Query Processing and Optimization in Different Databases

1st Luís Pinto
Departamento de Engenharia Informática
ISEP
Porto, Portugal
1190818@isep.ipp.pt

2nd João Henriques
Departamento de Engenharia Informática
ISEP
Porto, Portugal
1231925@isep.ipp.pt

3rd André Teixeira
Departamento de Engenharia Informática
ISEP
Porto, Portugal
1190384@isep.ipp.pt

***Abstract*— Este artigo científico tem como principal objetivo documentar o trabalho realizado no âmbito da unidade curricular de Tópicos Avançados de Bases de Dados, integrada no mestrado em Engenharia de Dados do Instituto Politécnico do Porto. Serão abordados os temas de Processamento de *Queries* e Otimização de *Queries*, bem como alguns conceitos básicos de base de dados.**

***Keywords*— Base de dados, SQL, NoSQL, Query, Oracle, MongoDB, PostgreSQL, MySQL, Otimização, Índice, Operador**

I. Introdução

A eficiência na obtenção de informações através de *queries* representa um elemento crucial em qualquer sistema de base de dados.

O processo de "*Query Processing and Optimization*" desempenha um papel fundamental nesse contexto, influenciando diretamente a capacidade de uma base de dados atender às solicitações de maneira eficaz.

Ao explorarmos esse tema, concentraremos a nossa análise em quatro notáveis softwares de base de dados: *Oracle*, *MongoDB*, *PostgreSQL* e *MySQL*.

Cada um desses sistemas possui abordagens distintas no processamento e otimização de *queries*, refletindo nuances específicas que impactam diretamente o desempenho e a eficácia na recuperação de dados.

Neste contexto, exploraremos as características particulares de cada base de dados, destacando as estratégias, algoritmos e melhores práticas associadas ao Processamento e Otimização de *Queries* do utilizador em *Oracle*, *MongoDB*, *PostgreSQL* e *MySQL*.

II. Conceitos Teóricos

A. Query

Uma pergunta feita a uma base de dados é chamada de "*query*". Simplesmente, é um comando para inserir, atualizar, excluir ou recuperar dados de uma base de dados. Normalmente, as *queries* são escritas numa linguagem de consulta específica, como *SQL* (*Structured Query Language*), que é amplamente utilizada em bases de dados relacionais.

Cada um dos vários tipos de *queries* realiza uma função específica na base de dados. A seguir estão alguns exemplos comuns:

- *SELECT*: Usado para recuperar dados de uma ou mais tabelas na base de dados.

  Exemplo:

```
SELECT nome, idade FROM CLIENTES WHERE cidade = 'Porto';
```

Figura 1 - Exemplo Select

- *INSERT*: Utilizado para adicionar novos registos a uma tabela.
  Exemplo:

```
INSERT INTO produtos (nome, preco) VALUES ('Relógio', 100.00);
```

Figura 2 - Exemplo Insert

- *UPDATE*: Usado para modificar dados existentes numa tabela.
  Exemplo:

```
UPDATE FUNCIONARIOS SET salario = 50000 WHERE cargo = 'Gerente';
```

Figura 3 – Exemplo Update

- *DELETE*: Utilizado para excluir registos de uma tabela.
  Exemplo:

```
DELETE FROM CLIENTES WHERE idade < 18;
```

Figura 4 - Exemplo Delete

Estas são apenas as operações básicas, mas dependendo das necessidades específicas, as *queries* podem ser mais complexas, incluindo junções de tabelas, subconsultas e outras características avançadas. O objetivo é interagir com a base de dados de maneira eficaz e obter todos os dados necessários.

B. Query Optimization

Query Optimization refere-se ao processo de otimização e redução de tempo de execução de uma query. Isto pode ser conseguido pela seleção de um plano de execução que otimize o custo e tempo de uma determinada query.

Com esta implementação é possível aceder à base de dados de uma maneira eficiente. Formalmente pode-se definir como a maneira de transformar uma *query* numa

query equivalente, mas que pode ser avaliada mais eficientemente.

É, portanto, importante que este processo seja utilizado da maneira mais eficiente possível, com a seleção do plano mais eficiente e redução do tempo de execução da *query*.

C. Query Processing

Query Processing traduz-se nas atividades que incluam a tradução de *queries* de Linguagens de Alto Nível em operações no nível físico do sistema e otimização, transformação e avaliação de *queries*. Pode-se observar um exemplo de estrutura dos passos existentes no processamento de *queries* na seguinte ilustração:

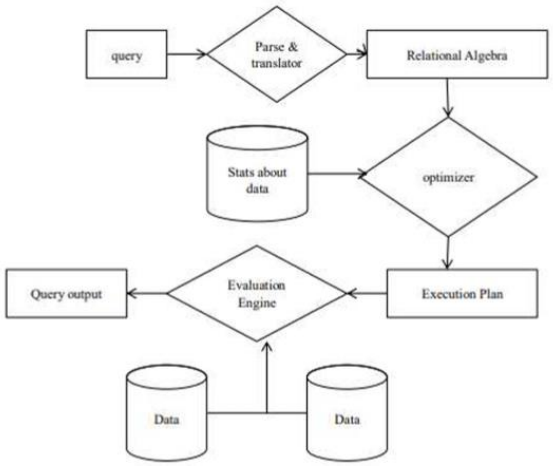

Figura 5 - Steps in Query Processing

D. SQL

Sigla para *Structured Query Language*, é uma linguagem desenvolvida para ajudar na gestão de bases de dados do tipo relacional. Segue uma estrutura tabular e um esquema rígido.

E. NoSQL

Not Only SQL é o termo utilizado para denominar bases de dados do tipo não relacional. Estas podem adotar modelagens diferentes das tradicionais tabulares.

III. Query Processing and optimization

A. Oracle

Oracle é uma das bases de dados mais utilizadas e na qual a otimização de *queries* é crucial para a sua performance. Esta base de dados possui uma variedade de técnicas de otimização de *queries* que proporciona um melhor desempenho e performance.

1) Técnicas

Algumas dessas técnicas passam por:

Começar com o *tuning* do *System Level Sql*, isto é importante ser realizado em primeiro lugar para que as alterações futuras não sejam revertidas automaticamente;

Reescrever *subqueries* complexas com tabelas temporárias, em Oracle podem-se utilizar *Global Temporary Tables* (GTT's) e operadores como o *SQL WITH*. Isto ajuda na simplificação de *sub-queries* complexas, tais como aquelas que fazem uso de clausulas como *WHERE* ou *SELECT*. Utilizando *GTT's* ou clausulas *WITH* consegue-se um aumento significativo na performance das *queries*;

Indexar os predicados. Para clausulas como *JOIN*, *WHERE*, *ORDER BY* e *GROUP BY*, é importante a indexação dos seus predicados, pois sem o correto indexamento serão efetuadas consultas e análises às tabelas desnecessariamente, causando problemas como a redução da performance e *locking issues*. É, portanto, de extrema importância a indexação de todos os predicados, exceto no caso de a coluna em questão ter uma baixa cardinalidade;

Utilizar *Inner Joins* no lugar de *Outer Joins*. Devem-se utilizar *Outer Joins* apenas quando estritamente necessário. Se este não for o caso, os *Inner Joins* serão sempre a melhor opção para que se obtenha uma melhor performance das *queries*. *Outer Joins* provocam um abrandamento na execução das *queries* o que acaba por afetar a sua performance, portanto, o uso de *Inner Joins* é recomendado pelo que resulta numa melhor performance das *queries*;

Utilizar colunas *CLOB*/*BLOB* no final das instruções *SQL*. Na versão 10g e 11g das bases de dados *Oracle* é necessário o uso destas colunas *CLOB*/*BLOB* no final das instruções. A falta das mesmas resulta em falhas de execução se o valor do input for mais elevado do que mil caracteres.

2) Query Optimizer

O *Oracle* conta com um *Query Optimizer* que ajuda no processo de otimização das suas *queries*.

Este componente é essencial para o processamento de todas as instruções *SQL*, bem como na determinação do melhor e mais eficiente plano de execução para cada instrução. O plano é definido para cada instrução tendo em mente a estrutura da *query*, a informação disponível quanto aos *underlying objects* e um conjunto de capacidades que permitem o *Optimizer* na execução de ajustes em *run-time* dos seus planos de execução para que este encontre informações adicionais, que, por sua vez, irão contribuir para a obtenção de melhores estatísticas. A este conjunto de capacidades atribui-se o nome de *Adaptive Query Optimization*.

Este tipo de implementação torna-se bastante útil quando as estatísticas presentes não são suficientes para a elaboração de um bom plano. Existem dois aspetos distintos que fazem parte do *Adaptive Query Optimization*: *Adaptive Plans*, que são responsáveis por melhorar a execução inicial de uma *query* e *Adaptive Statistics*, que por sua vez se responsabilizam pela contribuição de informações adicionais com o intuito de melhorar as próximas execuções.

O mecanismo de elaboração de planos denomina-se de *SQL Plan Management* e permite ao *Optimizer* o manuseamento dos planos de forma a assegurar que a base de dados utilize apenas os planos conhecidos ou verificados. Os principais objetivos do *Optimizer* dentro deste *SQL Plan Management* passam por identificar instruções *SQL* repetidas, manter um histórico de planos e *SQL plan baselines* para um conjunto de instruções *SQL*, detetar planos que não constem no histórico e detetar planos que sejam potencialmente melhores e não estejam inclusos dentro do *SQL plan pipeline*.

Na ilustração da Figura 1, podem-se observar os componentes que fazem parte do *Adaptive Query Optimization*, sub-*dividido* em *Adaptive Plans* e *Adaptive Statistics*.

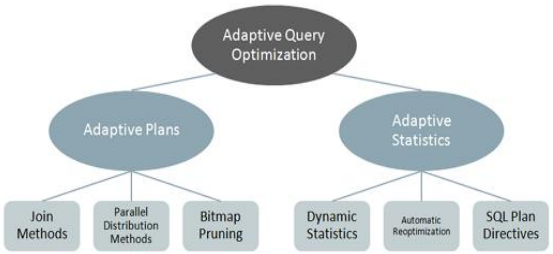

Figura 6 - Adaptive Query Optimization Schema

3) Adaptive Plans

Um *adaptive plan* vai ser escolhido pelo *Optimizer* consoante as condições requeridas, como por exemplo quando existe uma *query* com *JOIN's* e predicados complexos que dificultam a estimativa da sua cardinalidade de forma apurada. Estes *Adaptive Plans* permitem ao *Optimizer* adiar a sua decisão quanto a plano a implementar até ao início do tempo de execução. O plano é inicializado pelo *Optimizer* com *statistics collectores* para que sejam estimadas, no caso de existirem, alterações quanto à cardinalidade do número de *rows* observadas pelas operações do plano. Assim, caso existam num número de grande significância, o plano ou parte deste será automaticamente adaptado para evitar esta perda de performance.

Podem-se subdividir os *Adaptive Plans* em três diferentes noções:

Adaptive Join Methods - O *Optimizer* é capaz de adaptar métodos de *Join* em qualquer momento porque este predetermina múltiplos *sub-planos* para partes do plano.

Por exemplo, na Figura 2 o plano *default* do nosso *Optimizer* é um *nested loop Join* através do *index* da tabela de *products*. Uma alternativa a esta implementação também foi determinada em forma de *sub-plan*. Esta alternativa permite ao *Optimizer* mudar o *Join* por um *Hash Join*, sendo

que neste sub-plano a tabela de *products* é acessada por um *scan* completo à tabela.

Perante isto, na execução inicial, os *statistic collectors* recolhem informação acerca da execução e envia para um buffer uma pequena porção das *rows* que irão para o sub-plano. O *Optimizer* então determina quais as estatísticas a serem recolhidas e como será executado o plano consoante os diferentes valores obtidos. Para um caso em que ambas as implementações do plano são equivalentes, ou seja, o valor obtido pelas estatísticas é o mesmo, é computacionado um "*inflection point*" que armazena este valor. Por exemplo, se o plano dos *nested loops join* é melhor quando uma tabela de *orders* produz menos de 10 *rows* e o plano de *hash joins* quando ela produz mais de 10 *rows*, podemos inferir que o "*inflection point*" será, portanto, 10. O *Optimizer* computa esta valor e configura um *statistics collector buffer* que vai contar até 10 *rows*. Se pelo menos 10 *rows* são contadas pelo *buffer*, então é implementado o plano do método *Join*, se isto não acontecer é implementado o plano do *hash join*. Baseada na informação adquirida pelo *statistics collector*, o *Optimizer* vai tomar a sua decisão acerca de qual o sub-plano a implementar. Neste caso, como o número de *rows* excedeu a estimativa por parte do *Optimizer*, será implementado o *sub-plano* de *Hash Join*.

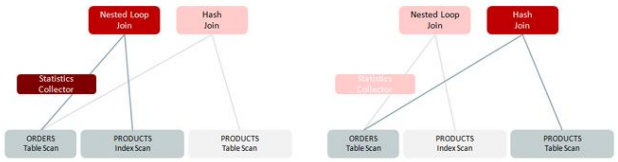

Figura 7 -Adaptive Join Methods Example

Adaptive Parallel Distribution Methods - Quando um *SQL statement* é executado em paralelo, certas operações como *sorts*, *aggregations* e *joins* precisam que estes dados sejam distribuídos pelos processos paralelos que executam o *statement*. O método de distribuição escolhido por parte do *Optimizer* deve ter em conta os seguintes aspetos: o tipo de operação, o número de processos paralelos envolvidos e o número de *rows*. Caso o número de *rows* estimado seja errado podem surgir problemas na seleção do método de distribuição contribuindo para uma performance sub-optimal e o pouco aproveitamento ou inutilização de alguns processos paralelos.

Para resolver estes problemas desenvolveu-se um novo método de distribuição adaptativo, o *HYBRID HASH*. Ele funciona da seguinte forma: Primeiro insere-se um *statistics collector* antes da operação ser efetuada e é estimado o número de *rows*, se o número de *rows* for inferior ao *threshold* do *buffer*, o método de distribuição muda de *HASH* para *BROADCAST*, caso contrário, se este atingir o *threshold* o método a ser utilizado é o *HASH*.

Adaptive Bitmap Index Pruning - Quando o plano é gerado, é necessária a escolha correta de bitmap indexes para reduzir os conjuntos de *ROWIDs* relevantes. Se existirem bastantes *Id´s*, alguns podem não reduzir substancialmente o número de *ROWIDs*, no entanto contam consideravelmente para o custo do processamento da *query*. Para que isso não aconteça, os *Adaptive Plans* realizam então um *pruning* para remover *Id´s* não relevantes e que não filtrem significantemente o número de *rows*.

4) Adaptive Statistics

Adaptive Statistics são fundamentais para a qualidade do plano escolhido pelo Optimizer, quanto melhor a qualidade das estatísticas, melhor a qualidade do plano.

No entanto, alguns predicados de *queries* são demasiado complexos para se apoiarem apenas em estatísticas de tabelas, para este propósito foram criadas as *adaptive statistics*. Estas podem ser decompostas pelos seguintes elementos:

Dynamic Statistics - Durante a compilação de um *statement* de *SQL* é avaliada a qualidade/quantidade das estatísticas, caso estas não sejam suficientes para a execução de um bom plano é considerada a utilização de *dynamic samples*. *Dynamic sampling* consiste na obtenção de estatísticas básicas em tabelas onde não existam estatísticas. No surgimento da versão 12c da *Oracle Database*, esta *Dynamic Sampling* convergeu para uma *Dynamic Statistics*. *Dynamic Statistics* permite o *Optimizer* obter pelo processo de *augmentation*, estatísticas com uma cardinalidade mais apurada, estimando não apenas isto em acessos a tabelas, mas como também em *joins* e *group-by predicates*.

Dynamic Statistics também permitem ao *Optimizer* decidir se deve usá-las ou não mesmo para tabelas onde já tenha informações quanto às estatísticas. Isto é decidido tendo em conta a complexidade dos predicados, as estatísticas bases existentes e o tempo de execução esperado do *SQL statement*.

Automatic Re-optimization - Durante a compilação da primeira execução do *SQL statement* um plano é gerado como habitual. Durante a otimização, certos tipos de estimativas são considerados de baixa qualidade, como é o caso de tabelas com poucos dados de estatística ou predicados complexos. Esta informação é guardada e a monitorização do plano escolhido começa. Se as estimativas quanto à cardinalidade diferirem do esperado, é necessário procurar outro plano, o *Optimizer* vai então utilizar a informação guardada na execução anterior para conseguir determinar um novo plano de execução. Isto é chamado de *Re-optimization* e pode ser utilizada várias vezes para uma dada *query*, melhorando com cada otimização o plano de execução.

SQL Plan Directives - *SQL Plan Directives* são *directives* que contêm informação adicional utilizada pelo *Optimizer* para gerar um plano de execução melhor e são automaticamente criados com base na informação adquirida pela *Automatic Re-Optimization*. Estes não são criados especificamente para *SQL statements* mas sim para *query expressions* utilizadas nestes *SQL statements*.

5) Concluindo

O *Optimizer* é considerado um dos componentes mais úteis do *Oracle Database* devido à sua complexidade. O seu propósito é determinar o plano de execução mais eficiente para cada declaração *SQL*. Ele toma essas decisões com base na estrutura da *query*, nas estatísticas disponíveis sobre os dados e em todos os recursos relevantes de otimização e execução. No *Oracle Database 18c*, o *Optimizer* dá um salto gigante com a introdução de uma nova abordagem adaptativa para otimização de consultas e melhorias nas informações estatísticas disponíveis para o mesmo. A nova abordagem adaptativa para otimização de consultas permite que o *Optimizer* faça ajustes em tempo de execução nos planos de execução e descubra informações adicionais que podem levar a estatísticas mais precisas. Aproveitar essas informações em conjunto com as estatísticas existentes torna o *Optimizer* mais envolvido no seu *environment* e permite que ele selecione um plano de execução ideal em todas as situações.

B. *MongoDB*

O *MongoDB* é uma base de dados do estilo *NoSQL*. Ao contrário das bases de dados relacionais tradicionais, o *Mongo* armazena os dados em documentos semelhantes a *JSON*, o que lhe permite lidar com um elevado número de dados de forma dinâmica e fácil de escalar.

Alguns dos benefícios de adotar *MongoDB* passam por:

1. Não ter um *schema* definido: não requer um *schema* pré-definido para armazenar os dados. Isto facilita a escalabilidade da base de dados pois permite os utilizadores criarem o número de campos que acharem necessários;

2. Orientado a documentos: estes documentos podem ser facilmente mapeados para tipos de dados nativos das diversas linguagens de programação;

3. Escalabilidade horizontal: permite distribuir os dados em diferentes servidores. Oferece uma melhor resposta a grandes volumes de dados e a tráfego intenso;

4. Alta performance: utiliza índices eficientes e oferece operações de leitura e gravação rápida e eficiente;

5.Consultas avançadas: suporta consultas complexas, nomeadamente agregações.

Por outro lado, existem as desvantagens:

1. Continuidade: o seu sistema automático de *failover* pode afetar a continuidade. Este processo garante a continuidade, mas pode demorar algum tempo a ser executado, não é instantâneo;

2. Consistência dos dados: as restrições de uso de chaves estrangeiras podem afetar a consistência dos dados;

3. Segurança: a autenticação do utilizador, por omissão, está desativada. Contudo, recentemente foram adicionadas definições que bloqueiam conexões que não sejam configuradas pelo administrador.

A otimização nas consultas no *MongoDB* é importante para garantir o desempenho eficiente em operações de leitura e escrita. A seguir são descritas algumas das estratégias que contribuem para esta otimização:

1) *Índices*

A criação de índices adequados pode aumentar a eficiência destas operações limitando a quantidade de dados que a *query* necessita de processar. Ao criar um índice para um determinado campo, previne-se que a *query* tenha de consultar toda a *collection* para conseguir encontrar e retornar os resultados. Para além de ajudarem na otimização, os índices suportam operações do tipo *sort* e contribuem para uma utilização de memória mais eficiente.

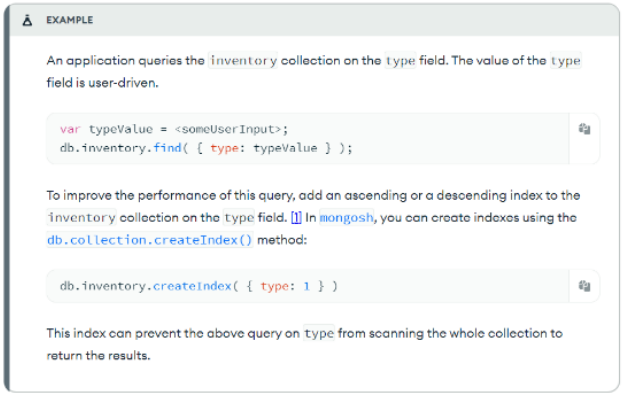

Figura 8 - Exemplo de aplicação de índices em MongoDB

Os índices podem tomar diferentes formas de modo a responder a uma gama diversificada de necessidades. Existem os índices simples que são mais eficazes nas consultas direcionadas a um campo, mas caso necessário, também existem os complexos que permitem considerar múltiplos campos simultaneamente. Os índices de texto facilitam as buscas de texto em dados não estruturados. Já os geoespaciais permitem consultas baseadas em localização, facilitam análises de proximidade geográfica. Existem ainda os índices *hashed* que são rápidos em comparação de valores em igualdades. São vários os tipos de índices e podem ser consultados na *wiki* da página oficial do *MongoDB*.

2) *Operadores*

Os operadores de consulta devem ser escolhidos dentro dos mais eficientes, tendo em conta a necessidade específica da consulta. Evitar ao máximo os operadores não muito seletivos.

Tal como nos índices, os operadores podem ser agrupados em diferentes tipos. Desde os de comparação, lógicos ou até de elemento, os operadores são ferramentas importantes para a construção de consultas precisas e eficientes. Na tabela seguinte é possível verificar alguns exemplos de operadores, bem como a sua função:

Operator	Description	Syntax
$eq	This operator compares values that are the same as the specified value.	db.inventory.find({ "Pid": { $eq: "PID0001"}})
$gt	This operator returns data if the value is greater than the specified value.	db.inventory.find({ "quantity": { $gt: 5700}})
$lt	This operator returns data if the value is less than the specified value.	db.inventory.find({ "quantity": { $lt: 5700}})
$gte	This operator returns data if the value is greater than or equal to the specified value.	db.inventory.find({ "quantity": { $gte: 5700}})
$lte	This operator returns data if the value is less than or equal to the specified value.	db.inventory.find({ "quantity": { $lte: 5700}})
$in	It matches values present in the array.	db.inventory.find({ "price": { $in: [3, 6]}})
$ne	This operator returns data that is not equal to the given value.	db.inventory.find({ "price": { $ne: 4.59}})
$nin	This operator returns data that does not match any of the values in an array.	db.inventory.find({ "price": { $nin: [4.29, 3, 6, 2.15, 4.95]}})

Figura 9 - Lista de operadores disponibilizados pelo MongoDb

3) Agregações

Utilizar o *Aggregation Framework* para realizar as operações mais complexas e maneira otimizada. Esta ferramenta opera através de um conjunto de estágios que são aplicados de forma sequencial aos documentos. Cada um destes estágios, é responsável por transformar o conjunto de dados de acordo com a necessidade da consulta. Alguns dos estágios mais comuns são:

1. *$match*: responsável por filtrar os documentos com base em critérios específicos, semelhante a uma operação "find";

2. *$project*: permite selecionar os campos que se pretende incluir no resultado. Pode ainda criar campos calculados ou modificar o formato de campos existentes;

3. *$group:* agrupa os documentos com ajuda de operações como soma, máximo, mínimo, média, entre outras;

4. *$sort:* tal como o nome indica, ordena os campos de forma ascendente ou descendente de acordo com o pedido pelo utilizador.

Existem uma vasta lista de outros estágios que podem ser aplicados tendo em conta a pesquisa que se pretende fazer. Caso necessário, pode-se consultar a página oficial do *MongoDB*.

Todos os estágios são combinados numa *pipeline* onde o *output* de um estágio é o *input* do seguinte. Ao organizar estes estágios de forma correta e sequencial, o utilizador pode fazer uma manipulação complexa dos documentos mesmo entre coleções diferentes.

4) Limitar e Paginar Resultados

Quando se expecta uma elevada quantidade de resultados, deve se ter o cuidado de limitar o número de documentos retornados. Limitar os resultados reduz a carga de trabalho do servidor, mas também agiliza o tempo de resposta por consulta, ou seja, torna o processo mais eficiente.

Para conjuntos de dados ainda mais extensos, pode mesmo ser necessário aplicar paginação. Através dos parâmetros *skip* e *limit* é possível apresentar o conjunto de dados de resultado conforme o necessário.

5) Comando explain( )

Este comando fornece informações relativas ao plano de execução, estatísticas de desempenho, uso de índices e estratégias de otimização.

Ao utilizar este comando, o utilizador pode identificar áreas de melhoria em consultas que possam melhoras a eficiência. A avaliação dos índices mostra se estes estão a ser utilizados de maneira eficaz.

C. PostgreSQL

O *PostgreSQL*, também conhecido como *Postgres*, é um poderoso sistema de gestão de base de dados relacional. Destaca-se pela sua confiabilidade, extensibilidade e conformidade com os padrões *SQL* que o tornam único. O *PostgreSQL*, originalmente criado pela Universidade da Califórnia, tem uma comunidade de desenvolvedores ativa e um longo histórico de desenvolvimento.

Uma das características mais notáveis do *PostgreSQL* é a sua capacidade de suportar uma ampla gama de tipos de dados, incluindo tipos personalizados, bem como as suas poderosas capacidades de extensão. Além disso, o *PostgreSQL* suporta vários recursos avançados de base de dados, incluindo visões materializadas, procedimentos armazenados e visões.

A arquitetura robusta do *PostgreSQL* torna-o adequado para uma ampla gama de aplicações, desde aplicações web básicas até sistemas empresariais mais complexos. Os seus compromissos com padrões abertos, juntamente com a capacidade de personalização e otimização avançada de consultas, fazem dele uma escolha popular para organizações que procuram uma base de dados relacional confiável e flexível.

PostgreSQL usa um processo de *back-end* que lida com todas as consultas enviadas pelos clientes conectados. Os cinco subsistemas que compõem esse processo de *back-end* podem ser vistos na imagem seguinte:

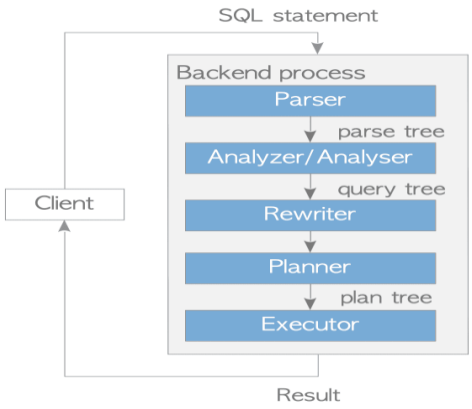

Figura 10 - Processo Backend PostgreSQL

1) Parser

Inicialmente e a partir de uma instrução *SQL* de texto simples, feita pelo cliente, o analisador cria uma árvore de *parse* que pode ser lida pelos sistemas subsequentes.

O analisador apenas verifica a sintaxe de uma entrada ao gerar uma árvore de *parse*. Portanto, só retorna um erro se houver um erro de sintaxe na *query*.
Para além disso, também não verifica a semântica de uma *query*. Por exemplo, mesmo se a consulta contiver um nome de uma tabela que não exista, o "*parser*" não retornará um erro. As verificações semânticas são realizadas pelo "*analyzer/analyser*".

2) Analyzer/Analyser

O *analyzer/analyser* examina a árvore de parse e cria uma árvore de consulta.

3) Rewriter

Se existirem regras, o *rewriter* transforma uma árvore de consulta usando as regras armazenadas no sistema de regras.

4) Planner

O "*planner*" recebe a árvore de consulta e cria árvore de plano, que pode ser executada com mais eficiência no "executor". É o subsistema mais complexo no *PostgreSQL*.

5) Executor

O executor executa a consulta acedendo às tabelas e índices na ordem que foi criada pela árvore de plano.

Embora o *PostgreSQL* seja feito para funcionar com grande eficiência, o desempenho das consultas ainda pode ser melhorado otimizando-as. Aqui estão algumas recomendações para melhorar a otimização das consultas no *PostgreSQL*:

1) Utilizar índices

Tendo em consideração que temos uma tabela chamada "funcionarios" com as colunas "idFuncionario", "nome" e "*e-mail*". Se realizarmos procuras frequentes pelos funcionários através do seu *id*, podemos criar um índice na coluna "idFuncionario" da seguinte forma:

```
CREATE INDEX funcionarios_idFuncionarios_idx ON funcionarios (idFuncionarios);
```

Figura 11 - Create Index Example

2) Utilizar o comando explain

O comando *EXPLAIN* fornecerá o plano de execução para a consulta, o que pode ajudar a identificar quaisquer problemas de desempenho.

3) Evitar a utilização de caracteres especiais

O uso de caracteres especiais no início de uma *string* dificulta o uso de índices pela base de dados. Embora os

índices sejam estruturas otimizadas para procura rápida, a base de dados precisa frequentemente de examinar todas as entradas para encontrar as correspondências quando os caracteres especiais estão no início.
Para além disso, a otimização de *queries* com caracteres especiais pode ser mais difícil porque o optimizador de *queries* pode ter menos opções para escolher os melhores planos de execução. Mesmo após esforços de otimização, isso pode resultar num pior desempenho.

4) Limitar o número de linhas retornadas

Tendo em conta que temos uma *query* em que é retornada todos os funcionários da tabela "funcionários":

```
SELECT * FROM funcionarios;
```

Figura 12 - Select "funcionarios"

Se a tabela for grande, esta consulta pode ser lenta e exigir muitos recursos. Assim, utilizando o comando *LIMIT*, podemos limitar o número de linhas retornadas para esta consulta:

```
SELECT * FROM funcionarios LIMIT 100;
```

Figura 13 - Select "funcionarios" limit 100

Esta consulta retornará apenas as primeiras 100 linhas, o que pode melhorar o desempenho.

5) Utilizar tipos de dados corretamente

Na tabela "funcionarios" há uma coluna chamada "idade" que contem a idade de cada funcionário como um número inteiro. Se calcularmos regularmente a média das idades dos funcionários, podemos otimizar o desempenho utilizando um tipo de dados menor, como um "*smallint*" em vez de um "*integer*":

Isto pode melhorar o desempenho, pois diminuirá a quantidade de memória necessária para armazenar a coluna "idade".

D. MySQL

O *MySQL* é um dos sistemas de gestão de base de dados relacional mais utilizado e reconhecido. Desde a sua criação na década de 90, que tem ganhado popularidade devido a sua robustez, confiabilidade e flexibilidade. Como uma base de dados do estilo relacional, utiliza tabelas para organizar e armazenar os dados, ou seja, segue um esquema predefinido que define a sua estrutura, relações e restrições.

Alguns dos benefícios de adotar *MySQL* passam por:

1. Oferece alguma flexibilidade em relação à definição de esquemas. Permite algumas adaptações e modificações.

2. Em ambientes com grandes volumes de dados, oferece operações de leitura e gravação rápidas, conseguindo manter um alto desempenho.

3. Permite a escalabilidade horizontal e vertical, influenciando a distribuição dos dados e garantindo uma resposta confiável.

Por outro lado, existem as desvantagens:

1. Manutenção: o desempenho pode diminuir com um grande volume de dados. Consultas complexas ou estruturas de tabelas mal projetadas podem influenciar negativamente este desempenho;

2. Rigidez: a alteração dos esquemas já existentes pode ser complexa e limitada;

3. Tipos de dados: alguns dados mais específicos podem ter suporte limitado;

De forma a mitigar parte destas desvantagens, o *MySQL* apresenta também algumas estratégias de otimização que serão discutidas de seguida:

1) Índices

À semelhança de outras ferramentas demonstradas anteriormente, aqui os índices também têm um papel fundamental na otimização do desempenho de consultas.

Também, à semelhança das ferramentas anteriores, existem diferentes tipos de índice que podem ser escolhidos pelo utilizador de forma a responder melhor às suas necessidades. Os índices simples são criados numa única coluna e são eficientes para consultas a campos específicos, como por exemplo para o comando *WHERE*. Já os compostos, criados em várias colunas, são úteis para consultas que envolvam múltiplas colunas. Os espaciais permitem fazer consultas baseadas em localização, úteis sobretudo para aplicações que lidam com informações geográficas. Outro exemplo, os índices de texto, que permitem as pesquisas de texto em colunas de texto longo.

Então e qual a importância de adotar índices e o seu impacto nas consultas? Os índices ajudam o *MySQL* a encontrar os registos mais rapidamente diminuindo a quantidade de dados que precisam de ser percorridos durante a execução da consulta. Com este processo mais agilizado, a carga no servidor também é reduzida significativamente, permitindo respostas mais rápidas e melhor escalabilidade.

Para utilizar os índices de forma eficiente, existem algumas estratégias que se devem adotar, nomeadamente, identificar as colunas chave que possam ser utilizadas em cláusulas *WHERE, JOIN* e *ORDER BY*. Apesar de trazerem vantagens, os índices também não devem ser criados em demasia, devem-se evitar os desnecessários. Deve-se também monitorizar regularmente os índices de forma a garantir que permanecem otimizados.

2) Otimizar operações SELECT

Tentar evitar ao máximo utilizar o comando "*SELECT**" ou seja, tentar fazer uma seleção dos dados de forma a reduzir a quantidade de dados transferidos e incrementar a performance da base de dados.

Utilizar outras funções de agregação, nomeadamente *SUM, COUNT* ou *AVG*, seletivamente também pode diminuir o processamento de dados e incrementar a performance.

3) Comando EXPLAIN

O comando *EXPLAIN* explica como o *MySQL* pretende executar a *query*, incluindo os índices escolhidos e a ordem de acesso à tabela. Este comando permite analisar o plano de execução e identificar potencias problemas que afetem a performance da consulta.

4) Limitar a quantidade de dados por consulta

Usar o comando *LIMIT* para limitar o número de linhas que a consulta irá retornar, especialmente em consultas que retornam um grande número de dados.

Caso necessário, também deve ser implementada paginação para consultar os dados, mas em porções mais pequenas, de modo a reduzir a carga e o tempo de resposta do servidor. A paginação pode ser construída através do comando *OFFSET*.

5) Utilizar joins e evitar subqueries desnecessárias

A utilização de *JOIN*s deve ser apropriada tendo em conta a relação entre as tabelas e o resultado desejado. Tentar evitar a utilização de *subqueries* e caso possível substituir estas por *JOIN*s, pois são mais eficientes.

6) Cache de consultas

Ao implementar um sistema de cache que armazene resultados frequentes de consultas, o peso de processamento da base de dados é reduzido e o tempo de respostas é acelerado.

7) Manutenção regular

Realizar manutenções periódicas, como otimização de tabelas, atualização de estatísticas ou limpeza de dados ultrapassados, também pode contribuir para um melhoramento significativo do desempenho da base de dados.

IV. Conclusões

Ao longo do estudo sobre a otimização de consultas em diferentes bases de dados, foi possível averiguar quatro ferramentas diferentes, nomeadamente, *Oracle*, *MongoDB*, *PostgreSQL* e *MySQL*.

Estas bases de dados, tanto do tipo *SQL* ou *NoSQL*, apresentam as diferenças em termos de estrutura e a forma de como fazem as suas consultas. O utilizador pode escolher aquela que mais se adequa a si.

Contudo, podemos verificar que alguns dos aspetos de otimização são comuns às diferentes bases de dados. Com maior enfâse na implementação de índices, conseguimos concluir que apesar das diferenças nas implementações e afins, o conceito converge na mesma função e otimização do processo.

Outros conceitos como perceber os padrões de acesso aos dados e realizar as consultas de forma mais eficiente possível também são adjacentes a todas as bases de dados estudadas.

Por fim, conclui-se que o processo de otimização é comum para todas as bases de dados, que, apesar das suas diferenças, convergem na busca pela eficiência e chegam mesmo a apresentar algumas semelhanças em parte dos processos de otimização.

V. Referências

ChatGPT. (12 de 11 de 2023). *ChatGPT*. (OpenAI) Obtido de https://chat.openai.com/c/31c00fba-3e74-4cd4-97a4-92d73fc58129

Antoshenkov, G., & Ziauddin, M. (1996). Query processing and optimization in Oracle Rdb. The VLDB Journal the International Journal on Very Large Data Bases, 5(4), 229–237. https://doi.org/10.1007/s007780050026

Vinaysirsal. (2020, April 20). Query Processing and Optimization in Oracle. Medium. https://vinaysirsal1.medium.com/query-processing-and-optimization-in-oracle-b256cfe49e2d

Optimizer with Oracle Database 18c A R Y 2 0 1 8. (n.d.). Retrieved November 19, 2023, from https://www.oracle.com/docs/tech/database/technical-brief-optimizer-oracle-db-0218.pdf

Query Optimization | Oracle. (2019). Oracle.com. https://www.oracle.com/database/technologies/datawarehouse-bigdata/query-optimization.html

Query Optimization — MongoDB Manual. (n.d.). Www.mongodb.com. Retrieved November 19, 2023, from https://www.mongodb.com/docs/manual/core/query-optimization/

Conducting MongoDB Query Performance Analysis | Hevo. (2022, February 16). https://hevodata.com/learn/mongodb-query-performance-analysis/

Index Types — MongoDB Manual. (n.d.). Www.mongodb.com. Retrieved November 19, 2023, from https://www.mongodb.com/docs/manual/core/indexes/index-types/#single-field-index

NodeTeam. (2023, February 16). How to Optimize PostgreSQL Queries. Medium. https://nodeteam.medium.com/how-to-optimize-postgresql-queries-226e6ff15f72

Markovtsev, V. (2022, March 9). How we optimized PostgreSQL queries 100x. Medium. https://towardsdatascience.com/how-we-optimized-postgresql-queries-100x-ff52555eabe

PostgreSQL Query Optimization | Learn to PostgreSQL Query Optimization. (2020, November 24). EDUCBA. https://www.educba.com/postgresql-query-optimization/

NodeTeam. (2023, February 16). How to Optimize PostgreSQL Queries. Medium. https://nodeteam.medium.com/how-to-optimize-postgresql-queries-226e6ff15f72

The Internals of PostgreSQL : Chapter 3 Query Processing (Part 1). (n.d.). Www.interdb.jp. Retrieved November 19, 2023, from https://www.interdb.jp/pg/pgsql03.html

The PostgreSQL Global Development Group. (2019). PostgreSQL: The world's most advanced open source database. Postgresql.org. https://www.postgresql.org/

Sachdeva, S. (2021, October 5). SQL Query Optimization | A Detailed Guide on SQL Query Optimization. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/10/a-detailed-guide-on-sql-query-optimization/

Győrödi, C. A., Dumşe-Burescu, D. V., Győrödi, R. Ş., Zmaranda, D. R., Bandici, L., & Popescu, D. E. (2021). Performance Impact of Optimization Methods on MySQL Document-Based and Relational Databases. Applied Sciences, 11(15), 6794. https://doi.org/10.3390/app11156794

Rupley, M. (n.d.). Introduction to Query Processing and Optimization Introduction to Query Processing and Optimization. https://clas.iusb.edu/computer-science-informatics/research/reports/TR-20080105-1.pdf

Rupley, M. (n.d.). Introduction to Query Processing and Optimization Introduction to Query Processing and Optimization. https://clas.iusb.edu/computer-science-informatics/research/reports/TR-20080105-1.pdf