql_small_result for usada o MySQL usa uma tabela temporária em memória.

– Cada indíce da tabela é considerado, e o melhor indíce é usado a menos que o optimizador acredite que é mais eficiente usar um table scan. O optimizador baseia a sua estimativa em fatores como a percentagem de tuplos da tabela presentes no melhor índice, tamanho da tabela, número de tuplos e tamanho de blocos I/O.

– Em casos específicos o MySQL pode apenas ler as linhas do índice sem consultar o ficheiro de dados. Se todas as colunas usadas do indíce forem numéricas, apenas a árvore do índice é usada para resolver a query.

– Antes de se tornarem output, todos os tuplos que não cumprirem a cláusula having são ignoradas.

4.1.4 Optimização da cláusula join

Left join e right join

O MySQL implementa o *A* LEFT JOIN *B* join_condition da seguinte maneira:

A tabela B fica dependente da tabela A e todas as tabelas de que A dependa.

A tabela A depende de todas as tabelas que são usadas na condição left join (exceptuando a B).

A condição left join é usada para decidir como obter os tuplos da tabela B (qualquer condição da cláusula where não é usada).

Todas as otimizações padrão são realizadas, com a exceção que a tabela é sempre lida após tidas as tabelas das quais ela depende. Se existe dependência circular o MySQL retorna um erro.

Todas as otimizações padrão da cláusula where são efetuadas.

Se existir um tuplo de A que verifique as condições na cláusula where e não existir um tuplo B que corresponda à condição on é gerado um tuplo extra com todas as colunas com o valor null.

O right join é implementado de maneira semelhante, sendo apenas invertidos os papéis.

4.1.5 Nested-loop join

O algoritmo NLJ lê tuplos da primeira tabela, passando 1 de cada vez para um nested loop que processa esse tuplo em conjunto com a próxima tabela presente no join. Este processo é repetido enquanto existirem tabelas a juntar. Devido ao facto do NLJ passar 1 tuplo de cada vez da tabela exterior para a tabela interior, tipicamente lê as tabelas dos loops interiores muitas vezes.

4.1.6 Block nested-loop join

O algoritmo BNLJ faz uso do buffering de tuplos lidos nos loops exteriores de modo a reduzir o número de vezes que as tabelas interiores têm que ser lidas.

O MySQL faz uso do buffering nas seguintes condições:

– A variável de sistema join buffer size determina o tamanho de cada buffer de junção.

– Buffering de junções pode ser de tipo all, index ou range, sendo também aplicável a outer joins.

– Um buffer é alocado para cada junção que pode ser buffered, deste modo uma só query pode usar múltiple buffers de junção.

– Um buffer de junção nunca é alocado à primeira tabela não constante, mesmo que este seja de tipo all ou index.

– Um buffer de junção é alocado antes da execução da junção e liberto após estar finalizada.

– Apenas as colunas com interessa para a junção são guardadas no buffer de junção, não as linhas inteiras.

O número de scans da tabela interior diminui à medida que o valor do join_buffer_size aumenta até chegar ao ponto em que o join_buffer_size é grande o suficiente para albergar todos os tuplos da tabela exterior. Após alcançar este ponto, não existe qualquer melhoria possível de tempo de execução em aumentar o join_buffer_size.

4.1.7 Optimização da cláusula order by

Em casos particulares o MySQL usa um índice para satisfazer uma cláusula order by sem ser necessário fazer qualquer ordenação extra. Mesmo que a cláusula order by não corresponda exatamente ao índice este pode ser usado, desde que a parte não usada do índice e as colunas extra da cláusula order by seja constantes na cláusula where.

Nalguns casos o MySQL pode não conseguir usar índices para satisfazer a cláusula order by, apesar de ainda is usar para obter os tuplos para satisfazer a cláusula where. Nestes casos incluem-se os seguintes:

– Uso de order by em chaves diferentes:

SELECT * FROM tabela1 ORDER BY ***chave1***, ***chave2***;

– Uso de order by em partes não consecutives de uma chave:

SELECT * FROM tabela1 WHERE ***chave2***=***constante*** ORDER BY ***parte_chave2***;

– Mistura de asc e desc:

SELECT * FROM tabela1 ORDER BY ***parte_chave1*** DESC, ***parte_chave2*** ASC;

– A chave usada para satisfazer a cláusula where não é a mesma da usada no order by:

SELECT * FROM tabela1 WHERE ***chave2***=***constant*** ORDER BY ***chave1***;

– Uso da cláusula order by com uma expressão que não inclui apenas a chave:

SELECT * FROM tabela1 ORDER BY ABS(***chave***);

SELECT * FROM tabela1 ORDER BY -***chave***;

– Junção de bastantes tabelas, não sendo todas as colunas da cláusula order by da primeira tabela não constante que é usada para obter tuplos. (A primeira tabela no output do explain que não tem um tipo de junção const.)

– Expressões distintas nas cláusulas order by e group by.

– Apenas um prefixo da coluna presente na cláusula order by está indexado. Neste caso, o indíce não é suficiente para determiner a ordenação.

Por exemplo, se tivermos uma coluna char(20), mas apenas os primeiros 10 bytes estarem indexados, o indíce não consegue distinguir a partir do 10º byte, sendo necessário usar o filesort.

– O tipo de indíce usado não mantém os tuplos ordenados (por exemplo o indíce hash).

4.1.8 Optimização da cláusula group by

A maneira mais comum de satisfazer a cláusula group by é fazendo um scan à tabela inteira e criar uma nova tabela temporária em que todos os tuplos de cada grupo são consecutivos, usando a tabela para descobrir grupos e aplicar funções de agregação. Em alguns casos o MySQL é capaz de fazer significativamente melhor que isso evitando a criação de tabelas temporárias fazendo uso de índices.

As pré-condições mais importantes para o uso de índices nas cláusulas group by é que as colunas referenciem atributos do mesmo indíce, e que o indíce tenha as chaves ordenadas (por exemplo ser um indíce btree e não hash). A substituição das tabelas temporárias por uso de índices também depende de que pates de um indíce são usadas na query, que condições são especificadas para essas partes, e das funções de agregação seleccionadas.

Existem 2 maneira de executar a cláusula group by fazendo uso de índices, como detalhado de seguida.

No primeiro caso (LIS) a operação de agrupamento á aplicada juntamento com todos os predicados range. No segundo caso (TIS), primeiro é efectuado o range scan e só depois se faz o agrupamento dos tuplos resultantes.

No MySQL o group by é usado para a ordenação, logo o servidor pode também aplicar otimizações order by para fazer o agrupamento.

4.1.9 Loose index scan

O processamento mais eficiente do group by acontece quando é usado um índice para obter diretamente as colunas a agrupar. Com estes métodos de acesso, o MySQL usa a ordenação natural das chaves de alguns tipos de índice (por exemplo btree). Isto permite que não considerar todas as chaves do índice que satisfazem a cláusula where. O nome **loose index scan** advém do facto de apenas uma parte das chaves do indíce serem consideradas. No caso de não existir cláusula where, o loose index scan apenas lê um número de

chaves do índice idêntico ao número de agrupamentos, que pode ser bastante mais reduzido que o número de chaves totais do índice. Caso existam condições range na cláusula where, o loose index scan procura a primeira chave que verifica a condição, lendo o menor número possível de chaves do índice. Isto é possível segundo as seguintes condições:

– A query é efetuada apenas sobre uma tabela.

– O group by apenas referencia colunas que constituem o prefixo de um índice. Por exemplo, se uma tabela tabela1 tem um index em (coluna1, coluna2, coluna3), o loose index scan é aplicável se a query tiver group by coluna1, coluna2 mas não é aplicável se tiver group by coluna2.

– As únicas funções de agregação usadas no select são min() e max(), e todas elas referenciam a mesma coluna. A coluna deve pertencer ao índice e deve estar a seguir às colunas usadas no group by.

– Todas as outras componentes do índice sem ser as usadas no group by devem ser constantes, isto é, devem ser referenciadas em igualdades com constantes – com a exceção dos argumentos para as funções de agregação min() e max().

– Para as colunas presentes no índice todos os seus valores deve ser indexados, e não apenas os prefixos. Por exemplo, se coluna1 varchar(20) e indice1 (coluna1(10)), o índice não pode ser usado para o loose index scan.

Assumindo que existe indice1(c1,c2,c3) na tabela1(coluna1, coluna2, coluna3, coluna4). O loose index scan poderá ser usado nas seguintes queries:

SELECT coluna1, coluna2 FROM tabela1 GROUP BY coluna1, coluna2;

SELECT DISTINCT coluna1, coluna2 FROM tabela1;

SELECT coluna1, MIN(coluna2) FROM tabela1 GROUP BY coluna1;

SELECT coluna1, coluna2 FROM tabela1 WHERE coluna1 < ***const*** GROUP BY coluna1, coluna2;

SELECT MAX(coluna3), MIN(coluna3), coluna1, coluna2 FROM tabela1 WHERE coluna2 > ***const*** GROUP BY coluna1, coluna2;

SELECT coluna2 FROM tabela1 WHERE coluna1 < ***const*** GROUP BY coluna1, coluna2;

SELECT coluna1, coluna2 FROM tabela1 WHERE coluna3 = ***const*** GROUP BY coluna1, coluna2;

E as seguintes queries não podem usar o loose index scan devido às razões apresentadas:

Existência de funções de agregação além de min() e max():

SELECT coluna1, SUM(coluna2) FROM tabela1 GROUP BY coluna1;

As colunas usadas no group by não formam o prefix do indice:

SELECT coluna1, coluna2 FROM tabela1 GROUP BY coluna2, coluna3;

A query referencia parte de uma chave que aparece na cláusula group by e para a qual não existe igualdade com uma constante:

SELECT coluna1, coluna3 FROM tabela1 GROUP BY coluna1, coluna2;

Se a query incluisse a **where coluna3 = *const*** o loose index scan poderia ser usado.

O loose index scan pode ser aplicado a outras funções de agregação além de min() e max():

avg(distinct), **sum(distinct)**,e **count(distinct)** sendo que **avg(distinct)** e **sum(distinct)**levam apenas 1 argumento. **count(distinct)** pode ter mais que 1 coluna de argumento.

Não poderá haver order by ou distinct na query.

As limitações apresentadas anteriormente ainda se aplicam.

Assumindo que existem **indice1(coluna1,coluna2,coluna3)** em **tabela1(coluna1,coluna2,coluna3,coluna4)**. O loose index scan poderá ser usado para as seguintes queries:

SELECT COUNT(DISTINCT coluna1), SUM(DISTINCT coluna1) FROM tabela1;

SELECT COUNT(DISTINCT coluna1, coluna2), COUNT(DISTINCT coluna2, coluna1) FROM tabela1;

E não é aplicável às seguintes queries:

SELECT DISTINCT COUNT(DISTINCT coluna1) FROM tabela1;

SELECT COUNT(DISTINCT coluna1) FROM tabela1 GROUP BY coluna1;

4.1.10 Tight index scan

O tight index scan pode ser um full index scan ou um range index scan dependente das condições da query.

Quando as condições para um loose index scan não forem cumpridas, pode ainda ser possível evitar a criação de tabelas temporárias para resolves as clásulas group by. Se existirem condições de range na cláusula where, este método apenas lê as chaves que verificam essas condições, caso cntrário realiza um index scan. O nome **tight index scan** advém do facto deste método resultar na leitura de todas as chaves (ou as que cumpram o range se presente na cláusula where). O agrupamento é apenas efectuado após

todas as chaves que satisfaçam a query terem sido encontradas. Para este método funcionar é suficiente que para qualquer coluna que seja necessária para a construção do prefixo da chave índice que esteja em falta no group by seja compensada por uma condição de igualdade constante. Se for necessário ordenar os resultados do group by, é possível formar chaves de pesquisa que são prefixos do índice, e deste modo o MySQL evita operações extra de ordenação porque a pesquisa com prefixos num índice ordenado já retornas as chaves ordenadas.

Assumindo que existe **indice1(coluna1,coluna2,coluna3)** em **tabela1(coluna1,coluna2,coluna3,coluna4)**. As seguintes queries não funcionam com o loose index scan, mas funcionam com o tight index scan.

Existe uma coluna da chave em falta na cláusula **group by**, sendo compensado por **c2 = 'a'**:

SELECT coluna1, coluna2, coluna3 FROM tabela1 WHERE coluna2 = 'a' GROUP BY coluna1, coluna3;

A clásula **group by** não começa com a primeira parte da chave, mas existe uma condição que fornece uma constante para o compensar:

SELECT coluna1, coluna2, coluna3 FROM tabela1 WHERE coluna1 = 'a' GROUP BY coluna2, coluna3;

4.2 Comparação com o Oracle

As maiores lacunas do InnoDB em relação ao Oracle são a falta de variedade de tipos de índices e a falta de materialized views. A falta de por exemplo índices bitmap limita significativamente o nível de otimização possível o que em geral é factor decisivo em data-warehousing, sendo o MySQL descartado como opção. Apesar do MySQL não suportar materialized views, pode-se 'criar' algo semelhante através da criação de uma tabela (que servirá de view), triggers e stored procedures que preenchem esta tabela, o que aproxima mais os dois sistemas neste ponto.

No que toca a otimização de perguntas, o Oracle 11g apresenta uma maior versatilidade ao incluir Pipelining e processamento paralelo, enquanto que o MySQL apenas inclui Materialização.

Com a inclusão de Pipelining, o Oracle 11g garante uma melhor execução de expressões complexas, pois permite o processamento em paralelo de várias operações, enquanto que no MySQL as operações tem que ser tratadas sequencialmente.

No que toca à otimização de expressões básicas, o Oracle 11g e MySQL são muito similares, pelo que _e impossível fazer uma comparação crítica neste aspeto. O Oracle 11g também utiliza índices durante o processamento e otimização de perguntas.

A diferença neste aspeto resume-se ao facto de o Oracle 11g incluir processamento de perguntas em paralelo através de Pipelining, pelo que o seu uso e mais adequado que o de MySQL caso se antecipe a execução de várias perguntas complexas.

5 Gestão de transações e controlo de concorrência

O termo transação no contexto de base de dados refere-se a um conjunto de operações que agem com uma única unidade lógica. Uma das principais razões para a escolha do engine InnoDB foi o seu suporte a transações e a forte aderência que tem ao modelo ACID de modo a manter a integridade da base de dados. O InnoDB é um sistema multi versionamento, que mantém informação de filas eliminadas ou alteradas de modo a suportar concorrência e rollback.

5.1 Propriedades ACID

5.1.1 Atomicidade (A)

A atomicidade garante que as várias operações da transação são todas efetuadas ou nenhuma o é. O MySQL faz uso do start transaction, commit, rollback e autocommit para controlar esta propriedade.

O autocommit é uma variável de sistema que se encontra activa por defeito, e é tratada como uma variável de sessão. Este pode ser desativado/ativado com o seguinte comando:

```
SET autocommit=0;
```

Quando se usa o start transaction o autocommit é desativado é ao final da transação que ocorre quando for efectuado um commit ou rollback. Os modos read write e read only definem o modo de acesso da transação, sendo que o modo read write está definido por defeito.
Algumas expressões causam um commit implicitamente antes e após a execução das mesmas, sendo de evitar a sua presença dentro de transações. A maioria das expressões create/alter/drop encontram-se neste grupo, excetuando as que afetam tabelas temporárias.

5.1.1.1 Rollback

O InnoDB mantém um log das operações efectuadas, mantendo uma imagem das páginas pré e pós alteração (physical logging) e alterações lógicas ao conjunto de dados (logical logging). Ao conjunto deste dois tipos de logging chama-se physiological logging. O rollback no InnoDB recorre a este log para recuperar o estado inicial da transacção.

5.1.2 Consistência (C)

Esta propriedade garante que a base de dados se mantém consistente no final de uma transação, podendo encontrar-se inconsistente durante a mesma. As restrições são apenas verificadas no final da transação. O MySQL InnoDB garante esta propriedade recorrendo ao doublewrite buffer e crash recovery. O InnoDB antes de escreve as páginas nos ficheiros de dados, escreve-as em primeiro lugar para uma área contígua chamada doublewrite buffer. Apenas após o sucesso desta primeira escrita é que o InnoDB escreve as páginas para o local respectivo nos ficheiros de dados. Caso haja alguma falha no meio da segunda escrita, o InnoDB poderá obter uma cópia fiável da área do doublewrite buffer durante o crash recovery.

Apesar dos dados serem escritos duas vezes, não requer o dobro das operações I/O. Os dados são escritos para o buffer como um bloco sequencial, com uma única chamada fsync() ao sistema operativo.
Quando o MySQL é reiniciado após um crash são iniciadas as atividades de limpeza. Alterações das transações incompletas são reiniciadas usando o redo log. Alterações que foram commited antes do crash e ainda não copiadas para os ficheiros de dados são reconstruídas através do doublewrite buffer. Quando a base de dados é desligada normalmente este tipo de atividades é tratada pela operação purge.

Durante a operação normal os dados commited são armazenados no change buffer durante algum tempo antes de serem escritos para os ficheiros de dados. Existe sempre um compromisso entre manter os ficheiros de dados actualizados – que causa algum overhead durante a operação normal; e o buffering de dados – que pode causar que um crash recovery após um shutdown demore mais tempo.

5.1.3 Isolamento (I)

O isolamento garante que, apesar de múltiplas transações serem executadas concorrentemente, estas permanecem ignorantes da existência umas das outras. Os resultados intermédios de uma transação devem permanecer escondidos das restantes transações. O MySQL garante esta propriedade através do locking, permitindo ainda controlar os moldes em que é efetuada através do isolation level.
Através da definição do isolation level, o balanço pode ser ajustado entre performance, confiança, consistência e reprodutibilidade dos resultados quando existem transações concorrentes que efetuam alterações e realizam queries ao mesmo tempo.
Os níveis de isolamento suportados pelo InnoDB são os seguintes (de maior proteção para menor): serializable, repeatable read, read committed e read uncommitted, sendo que o nível de isolamento por defeito é o repeatable read para todas as operações.
Utilizadores mais avançados tendem a usar o read committed caso necessitem da escalabilidade para o processamento OLTP, ou durante operações de data warehousing onde pequenas inconsistências não afectam os resultados agregados de grandes volumes de dados. Os níveis limite (serializable e read uncommitted) mudam o comportamento do processamento de tal modo que raramente são usadas.

5.1.3.1 Serializable

Este é o nível de isolamento que mais restricções impõe, prevenindo que qualquer outra transacção insira ou mude dados lidos pela transacção até estar ser finalizada. Deste modo a mesma query pode ser executada várias vezes dentro de uma transacção e ter a certeza que irá obter sempre os mesmos resultados. Qualquer tentativa de alterar dados committed por outra transacção forçam a transacção actual a esperar.

Este é o nível de isolamento por defeito nos standards SQL, mas sendo este nível de isolamento raramente necessário, no InnoDB o defeito é o nível de isolamento seguinte (repeatable read).

5.1.3.2 Repeatable read

O nível de isolamento por defeito no InnoDB. Este nível de isolamento bloqueia os non-repeatable reads mas não os phantom reads. Usa uma estratégia moderada de locking, de modo a que para que todas as queries dentro de uma transacção vejam os mesmos dados, tal como eles eram no início da transacção.

Quando uma transacção com este nível de isolamento realiza update … where, delete … where, select … where as outras transacções têm que esperar.

5.1.3.3 Read committed

Este é o nível de isolamento que relaxa algumas das proteções entre transações de modo a beneficiar a performance. As transações não podem ver dados uncommitted de outras transações mas podem ver as committed. Deste modo, apesar dos dados serem fiáveis, estes dependem do timing das transações.

Quando uma transação com este nível de isolamento realiza update … where ou delete … where as outras transações podem ter que esperar. No caso do select … for update este pode ser realizada ser fazer esperar outras transações.

5.1.3.4 Read uncommitted

O nível de isolamento que providencia o menor nível de proteção entre transações. De modo a maximizar a performance relaxasse ao máximo a estratégia de locking, de tal modo que se podem aceder a alterações de outras transações não committed (dirty read). Este nível de isolamento deve ser usado com o máximo de cuidado, pois os resultados podem não ser consistentes ou reproduzíveis, dependendo do que as outras transacções estiverem a fazer. Tipicamente as transações este nível de isolamento apenas realizam queries, e não inserts/updates/deletes.

5.1.3.5 Dirty read

Os dados ainda não committed de outras transações podem ser consultados.

5.1.3.6 Non-repeatable read

Tuplos actualizados ou apagados por outras transacções commited durante a transacção actual podem ser lidos.

5.1.3.7 Phantom read

Tuplos inseridos por outras transacções commited a durante a transacção actual podem ser lidos.

Isolamento	Serializable	Repeatable read	Read committed	Read uncommitted
Dirty read	x	x	x	✓

Non-repeatable read	x	x	✓	✓
Phantom read	x	✓	✓	✓

Tabela 1: Tipo de leitura permitida x Nível de isolamento no InnoDB

5.1.4 Durabilidade (D)

Esta propriedade garante que após o sucesso de uma transacção (commit), as alterações persistem na base de dados. É garantida através do uso do registo no log buffer no momento do commit e do flush que é efetuado de seguida. Esta propriedade envolve a interacção do MySQL com o CPU, rede e dispositivos de armazenamento.

5.1.4.1 Locking

Além do InnoDB permitir o standard row-level-lock onde existem dois tipos de locks (shared (s) locks e exclusive (x) locks), este permite a múltipla granularidade de locks / coexistência de row-level-locks com intention locks – ou table locks – (intention shared (Is) e intention exclusive (Ix). Qualquer destes modos não permitem a escrita nas linhas/tabelas correspondentes.

5.1.4.2 Shared (s) locks

Este tipo de locks permite que uma transação concorrente efetue um read na linha. Se existir um segundo pedido de shared(s) lock este pode ser aceite imediatamente. Neste caso ambas as transações passam a deter um shared(s) lock na linha. Caso exista um pedido de shared(x) lock este terá que esperar.

5.1.4.3 Exclusive (x) locks

Este tipo de locks não permite o acesso de outras transações à linha da tabela. Qualquer pedido de lock a esta linha terá que esperar.

5.1.4.4 Shared (Is) locks

Antes de uma transacção poder obter um shared (s) lock tem que obter em primeiro lugar um shared (Is) lock. Este tipo de transações apenas bloqueia operações que afetam a tabela completa.

5.1.4.5 Exclusive (Ix) locks

Antes de uma transacção poder obter um exclusive (x) lock tem que obter em primeiro lugar um exclux) lock. Este tipo de transacções apenas bloqueia operações que afectam a tabela completa.

Lock	X	IX	S	IS
X	x	x	x	x
IX	x	✓	x	✓
S	x	x	✓	✓
IS	x	✓	✓	✓

Tabela 2: Compatibilidade de locks entre transacções concorrente

5.1.4.6 Deadlocks

O InnoDB detecta automaticamente os deadlocks, efectuando um rollback à transacção (ou transacções) para os desbloquear. A transacção escolhida para ser alvo do rollback é determina pelo menor número de linhas inseridas/actualizadas/apagadas.
Caso as seguintes definições **innodb_table_locks = 1** (valor por defeito) e **autocommit = 0** não se verifiquem o InnoDB não detecta os table deadlocks automaticamente, recorrendo à variável de sistema **innodb_lock_wait_timeout** para periodicamente verificar a existência desta situação.
Quando o InnoDB executa um rollback duma transacção, todos os locks dessa transacção são libertos.

5.1.4.7 Savepoints

Tal como o nome sugere, os savepoints marcam um ponto na transacção após o qual, se for efectuado um rollback to savepoint (no qual se pode nomear o savepoint especifico), retorna o estado da transacção a esse savepoint. Todos os locks efectuados posteriormente não são libertos porque o MySQL não consegue identificar qual a operação dentro da transacção que provocou o lock (identifica apenas a transacção).

5.2 Comparação com o Oracle

Ambos os sistemas de base de dados são muito semelhantes no que toca a transacções, sendo que ambos respeitam as propriedades ACID, usam os mesmos níveis de isolamento e suportam Row e Table level locking. A única diferença mais evidente será a não existência de nested transactions no mySQL ao contrário do Oracle, sendo que o que mais se assemelha a tal do lado do MySQL é o mecanismo de savepoints.

6 Suporte Para Bases De Dados Distribuídas

Com a crescente demanda de serviços globalizados na internet, as bases de dados distribuídas têm um peso cada vez mais importante nesse tipo de sistemas, o MySQL não foge á regra. Neste capítulo, falaremos de como o MySQL integra os vários tipos de BDs distribuídas, suas características e limitações, bem como exemplos práticos das mesmas.

6.1 Homogéneas

6.1.1 Replicação

No MySQL a replicação define-se de um dono dos dados (master), e as copias replicada (slaves), permitindo que os dados de um servidor de BD MySQL (master), possam ser replicado para um ou mais servidores de BD MySQL (slaves). A replicação é assíncrona por defeito, logo os *slaves*, não precisam estar conectado permanentemente para receber atualizações do *master*. Dependendo da configuração, é nos possível **replicar** todas as BDs, algumas BDs, ou só mesmo tabelas selecionadas dentro de uma BD.

Tipos de repicação suportada no MySQL:

Statement Based Replication (SBR), que replica instruções SQL na sua intrega.

Row Based Replication (RBR), o qual replica apenas as linhas alteradas.

Mixed Based Replication (MBR), que replica as instruções SQL em conjunto com as linhas alteradas.

MySQL suporta replicação de transação com base em Global Transaction Identifiers (GTIDs). Ao usar este tipo de replicação, não é necessário para trabalhar diretamente com ficheiros de log ou posições dentro desses ficheiros, o que simplifica muitas tarefas de replicação comum. Como a replicação usando GTIDs é totalmente transacional, a coerência entre o *master* e o *slave* é garantida, desde que todas as transações commited no *mastre* também sejam aplicadas no *slave*.

Configuração do *master*:

É necessário ativar o log binário no ficheiro de configuração do *master* e estabelecer um ID único ao servidor *master* (dentro do esquema de servidores *master slave*). Se isso já não foi feito, esta parte da configuração mestre requer a reinicialização do servidor. Log binário deve estar ativado no master porque o log binário é a base para o envio de alterações de dados do *master* para os seus *slaves*. Se o log binário não estiver ativado, a replicação não será possível.

Ficheiros `my.cnf` ou `my.ini` do *master*:

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

Configuração do *slave*:

No *slave* a configuração é muito semelhante ao do *master*, com a exceção da ativação do log binário. Relativamente ao *sever-id*, terá de atribuir um valor não atribuído.

Ficheiros `my.cnf` ou `my.ini` do *slave*:

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

Criação de um utilizador para a replicação:

```
mysql> CREATE USER 'repl'@'%.mydomain.com' IDENTIFIED BY 'slavepass';
mysql> GRANT REPLICATION SLAVE ON *.* TO 'repl'@'%.mydomain.com';
```

Criar uma snapshot usando *mysqldump*:

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

Para iniciar as threads do *slave*:

```
mysql> START SLAVE;
```

Alterar a configuração do *slave* para *master*:

```
mysql> CHANGE MASTER TO
```

```
-> MASTER_HOST='master_host_name',
-> MASTER_USER='replication_user_name',
-> MASTER_PASSWORD='replication_password',
-> MASTER_LOG_FILE='recorded_log_file_name',
-> MASTER_LOG_POS=recorded_log_position;
```

6.2 Heterogéneas

6.2.1 Fragmentação

O MySQL um conjunto de possibilidades de fragmentação, tanto a nível vertical (separação de clunas ou de tabelas TABLESPACE). Mas para bases de dados distribuídas praticamente não tem qualquer suporte, dependendo utilizador/programador com o uso de outras funcionalidades com a replicação assíncrona, views, triggers, e afins em conjunção com a TABLESPACE, criar um sistema de particionamento distribuído.

6.3 Transações distribuídas (XA)

Transações XA no MySQL-*innodb* são projetadas para permitir transações distribuídas, onde um *Transaction Maneger(TM)* controla uma transação que envolve vários recursos. Tais recursos são geralmente um SGBD, mas poderia ser qualquer tipo de recursos. Todo o conjunto de operações transacionais necessárias, é chamada uma transação global (*global-transaction*). Cada subconjunto de operações que envolvem um único recurso é chamado de uma transação local (*local-transaction*). Estas transações seguem o protocolo 2-Phase Commit(2PC) por defeito, garantindo as propriedades ACID. Com o *first-commit(fc)*, o TM a diz cada *Resource Maneger(RM)* para preparar um *effective-commit*, e aguarda uma mensagem de confirmação. As alterações não são ainda efetivada neste momento. Se qualquer um dos RMs encontrou um erro, o TM reverte a *global-transaction*. Se todos os recursos comunicar que o *fc* foi bem sucedido, o TM pode requerer um *second-commit*, o que faz com que as alterações entrem em vigor.

Sintaxe:

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]

Uma transação XA progride através dos seguintes estados:

Use XA START para iniciar uma transação XA e colocá-lo no estado ativo.

Para uma transação XA "ACTIVE", declarar as instruções SQL que compõem a operação e em seguida, emitir uma declaração XA END, Que coloca a transação em IDLE.

Para uma transação XA "IDLE", é possível emitir tanto uma instrução XA PREPARE ou uma instrução XA COMMIT ... ONE PHASE:

XA PREPARE coloca a transação no estado *prepared*. Neste momento declaração XA RECOVER irá incluir no seu retorno o valor da transação *xid*, porque XA RECOVER listas de todas as transações XA que estão no estado *prepared*.

XA COMMIT ... ONE PHASE prepara e confirma (commit) a transação. O valor *xid* não será listado por XA RECOVER porque a transação é dada como termina.

Para uma transação XA *prepared*, é possível emitir uma XA COMMIT para confirmar (commit) e terminar a transação, ou XA ROLLBACK para reverter e finalizar a transação.

Exemplo simples de uma transação XA:

mysql> **XA START 'xatest';**

Query OK, 0 rows affected (0.00 sec)

```
mysql> INSERT INTO mytable (i) VALUES(10);
Query OK, 1 row affected (0.04 sec)

mysql> XA END 'xatest';
Query OK, 0 rows affected (0.00 sec)

mysql> XA PREPARE 'xatest';
Query OK, 0 rows affected (0.00 sec)

mysql> XA COMMIT 'xatest';
Query OK, 0 rows affected (0.00 sec)
```

6.4 Outras ferramentas (motores)

6.4.1 MySQL Cluster (NBD)

MySQL Cluster é desenhado, em torno de uma arquitetura ACID multi-master distribuídos, do qual resulta a inexistência de um ponto central de falha. MySQL Cluster usa sharding automático (particionamento) para dimensionar as operações de leitura e escrita em função do hardwarer disponível.

6.4.2 Federated

O Federated permite um administrador de base de dados (ABD) criar na BD apontadores lógicos para tabelas que existem em outros servidores MySQL, e assim unir as bases de dados (físicas) distribuídas, de forma a criar uma ou mais base de dados lógicas.

6.5 Comparação com o SGBD da Oracle

No âmbito das bases de dados distribuídos BDD, consegue disponibilizar soluções muito mais eficientes robustas, tanto para BDDs homogenias com heterogenias, o que seja um SGBD mais tolerante a falhas. Mas se a necessidade é ter um sistema simples, leve, barato e sem muito ênfase na sincronização, o MySQL é o eleito.

7 Outras características do MySQL

Neste capítulo vamos estudar, algumas das características não abordas em capítulos anteriores. Sendo o MySQL um dos SGBD de eleição, quando se fala em aplicações e serviços WEB, o mesmo oferece um conjunto de ferramentas que permitam a implementação e uso das mesmas.

7.1 Suporte para a WEB

7.1.1 Suporte para XML

Para importar dados em formato XML, o MySQL disponibiliza o comando *LOAD XML*.

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

Como apresentado no exemplo em cima, o comando *LOAD XML* importa do ficheiro *file_name.xml*, os dados em formado XML para a tabela *[db_name.]tblname*.

Este comando suporta três tipo de formatos em XML, eles são:

- Nome da coluna como atributos e valor da coluna como valor de atributo:

<*tuplo coluna1*="*valor1*" *coluna2*="*valor2*" .../>

- Nome de coluna como **tag** e seu valor como conteúdo:

<*tuplo*>
<*coluna1*>*valor1*</*coluna1*>

<*coluna2*>*valor2*</*coluna2*>
</tuplo>

- O atributo *nome* da **tag** *<campo>* contém o nome da coluna, e o conteúdo o seu valor:

<tuplo>
<campo nome='*coluna1*'>*valor1*</campo>
<campo nome='*coluna2*'>*valor2*</campo>
</tuplo>

Para exportação e dados, o MySQL oferece um conjunto de comandos, *mysql –xml* e *mysqldump –xml*.

shell> ***mysql --xml 'SELECT * FROM mytable' > file.xml***

e

shell> ***mysqldump --xml 'SELECT * FROM mytable' > file.xml***

Exemplo de importação e exportação de XML de e para Mysql.

USE test;

CREATE TABLE person (
person_id INT NOT NULL PRIMARY KEY,
fname VARCHAR(40) NULL,
lname VARCHAR(40) NULL,
created TIMESTAMP
);

Apos a criação da tabela *person*, importamos o conteúdo do ficheiro em xml, *person.xml* para a mesma.

<?xml version="1.0"?>
<list>
<person person_id="1" fname="Pekka" lname="Nousiainen"/>
<person person_id="2" fname="Jonas" lname="Oreland"/>

<person person_id="3"><fname>Mikael</fname><lname>Ronström</lname></person>
<person person_id="4"><fname>Lars</fname><lname>Thalmann</lname></person>
<person><field name="person_id">5</field><field name="fname">Tomas</field>
<field name="lname">Ulin</field></person>
<person><field name="person_id">6</field><field name="fname">Martin</field>
<field name="lname">Sköld</field></person>
</list>

Importação do *person.xml*.

```
mysql> LOAD XML LOCAL INFILE 'person.xml'
    -> INTO TABLE person
    -> ROWS IDENTIFIED BY '<person>';
```

Query OK, 6 rows affected (0.00 sec)
Records: 6 Deleted: 0 Skipped: 0 Warnings: 0

Apos uma query à tabela em questão, obtemos o seguinte resultado.

```
mysql> SELECT * FROM person;
+-----------+--------+------------+---------------------+
| person_id | fname  | lname      | created             |
+-----------+--------+------------+---------------------+
|         1 | Pekka  | Nousiainen | 2007-07-13 16:18:47 |
|         2 | Jonas  | Oreland    | 2007-07-13 16:18:47 |
|         3 | Mikael | Ronström   | 2007-07-13 16:18:47 |
|         4 | Lars   | Thalmann   | 2007-07-13 16:18:47 |
|         5 | Tomas  | Ulin       | 2007-07-13 16:18:47 |
|         6 | Martin | Sköld      | 2007-07-13 16:18:47 |
+-----------+--------+------------+---------------------+
6 rows in set (0.00 sec)
```

Finalmente exporta-se a tabela *person*, para o ficheiro *person-dump.xml* e visualiza-se o seu conteúdo.

```
shell> mysql --xml -e "SELECT * FROM test.person" > person-dump.xml
shell> cat person-dump.xml
<?xml version="1.0"?>

<resultset statement="SELECT * FROM test.person" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <field name="person_id">1</field>
    <field name="fname">Pekka</field>
    <field name="lname">Nousiainen</field>
    <field name="created">2007-07-13 16:18:47</field>
  </row>

----------------------------//----------------------------
----------------------------//----------------------------

  <row>
    <field name="person_id">6</field>
    <field name="fname">Martin</field>
    <field name="lname">Sköld</field>
    <field name="created">2007-07-13 16:18:47</field>
  </row>
</resultset>
```

7.1.2 Suporte para xPath

O MySQL permite fazer queries a um ficheiro/string(fragemento) XML, com o uso da comando *ExtractValue(xml_frag,xpath_expr)*, em que *xml_frag* é o fragmento XML e *xpath_expr* a expreção da query em xPath.

Exemplo de query a um ficheiro XML.

Importação do conteúdo no ficheiro *client_citizenships.xml*, para a variável *@xml*.

```
SET @xml = LOAD_FILE("c:\\client_citizenships.xml");
```

Query xPath.

```
SELECT    ExtractValue(@xml, '//row[2]/field[1]/@name'),
     ExtractValue(@xml, '//row[2]/field[1]');
```

Fragmento da variável @*xml*.

```
...
</row>

<row> (row[2])
   <field name="client_id">2</field> (field[1])
   <field name="date_of_birth">1944-01-15</field>
...
```

Resulto da query *xPath*.

```
+-----------------------------------+-----------------------------+
|EV(@xml, "//row[2]/field[1]/@name") |EV(@xml, '//row[2]/field[1]')
|
+-----------------------------------+-----------------------------+
|client_id                  |2                    |
+-----------------------------------+-----------------------------+
```

Para fazer updates no XML, o MySQL disponibiliza o comando *UpdateXML(xml_target, xpath_expr, new_xml)* , em que *xml_target* é o alvo do update, *xpath_expr* a expressão da query em xPath e *new_xml* a alteração a ser feita.

7.1.3 Suporte para Semantic Web (RDF/Schema e SPARQL)

Neste campo o MySQL não tem qual quer suporte direto RDF/Schema ou SPARQL, este suporte só é possível mediante aplicações de terceiros.

7.2 Stored Procedures & Functions

Um *Procedure ou Function* é um conjunto de declarações em SQL que podem ser guardadas na base de dados. Uma vez assim feito, é possível ao utilizador usar os mesmos ao invés escrever as instruções individuais repetidamente.

Sintaxe:

CREATE
CREATE PROCEDURE *sp_name* ([*parameter*[,...]])
  [*characteristic* ...] *routine_body*

CREATE FUNCTION *sp_name* ([*parameter*[,...]])
  [RETURNS *type*]
  [*characteristic* ...] *routine_body*

parameter:
  [IN | OUT | INOUT] *param_name type*

type:
  Any valid MySQL data type

characteristic:
  LANGUAGE SQL
 | [NOT] DETERMINISTIC
 | SQL SECURITY {DEFINER | INVOKER}
 | COMMENT '*string*'

routine_body:
  Valid SQL procedure statements or statements

ALTER
ALTER {PROCEDURE | FUNCTION} *sp_name* [*characteristic* ...]

characteristic:
  NAME *new_name*
 | SQL SECURITY {DEFINER | INVOKER}
 | COMMENT '*string*'

DROP
DROP {PROCEDURE | FUNCTION} [IF EXISTS] *sp_name*

Apesar da sua natureza muito semelhante, as suas diferenças fazem com que sejam usadas de forma muito destinta. O Procedure é invocado através do comando *CALL PROCEDURE*, em quanto a Function, devido fato de retornar um valor pode ser usada numa query SQL.

```
mysql> delimiter //

mysql> CREATE FUNCTION hello (s CHAR(20)) RETURNS CHAR(50)
    -> RETURN CONCAT('Hello, ',s,'!');
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;

mysql> SELECT hello('world');
+----------------+
| hello('world') |
+----------------+
| Hello, world!  |
+----------------+
1 row in set (0.00 sec)
```

7.3 Trigger

Os triggers são suportados em MySQL, sedo possível fazer *CREATE/ DROP TRIGGER*.

Sintaxe:

```
CREATE TRIGGER
CREATE
    [DEFINER = { user | CURRENT_USER }]
    TRIGGER trigger_name
```

```
  trigger_time trigger_event
  ON tbl_name FOR EACH ROW
  [trigger_order]
  trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger_name

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

7.4 Conectores e APIs

Os MySQL *Connecters* fornecem conectividade do MySQL *Server* a programas clientes. APIs fornecem acesso de baixo nível ao protocolo e recursos do MySQL. Ambos os conectores e as APIs permitem conectar e executar instruções do MySQL a partir de outra linguagem ou ambiente, incluindo ODBC, Java (JDBC), Perl, Python, PHP, Ruby , C e instâncias de *embedded* MySQL.

7.5 MySQL Access Privilege System (APS)

A principal função do sistema de privilégios do MySQL é autenticar um utilizador que se conecta de um determinado host e associar aquele utilizador a uma BD com privilégios de SELECT, INSERT, UPDATE e DELETE.

MySQL fornece privilégios que se aplicam em diferentes contextos e em diferentes níveis de operação:

Privilégios administrativos permitem aos utilizadores gerenciar a operação do MySQL *server*. Estes privilégios são globais, porque eles não são específicos de uma BD em particular.

Privilégios de base de dados aplicam-se a uma BD e todos os objetos dentro dela. Estes privilégios podem ser concedidos para BD específicas, ou a nível global, de modo que eles se aplicam a todas as BDs.

Privilégios para objetos de base de dados, tais como *tables*, *indexes*, *views*, e *stored routines* podem ser concedidos para objetos específicos dentro de uma BD, para todos os objetos de um determinado tipo dentro de uma BD (por exemplo, todas as tabelas em uma BD), ou globalmente para todos objetos de um determinado tipo em todas as BSs.

7.6 Tipos de dados

MySQL vs Oracle SQL

MySQL	**Oracle SQL**
BIGINT	NUMBER(19, 0)
BIT	RAW
BLOB	BLOB, RAW
CHAR	CHAR
DATE	DATE
DATETIME	DATE
DECIMAL	FLOAT (24)
DOUBLE	FLOAT (24)
DOUBLE PRECISION	FLOAT (24)
ENUM	VARCHAR2
FLOAT	FLOAT
INT	NUMBER(10, 0)
INTEGER	NUMBER(10, 0)
LONGBLOB	BLOB, RAW
LONGTEXT	CLOB, RAW
MEDIUMBLOB	BLOB, RAW
MEDIUMINT	NUMBER(7, 0)
MEDIUMTEXT	CLOB, RAW
NUMERIC	NUMBER

MySQL	Oracle SQL
REAL	FLOAT (24)
SET	VARCHAR2
SMALLINT	NUMBER(5, 0)
TEXT	VARCHAR2, CLOB
TIME	DATE
TIMESTAMP	DATE
TINYBLOB	RAW
TINYINT	NUMBER(3, 0)
TINYTEXT	VARCHAR2
VARCHAR	VARCHAR2, CLOB
YEAR	NUMBER

7.7 Ferramentas Workbench

MySQL Workbench fornece uma ferramenta gráfica para trabalhar com servidores e bases de dados MySQL. MySQL Workbench suporta totalmente as versões do MySQL Server 5.1 e acima. Também é compatível com o MySQL Server 5.0, mas não todos os recursos do 5.0 podem ser suportadas. Ele não suporta as versões do servidor MySQL 4.x.

MySQL Workbench oferece funcionalidades em cinco áreas principais:

Desenvolvimento SQL: Permite criar e gerenciar conexões com servidores de base de dados. Além de permitir que configure os parâmetros de conexão, MySQL Workbench oferece a capacidade de executar consultas SQL sobre as conexões de base de dados usando o built-in SQL Editor.

Modelação de Dados: Permite criar graficamente modelos de um esquema de base de dados, *reverse* e *forward engineer* entre um esquema e uma *live* BD, e editar todos os aspectos da BD usando o editor de tabela completa. O Editor de Tabelas fornece instalações de fácil utilização para

edição de Tabelas, Colunas, Índices, Triggers, Particionamento, Opções, Inserções, Privilégios, Rotinas e Views.

Administração de Servidor: permite criar e administrar instâncias do servidor.

Migração de Dados: Permite migrar do Microsoft SQL Server, Sybase ASE, SQLite, SQL Anywhere, PostreSQL, e outras tabelas de RDBMS, objetos e dados para MySQLEsta funcionalidade também suporta a migração de versões anteriores do MySQL para a ultima versão.

8 Conclusão

A realização deste relatório permitiu-nos conhecer um sistema de base de dados que não nos era familiar, e as suas funcionalidades e limitações. Sentimos que no momento de selecionar um sistema de base de dados para um novo projeto, estaremos mais cientes dos pontos mais relevantes a considerar, e do que o MySQL oferece nesses pontos. Sendo um dos grandes desafios dum engenheiro informático a seleção das tecnologias a usar num novo projeto, pensamos que este trabalho foi uma experiência particularmente enriquecedora, sobre aquele que é o sistema de base de dados open source mais usado (MySQL), e a sua comparação com o sistema de base de dados pago mais usado (Oracle).

9 Referencias

[1] Documentação Oficial: http://dev.mysql.com/doc/refman/5.7/en/.