



Complementos de Bases de Dados Teóricas

Resumos

- O foco nos dados
- Modelos de bases de dados
- Armazenamento e recolha de dados
- Formatos de dados
- Bases de dados documentais
- Bases de dados orientadas a colunas
- Bases de dados orientadas a grafos
- Bases de dados distribuídas e paralelas
- Replicação de bases de dados
- Partição de bases de dados
- Processamento de dados distribuídos

Gonçalo Matos, 92972

Licenciatura em Engenharia Informática

3.º Ano | 1.º Semestre | Ano letivo 2020/2021

Última atualização a 15 de janeiro de 2021

Este documento é baseado nos *slides* teóricos dos docentes José Luís Oliveira e Carlos Costa e na primeira edição do livro **Designing Data-Intensive Applications**, escrito por Martin Kleppmann. Fontes adicionais são referenciadas no início dos capítulos onde foram utilizadas.

Índice

1. O foco nos dados.....	7
Requisitos.....	8
Bases de dados.....	8
2. Modelos de bases de dados.....	9
Bases de dados relacionais.....	9
Impedance Mismatch.....	9
Normalização.....	9
Responder ao aumento do fluxo de dados.....	10
O movimento NoSQL.....	10
Transações BASE.....	10
Teorema CAP.....	10
Bases de dados NoSQL.....	11
Key-value.....	11
Documentais.....	11
Column databases.....	12
3. Armazenamento e recolha de dados.....	13
Append-only log: Um exemplo em bash.....	13
Índices.....	13
Gestão do espaço em disco.....	14
Outros problemas.....	14
Sorted String Table (SSTable).....	15
Ordenação.....	15
Recuperação a falhas.....	15
B-trees.....	16
Operações.....	16
Aspetos a considerar.....	16
Otimizações.....	16
Outras estruturas de indexação.....	17
Bases de dados em memória.....	17
Processamento e análise transacional.....	18
Online Transaction Processing (OLTP).....	18
Online Analytic Processing (OLAP).....	18
OLTP vs. OLAP.....	18
Implementações distintas.....	19
Data warehouse.....	19

Esquema da base de dados.....	19
Queries.....	20
Armazenamento orientado a colunas.....	20
Escrita.....	20
Materialized views.....	20
4. Formatos de dados.....	21
Formatos específicos de linguagens.....	21
Formatos textuais.....	21
Codificação binária.....	21
Formatos textuais.....	22
CSV (Comma-Separated Values).....	22
XML (Extensible Markup Language).....	22
JSON (JavaScript Object Notation).....	22
BSON (Binary JSON).....	22
RDF (Resource Description Framework).....	22
Protocol Buffers (.proto).....	22
6. Bases de dados documentais.....	23
7. Bases de dados orientadas a colunas.....	24
Modelo de dados.....	24
Casos de uso.....	25
Cassandra.....	25
Topologia do sistema.....	25
Arquitetura do sistema.....	25
8. Bases de dados orientadas a grafos.....	26
Estruturas de dados.....	26
Matriz de adjacências.....	26
Lista de adjacências.....	26
Matriz de incidência.....	27
Matriz laplaciana.....	27
Travessias nos grafos.....	27
Tipos de pesquisas.....	28
Bases de dados orientadas a grafos.....	28
Modelos de base de dados.....	29
Neo4j.....	29
9. Bases de dados distribuídas e paralelas.....	30
Bases de dados paralelas.....	30
Vantagens.....	30
Desvantagens.....	31

Arquiteturas.....	31
Bases de dados distribuídas.....	32
Bases de dados paralelas vs. distribuídas.....	32
Técnicas de bases de dados paralelas e distribuídas.....	33
Localização dos dados.....	33
Otimização de queries em paralelo.....	33
Processamento de dados em paralelo.....	34
Gestão da carga.....	35
Gestão de transações distribuídas.....	35
10. Replicação de bases de dados.....	37
Replicação single-leader.....	37
Sincronismo.....	37
Novos seguidores.....	38
Saídas de seguidores.....	38
Logs de replicação.....	39
Lag na replicação.....	39
Replicação multi-leader.....	40
Cenários de utilização.....	40
Técnicas de replicação.....	41
Replicação leaderless.....	41
Recuperação de falhas.....	41
Quórum para leituras e escritas.....	42
Cenários de utilização.....	42
Gestão de escritas concorrentes.....	43
11. Partição de bases de dados.....	44
Tipos de partição.....	44
Partição de dados chave-valor.....	44
Partição e índices secundários.....	45
Equilíbrio das partições.....	46
Round-robin.....	46
Número fixo de partições.....	46
Partição dinâmica.....	46
Processo de roteamento.....	47
12. Processamento de dados distribuídos.....	48
Frameworks MapReduce.....	49
Apache Hadoop.....	49
HDFS.....	49
YARN.....	50
Hadoop MapReduce.....	50

1. O foco nos dados

Slides teóricos, "Designing Data-Intensive Applications" e [Batch processing vs. Stream Processing](#)

Cada vez as aplicações são focadas nos dados, em detrimento da capacidade de computação, que é raramente uma condicionante. A capacidade da CPU passa assim a ser um problema secundário quando comparado com a **quantidade, complexidade e velocidade de atualização** dos dados.

De forma a otimizar a sua performance, um sistema de dados tipicamente oferece as seguintes funcionalidades:

Bases de dados armazenam os dados para utilização futura;

Caches guardam os resultados de operações dispendiosas, de forma a tornar a leitura mais rápida;

Search indexes permitem aos utilizadores procurarem por palavras-chave ou filtrar os dados

Message queues permitem a comunicação assíncrona entre processos

Stream processing processamento de dados em tempo real, assim que que gerados;

Batch processing processamento de dados (passados) num determinado intervalo de tempo;

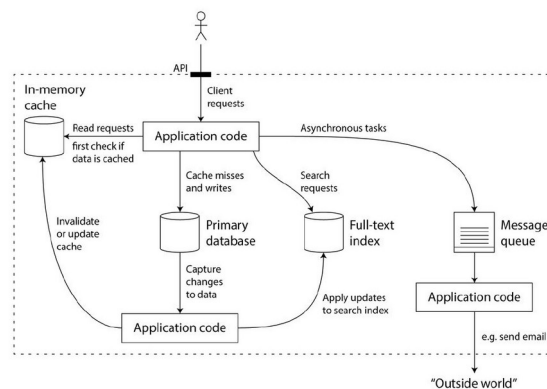
Um exemplo de **stream processing** ocorre na banca. Sempre que é realizada uma transação, os dados da mesma são imediatamente processados de forma a que o saldo esteja sempre atualizado.

O **batch processing** é visível na faturação dos serviços pós-pagos pelas operadoras de telecomunicações. No final de cada mês, é feita uma consulta às suas bases de dados de forma a identificar todos os consumos do cliente, que são somados e depois gerada a fatura.

No **stream** os dados são processados antes de armazenados, enquanto que no **batch** são processados depois de armazenados.

No entanto, nenhum desenvolvedor de uma aplicação vai desenvolver um sistema de bases de dados. Já existem demasiados e bastante eficientes nas tarefas a que se propõe. Ainda assim, não é aqui que a tarefa fica mais simples, pois ainda há a necessidade de **escolher um sistema adequado, que responda às exigências do sistema a desenvolver**.

Muitas vezes haverá ainda a situação de que não existe um único sistema que responda a todas, havendo a necessidade de trabalhar com vários de forma integrada, através do código da aplicação.



Requisitos

De forma a garantir que os dados mantêm a sua integridade em casos de falhas internas, que a performance é assegurada mesmo em cenários de degradação dos componentes ou de aumentos repentinos na quantidade de informação, há requisitos que os sistemas devem cumprir.

Fiabilidade. O sistema deve continuar a funcionar em caso de adversidades (falhas de *software*, *hardware* ou mesmo humanas).

Escalabilidade. O sistema deve ser capaz de responder ao crescimento seja do volume de dados, do tráfego ou mesmo da complexidade.

Manutenção. Deve ser possível que o sistema sofra alterações ao longo do tempo por várias pessoas diferentes de forma produtiva.

Bases de dados

São definidas como um **conjunto de dados relacionados entre si e a sua organização**.

Dividem-se em vários tipos, sendo atualmente as mais comuns as **relacionais**, seguidas das **documentais**, **motores de busca**, **chave-valor**, entre outras.

O controlo às bases de dados é realizado por **Sistemas de Gestão de Base de Dados (SGBD** ou DBMS em inglês). Estes fornecem funções que permitem a manipulação de grandes quantidades de informação.

2. Modelos de bases de dados

Slides teóricos e aula assíncrona

Os modelos de dados com os quais os programas vão trabalhar têm um papel fundamental na sua programação.

Bases de dados relacionais

Este tipo de base de dados oferece vários benefícios, entre os quais a **persistência** dos dados, a **integração** com várias aplicações e a **atomicidade**, **consistência**, **isolamento** e **durabilidade** oferecidas pelas transações (ACID).

Atomicidade Numa transação, ou todas as operações são realizadas, ou nenhuma o é.

Consistência É garantido que as restrições de integridade antes da transação se mantêm após esta.

Isolamento As alterações feitas na BD só são visíveis quando a transação termina.

Durabilidade Assim que *committed*, as alterações de uma transação persistem mesmo em caso de falhas.

A durabilidade é garantida através das *transactions log*, que permite a reconstrução das transações perdidas em caso de falhas.

Impedance Mismatch

Nos últimos tempos, tem-se assistido a um **aumento do volume de dados e tráfego**, a par da redução do relacionamento entre eles, ou seja, **cada vez há mais dados não relacionados**.

Tem-se também verificado um **conflito entre os princípios de engenharia de *software*, onde o paradigma é orientado a objetos e os princípios relacionais baseados em modelos matemáticos**. Este problema é designado por **Impedance¹ Mismatch²**.

Atualmente, este verifica-se nas estruturas isoladas, que violam os princípios da **normalização**.

Por exemplo, vários objetos que representem funcionários numa empresa. Cada funcionário terá o seu departamento, mas vários funcionários podem trabalhar no mesmo departamento. Se a base de dados refletir o paradigma orientado a objetos, iremos ter uma repetição dos departamentos nos vários funcionários e base de dados não estará normalizada!

No entanto, fazer múltiplos *selects* e *joins* para construir uma entidade às vezes não é a melhor opção!

Normalização

A **normalização** visa **reduzir a redundância dos dados**.

Em bases de dados, esta reflete-se na **utilização de IDs** para identificar entidades, oferecendo uma **consistência** de utilização, ao mesmo tempo que **previne ambiguidades** caso hajam entidades semelhantes (com o mesmo nome, p.e.), **facilita alterações** das entidades, uma vez que a sua informação está armazenada numa única tabela, motivo pelo qual também **facilita a tradução**.

Uma base de dados que verifica estas características diz-se **normalizada**.

¹ Oposição que um circuito elétrico faz à passagem de corrente elétrica quando é submetido a tensão.

² Disparidade, incompatibilidade.

Responder ao aumento do fluxo de dados

Existem duas abordagens possíveis: **construir bases de dados maiores** ou **criar um grupo de máquinas mais pequenas que se complementam**.

A primeira abordagem tem alguns problemas, uma vez que o **custo** de duplicar a capacidade de uma base de dados é geralmente mais do dobro do custo de uma base de dados “normal” e mesmo com recursos financeiros, há **limitações físicas** e de engenharia à sua capacidade.

A segunda, apesar de mais exequível, tem também alguns defeitos, uma vez que por ser uma solução **mais barata**, pode refletir-se em **menos fiabilidade**. É ainda necessário a integração com um DBMS compatível com esta tipologia.

Os Sistemas de Gestão de Bases de Dados têm alguma dificuldade em gerir a escalabilidade horizontal (distribuição da BD).

O movimento NoSQL

Este movimento, cujo acrónimo significa *Not only SQL*, pretende promover a utilização de bases de dados não relacionais (SQL). Tem por base vários princípios.

Não relacional

API simples sem necessidade de fazer *joins*.

Teorema BASE & CAP Viola os princípios ACID (atomicidade, consistência, isolamento e durabilidade).

Schema-free Esquema implícito e gerido pela aplicação (sem verificações do lado da BD).

Distribuídas A maior parte são.

Open source Mais uma vez a maior parte.

Transações BASE

Este acrónimo nasceu em oposição aos princípio ACID, principalmente em resposta às limitações de consistência que um cenário de um sistema distribuído impõe.

Basic availability A base de dados funciona a maior parte do tempo.

Soft-state As manipulações dos dados não têm de ser *write-consistent*

Escritas num nó da base de dados não têm de ser escritas garantidamente em simultâneo nos restantes nós.

Eventual consistency A consistência é assim eventual. Há de ser atingida em algum momento.

As bases de dados NoSQL caracterizam-se então por ser **otimistas** e **simples**, o que torna a base de dados mais **rápida**.

Teorema CAP

Este teorema diz que **um sistema distribuído só pode apresentar duas de três características**:

Consistência Escritas atómicas em toda a base de dados em simultâneo

Disponibilidade A base de dados responde sempre aos pedidos

Tolerância a falhas O sistema consegue funcionar mesmo que um nó deixe de responder



No exemplo do lado esquerdo temos uma base de dados que implementa a **consistência** e a **tolerância a falhas**. Quando um cliente faz um pedido de consulta de um valor, caso o nó não consiga contactar os restantes de forma a confirmar que todos têm o mesmo valor, retorna uma mensagem de erro.

No lado direito temos a **disponibilidade** e **tolerância a falhas**. Neste caso, mesmo com uma falha de comunicação entre os nós, o nó contactado pelo cliente vai responder com o valor pedido, mesmo que este não seja o mais atual.

Bases de dados NoSQL

Existem vários tipos de bases de dados NoSQL.

Key-value

Estas base de dados estão focadas num **armazenamento chave/valor** numa tabela de dispersão.

Chave Identificador (chave primária)

Valor Pode assumir uma de várias estruturas de dados

As operações são realizadas sobre um valor de uma determinada chave.

A sua **simplicidade** permite uma boa **performance** e facilidade na **escalabilidade**. No entanto, não permite a realização de *queries* complexos nem o armazenamento de dados complexos.

Usadas para perfis de utilizadores, informações de sessão, carrinhos de compras, preferências do utilizador...

Tipicamente armazenam dados não persistentes. Não permitem relações entre entidades.

Documentais

O **modelo de dados são estruturas compactas** (tipicamente JSON), **self-describing**, uma vez que o nome dos atributos se decreve a si próprio, organizadas numa **estrutura hierárquica** e onde cada documento tem um **identificador único**.

Permitem *queries* sobre vários documentos, não só pela sua chave (identificador), mas também pelo seu valor.

Usadas para log de eventos, blogs, *web analytics*, aplicações *e-commerce*, ...

Não são aconselhadas em relações many-to-many.

Column databases

Os dados são armazenados em colunas em vez de em linhas.

Os *queries* são feitos sobre uma linha de uma dada família de colunas.

Usadas em *log* de eventos, blogs, sistemas de gestão de conteúdos

// TODO Completar a partir do slide 31 (de 41)

3. Armazenamento e recolha de dados

Slides teóricos e aulas assíncronas

Até aqui as bases de dados foram analisadas da perspetiva do utilizador, de quem armazena dados e opera sobre eles. No entanto, a escolha de uma base de dados para um projeto implica um conhecimento mais profundo sobre o seu modo de funcionamento interno.

Neste capítulo será explorada em detalhe a forma como as bases de dados armazenam e obtêm os dados e distinguidas as bases de dados otimizadas para trabalhos transacionais das otimizadas para analíticos.

Append-only log: Um exemplo em bash

O *script* abaixo define uma base de dados em bash.

```
#!/bin/bash

cbd_set () { echo "$1,$2" >> database }

cbd_get () { grep "^$1," database | sed -e "s/^$1,/" | tail -n 1 }

# Usage: $ cbd_set <key> <value> $ cbd_get <key>
```

Este limita-se a adicionar a um ficheiro (*append*) um par chave/valor em cada linha através da função *cbd_set* e a filtrar o seu conteúdo para as linhas que contenham a chave no *cbd_get*, mostrando a última correspondência (chave mais recente).

Uma vez que não altera o conteúdo do ficheiro, limitando-se apenas a acrescentar-lhe linhas no final, a função de inserção tem uma **excelente performance**, $O(1)$, que é independente do tamanho da base de dados.

Por outro lado, as consultas são **pouco eficientes** uma vez que obrigam à consulta de todo o ficheiro à procura da chave mais recente que corresponda. Apresenta por isso um desempenho $O(n)$.

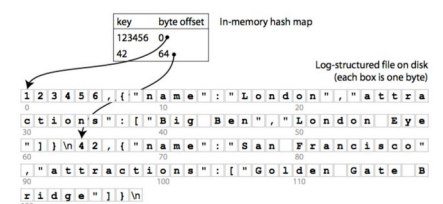
A não repetição de chaves iria aumentar a eficiência da consulta uma vez que assim que encontrasse uma correspondência não precisava de continuar à procura da chave mais recente. No entanto, neste cenário a inserção iria ser prejudicada, uma vez que a alteração de uma linha implica a leitura e reescrita da totalidade do ficheiro, uma operação que é bastante mais dispendiosa que o *append* que faz atualmente.

Índices

Uma solução para aumentar a eficiência das consultas são os **índices**, **estruturas adicionais às tabelas das BD que mantêm um mapeamento entre as chaves e a sua posição na base de dados**.

A localização pode ser por exemplo o *byte offset* dentro do ficheiro da BD.

Apesar de agilizarem as consultas, prejudicam ligeiramente as inserções, uma vez que para além de registar os pares chave/valor é necessário atualizar também a sua localização nos índices associados.



Gestão do espaço em disco

Uma base de dados como a analisada em que o conteúdo é sempre adicionado e nunca reescrito vai ocupar cada vez mais espaço ao longo do tempo, aumentando também o tempo das operações de consulta. A solução passa por **segmentar** e **compactar e combinar**.

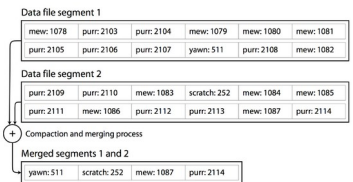
Segmentar Em vez de utilizar um único ficheiro, utilizar vários.

Os ficheiros podem ter um tamanho máximo que quando é atingido faz com que os dados passem a ser armazenados num novo ficheiro. Com este modo de funcionamento pode inveter-se a pesquisa e começá-la pelos ficheiros mais recentes, evitando assim a análise de versões mais antigas da chave a ser procurada.

O pior caso passa no entanto a ser menos eficiente, que é quando a chave tiver sido escrita apenas no primeiro ficheiro e nunca alterada, o que implica a análise de todos os ficheiros desde o mais recente ao mais antigo. Ao tempo da análise soma-se o tempo de abertura dos vários ficheiros.

Compactar e combinar Combinar os ficheiros antigos em versões mais compactas, mantendo apenas as versões mais recentes das chaves.

Esta operação pode ser realizada periodicamente por *threads* em *background* e não irá interferir com a disponibilidade da BD, uma vez que as operações são realizadas sobre os ficheiros mais recentes, que não estão a ser manipulados.



Outros problemas

Fonte adicional: [Eddie Woo](#) sobre Checksum

Para além dos problemas analisados anteriormente há outros fatores que devem ser considerados.

O **formato de ficheiro CSV** não é o mais eficiente, sendo **preferível a utilização de ficheiros binários**.

Quanto à **eliminação de registos**, a remoção de todos os pares de uma chave em todos os ficheiros torna-se muito dispendiosa uma vez que como já foi mencionado implica a leitura e reescrita de todos os ficheiros alterados. A solução mais eficiente passa pela **adição de um registo especial que indica a sua remoção** (*tombstone*³).

No que toca à **recuperação de falhas**, se o sistema utilizar estruturas de mapeamento (índices) em memória, o sistema irá demorar algum tempo a reconstruí-los em caso de falha do sistema uma vez que serão perdidos. A solução passa pela **criação de snapshots em memória dos índices**.

Há ainda a hipótese de **registos escritos parcialmente** caso o sistema falhe durante um processo de inserção. Uma solução possível é a **implementação de checksums**.

Por fim é ainda importante controlar o **acesso em concorrência** aos ficheiros. Sendo feita de forma sequencial, a **escrita deve ser feita apenas por uma thread única**, enquanto que pelos dados serem imutáveis e *append-only* a **escrita pode ser feita em simultâneo por várias threads**.

Mesmo havendo soluções para todos os problemas apresentados, há alguns que não têm resolução, como a necessidade da **hash-table caber em memória** e a dificuldade em fazer **queries para intervalos de valores**.

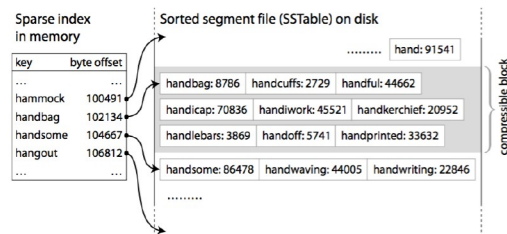
Sorted String Table (SSTable)

Neste tipo de base de dados os **pares chave/valor são ordenados pela chave, que é única** (não pode aparecer mais do que uma vez).

Aqui a **segmentação** torna-se ainda **mais eficiente**, uma vez que em cada segmento estarão os pares cujas chaves estão próximas em termos de ordenação.

Como consequência os **índices** tornam-se **menos densos** porque deixam de ter a necessidade de indexar todas as chaves e passam a integrar apenas os padrões que identificam o início de cada segmento (por exemplo “aa”, “br”, “h”).

A segmentação deve ser um compromisso entre ter o menor número de blocos possível e ter blocos o mais pequenos possível.



Como os valores não são repetidos, a atualização do seu valor implica a reescrita do segmento em que se insere. Uma abordagem que **previne escritas constantes** é o **armazenamento das atualizações dos valores numa árvore binária em memória**, a partir da qual os valores em disco são periodicamente atualizados.

Ordenação

Quando a estrutura em árvore em memória atinge um determinado limite, esta pode ser armazenada em disco gerando um novo segmento.

As consultas analisam primeiro a árvore binária e se não encontrarem vão procurar no segmento mais recente, regredimento depois no tempo até encontrar uma correspondência.

Esta abordagem, apesar de agilizar os processos de escrita prejudica os de leitura. Deve para evitar isto ser aplicada periodicamente a **combinação e compactação**.

Recuperação a falhas

Uma vez que os **dados mais recentes são armazenados em memória**, em caso de falha do **sistema serão perdidos**. Para evitar este problema, muitas SST mantêm um **append-only log** em memória, que atualizam em casa escrita, de forma a permitir esta recuperação.

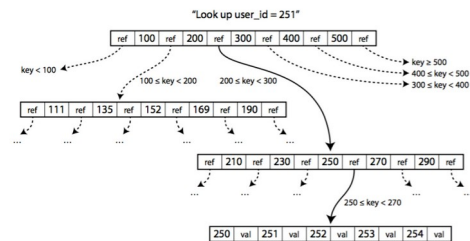
Este log pode ser limpo periodicamente, após estar garantida a escrita do conteúdo em memória no disco.

B-trees

Este é o mecanismo de indexação mais comum em sistemas de bases de dados. Tal como as SST estudadas anteriormente, as B-trees **mantêm os pares chave/valor ordenados pela chave**, o que torna as **leituras bastante eficientes**, mesmo com queries com intervalos.

Destacam-se ainda por **dividir a base de dados em blocos/páginas de tamanho fixo**, o que facilita o seu armazenamento uma vez que estão em sintonia com o *hardware*.

Na camada do *hardware* os discos também são compostos por blocos de tamanho fixo.



Em cada página há **k valores e k+1 referências** a outras páginas, sendo k geralmente um valor na ordem das centenas.

Operações

As operações de pesquisa, inserção e remoção são de complexidade $O(\log_k n)$.

A **inserção** é a operação com maior custo, uma vez que no caso de ser feita numa página que já tenha atingido o tamanho k, obriga à sua divisão em duas páginas e à reorganização dessa área da árvore, que pode impactar várias páginas.

Aspetos a considerar

A **gestão de falhas** pode ser feita tal como nas SST com **append-only logs**.

A **gestão de concorrência** torna-se também fundamental, uma vez que não são desejáveis escritas em simultâneo. Utilizam-se **semáforos**.

Otimizações

De forma a responder aos problemas endereçados no último tópico pode ainda ser aplicada **a técnica copy-on-write scheme**.

A técnica **copy-on-write** consiste na criação de páginas novas cada vez que alguma vai ser reescrita, mudando a referência da antiga para a nova apenas quando a última estiver concluída.

Garanta-se assim que caso haja alguma falha no processo de escrita a integridade da base de dados não é comprometida (embora as atualizações que estavam a ser feitas possam ser perdidas) e ainda que podem continuar a haver leituras sobre os dados antigos com garantia de que a base de dados está íntegra durante o processo de cópia.

Podem ainda ser aplicadas técnicas de **abreviação das chaves**.

Outras estruturas de indexação

Por vezes é útil a criação de **índices secundários** cujas **chaves são atributos não únicos**. Podem fazer corresponder a uma chave as suas várias referências.

Há ainda cenários em que se justifica o **armazenamento dos valores no índice**, total ou parcialmente. Tem limitações dependendo do espaço da memória. Facilitam a leitura, mas prejudicam escritas.

Para situações em que seja necessário fazer *queries* sobre várias colunas podem ser definidos **índices multi-column**.

Todos os anteriores pressupõe a pesquisa por termos exatos. Os **índices fuzzy** fornecem uma solução para **pesquisas por similaridade**. Muito usado em pesquisas textuais (like).

Bases de dados em memória

Hoje em dia já existem bastantes bases de dados que funcionam baseadas em memória, oferecendo redundância e persistência, ou seja, tolerantes a falhas.

São geralmente mais rápidas que as suas homólogas⁴, uma vez que não têm de codificar os dados de forma a poderem ser armazenados em disco. Permitem por isso trabalhar com estruturas que são difíceis de guardar em disco como *priority queues* e *sets*.

A rapidez não se deve por isso diretamente ao facto de não terem de escrever em disco, uma vez que as bases de dados em disco são geralmente carregadas em memória e acedidas a maior parte das vezes a partir da memória.

⁴ Lados opostos.

Processamento e análise transaccional

Online Transaction Processing (OLTP)

O **processamento transaccional** define-se por **permitir aos clientes fazer leituras e escritas com baixa latência**. As bases de dados que seguem este padrão de acesso dizem-se **OLTP**.

Contrasta com o **batch processing**, **uma tarefa de processamento realizada periodicamente** (p.e. uma vez por dia) para algum fim.

Online Analytic Processing (OLAP)

Com a evolução das bases de dados, estas começaram a ser também utilizadas para **análise de dados analíticos**, que geralmente consiste em **ler e processar uma grande quantidade de dados**.

Do ponto de vista técnico a construção do *query* não é complexa, mas a tarefa em si pode tornar-se dependendo do volume de dados em análise, que pode levar algum tempo a ser processado. Pode ainda haver o cenário em que um *query* para dados do ano anterior ser feito várias vezes, desnecessariamente uma vez que os dados não vão mudar.

OLTP vs. OLAP

O OLTP faz leituras de poucos registos de cada vez e de acesso aleatório, tal como a escrita, que deve ter uma baixa latência. Os dados, utilizados maioritariamente pelo segmento operacional das empresas, devem estar sempre atualizados e ocupam gigas.

Já no OLAP a leitura é feita em massa e a escrita em *bulk import*, sobre dados históricos, que ocupam teras. É utilizado pro gestores como suporte às suas tarefas de controlo e planeamento.

Property	OLTP	OLAP
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Primarily used by	End user/customer, via web application	Internal analyst, for decision support
What data represents	Latest state of data (current point in time)	History of events that happened over time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

Os sistemas **OLAP geralmente ocupam mais espaço que os OLTP porque não apresentam normalização dos dados**, ou seja, não tiram partido das características relacionais.

Dadas as suas características distintas, estes padrões são geralmente implementados por bases de dados distintas.

As focadas em OLTP dizem-se **relacionais**, enquanto que as OLAP se dizem **data warehouses**.

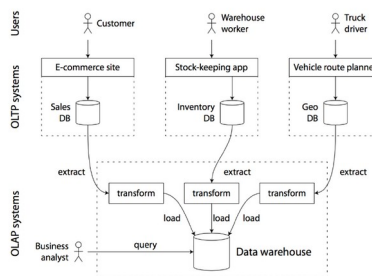
Implementações distintas

Apesar das bases de dados relacionais permitirem fazer processamento analítico de forma bastante simples, há alguns motivos que levam a que as bases de dados OLAP sejam independentes, nomeadamente as **funcionalidades** e **tipologia de dados**:

- x Nas bases de dados operacionais, de forma a manter a eficiência os **dados mais antigos são geralmente arquivados** e retirados de produção;
- x A necessidade de **dados consolidados** para análise, que envolvem agregações e processamento de dados de fontes heterogéneas;
- x A **qualidade dos dados**, que por terem origem em fontes heterogéneas pode requerer processamentos complexos.

Data warehouse

Estas bases de dados focadas na análise de dados geralmente estão associadas às transacionais, das quais **extraem** os dados, que são **transformados** e **limpos** antes de **carregados**.



Este processamento é conhecido por **ETL (Extract, Transform, Load)**.

Os dados podem ser provenientes de várias bases de dados e ser **transformados** de forma a ser integrados de forma unificada. Pode ainda ser feita uma **limpeza** eliminando registos nulos ou não consistentes.

Esquema da base de dados

O esquema destas bases de dados é conhecido por **esquema em estrela** e **consiste numa tabela principal, designada por fact table**, onde cada linha corresponde a um evento e cada atributo um valor ou referência a outra tabela.

Este esquema apresenta apenas duas dimensões. Ou seja, apesar da **fact table** poder ter referências para outras tabelas (segunda dimensão), estas por sua vez já não vão poder fazer referências a outras.

dim_product table					dim_store table		
product_sk	sku	description	brand	category	store_sk	state	city
30	OK4012	Bananas	Freshmax	Fresh fruit	1	WA	Seattle
31	KA9511	Fish food	Aquatech	Pet supplies	2	CA	San Francisco
32	AB1234	Croissant	Delectious	Bakery	3	CA	Palo Alto

fact_sales table							
date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	31	3	NULL	NULL	1	2.49	2.49
140102	69	5	19	NULL	3	14.99	9.99
140102	74	3	23	191	1	4.49	3.89
140102	33	8	NULL	235	4	0.99	0.99

Existe ainda um modelo alternativo chamado **snowflake**, em que divide as duas dimensões do anterior em várias. É por isso mais normalizada, mas como consequência mais complexa de utilizar.

Queries

O facto de armazenar os dados históricos não normalizados leva a que as tabelas destas bases de dados tenham um **volume gigante de registos com um número muito elevado de colunas**. Isto não é um problema nas consultas às linhas, porque geralmente são apenas lidas poucas de cada vez.

No entanto, **para obter todos os valores de uma coluna é necessário ler a totalidade da tabela, desperdiçando imensos recursos**. Neste caso, a solução passa pelo **armazenamento orientado às colunas**.

Armazenamento orientado a colunas

Este tipo de armazenamento consiste em **separar cada linha pelas suas colunas e guardar cada coluna num ficheiro/bloco separado**. Tem por base o princípio que cada ficheiro tem as **colunas de cada linha pela mesma ordem**.

Columnar storage layout:

```
date_key file contents: 140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103
product_sk file contents: 69, 69, 69, 74, 31, 31, 31, 31
store_sk file contents: 4, 5, 5, 3, 2, 3, 3, 8
promotion_sk file contents: NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk file contents: NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity file contents: 1, 3, 1, 5, 1, 3, 1, 1
net_price file contents: 13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price file contents: 13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99
```

Este tipo de armazenamento não é eficiente para leituras por linha!

Devido à **homogeneidade entre os dados** da mesma coluna é **facilitada a compressão**.

A **ordenação** pode ser realizada por uma determinada coluna dependendo dos dados e do fim a que se destinam, mas não esquecer que a mesma posição em todos os ficheiros tem de corresponder à mesma linha, pelo que todos os ficheiros têm de ser reordenados.

Escrita

Se os dados forem ordenados, a escrita de novos valores na BD torna-se um processo algo complexo, de forma a garantir a consistência.

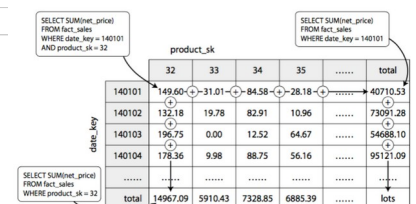
Em **estruturas operacionais** que implementam este tipo de armazenamento geralmente opta-se por **B-trees** com **append-log**. Uma boa solução é uma **LSM-tree**, que na prática é uma **SSTable**.

Materialized views

Materialized views são um **conjunto de agregações agrupadas em várias dimensões**.

Dada a complexidade dos *queries* de consulta às bases de dados OLAP, geralmente estas têm a si associadas **data cubes**, um tipo de **materialized view**, que consistem numa **tabela da base de dados em cache que armazena dados agregados para efeitos de eficiência**.

Alterações nos dados implica a necessidade de atualizar as **materialized views**.



		product_sk					
		32	33	34	35	total
date_key	140101	149.60	31.01	84.58	28.18	40710.53
	140102	132.18	19.78	82.91	10.96	73091.28
	140103	196.75	0.00	12.52	64.67	54688.10
	140104	178.36	9.98	88.75	56.16	95121.09
	total	4967.09	5910.43	7328.85	6885.39	lots

4. Formatos de dados

Slides teóricos e aula assíncrona

Devido às atualizações constantes a que as aplicações estão sujeitas, é fundamental garantir uam **independência dos dados** em relação a estas, de forma a permitir uma **compatibilidade** simultânea com o **passado** e com o **futuro**.

É fundamental que a aplicação atual consiga ler dados gerados por uma versão mais antiga da aplicação, assim como que uma versão futura consiga ler dados gerados pela versão atual.

Geralmente o programador é abstraído, sabendo apenas que ao guardar um valor de um determinado tipo, quando o for consultar, este vai continuar a ser desse tipo. Esta **abstração** é garantida pelos processos de **encoding/serialização**, que fazem a tradução dos dados para sequências de *bytes*, e de **decoding/desserialização**, responsáveis pelo inverso.

❖ Relational:	CSV
❖ Tree:	XML, JSON
❖ Graph:	RDF
❖ Binary:	BSON, Protocol Buffers

Formatos específicos de linguagens

Muitas linguagens de programação oferecem bibliotecas que permitem a realização deste processo. No entanto, os mecanismos que oferecem estão muitas vezes acoplados a essa linguagem (não permitem a leitura por outras) e dependentes da sua versão, pelo que devem ser apenas aplicados em **soluções temporárias**.

Java mapeia *Serializable*, Ruby *Marshal* e Python *Pickle*

Formatos textuais

A alternativa a este modelo são os formatos textuais, que geralmente até são **legíveis** pelo ser humano. No entanto têm alguns problemas uma vez que implementam menos restrições, que geram alguma **ambiguidade entre tipos de dados**, nomeadamente *strings* e números.

Os mais comuns são JSON e XML.

Codificação binária

A terceira alternativa consiste no armazenamento em formato binário, que apesar de não ser legível pelo ser humano, é **mais compacto e lidos mais rapidamente**, uma vez que o esquema nos indica os tipos de dados a ler, pelo que são lidos diretamente no seu formato.

Num formato textual este é lido por completo como uma *string* e dependendo do seu tipo é depois convertido.

Os formatos mais utilizados são binários de JSON (BSON, BSON, UBJSON, ...).

Formatos textuais

CSV (Comma-Separated Values)

Este formato é **pouco estandardizado**, uma vez que aceita dois tipos de separadores (vírgula e ponto e vírgula), não tem um formato escape definido, nem informação quanto à sua codificação.

XML (Extensible Markup Language)

Este formato é **semi-estruturado**, estando definidas normas quanto à forma como os dados são organizados.

A sua estrutura é composta por **constructs**, que são delimitados pelos marcadores de abertura e fecho, que podem ser do tipo vazio, textual, elemento (se tiverem *nested elements*) ou uma mistura dos anteriores.

Os atributos são feitos numa noção de **chave**(marcador)/valor.

JSON (JavaScript Object Notation)

Este é um XML simplificado baseado na notação JavaScript, como sugerido pelo nome.

Consiste numa coleção de **pares chave/valor** e **listas ordenadas**.

BSON (Binary JSON)

Formato JSON é serializado em memória num formato binário.

Na serialização são adicionados dados adicionais como os tipos de dados a serem escritos, que depois vão tornar a escrita mais rápida.

RDF (Resource Description Framework)

Formato textual muito utilizado em *semantic web*. Cada recurso tem as suas **propriedades e valores** associados e um **recurso**, que é um identificador único, ou IRI (*Internationalized Resource Identifier*).

<http://www.example.org/index.html> has an **author** whose name is **Pete Maravich**.

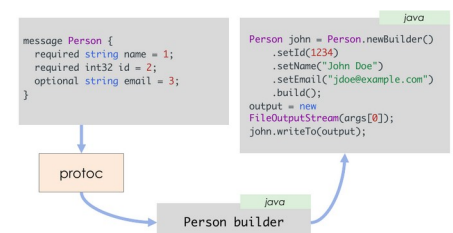
A azul temos um recurso, que tem uma propriedade a vermelho e um valor a verde.

```
<?xml version="1.0"?><br>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:same="http://msome.lycos.com/elements/1.0/myschema#">
  <rdf:Description rdf:about="http://www.cio.com/archive/km.html">
    <same:title>Less for Success</same:title>
    <same:author>Alice Dragoon</same:author>
    <same:subject>
      <rdf:Bag>
        <rdf:li>knowledge management</rdf:li>
        <rdf:li>technology investments</rdf:li>
      </rdf:Bag>
    </same:subject>
    <same:format>text/html</same:format>
    <same:status>active</same:status>
    <same:created>2004-10-19</same:created>
    <same:funFactor>3</same:funFactor>
  </rdf:Description><br>
</rdf:RDF>
```

O esquema é identificado no marcador inicial e tem de ser aplicado exatamente nesse ficheiro.

Protocol Buffers (.proto)

A Google propôs a criação de um esquema binário para representação de dados.



6. Bases de dados documentais

Slides teóricos

As bases de dados documentais são bastante eficientes em cenários *one-to-many*.

Oferecem um *esquema flexível* (mesmo dentro das mesmas coleções) e *melhor performance* (devido ao armazenamento da informação junto à entidade a que esta se refere) que são manipulados através de código simples.

A flexibilidade permite que existam objetos na mesma coleção com atributos ligeiramente diferentes, sem necessidade de criarmos uma tabela para cada tipo de objeto.

A localidade pode levar à duplicação de dados entre documentos.

Um **documento** caracteriza-se por uma *string* contínua codificada em JSON, XML, ou outro formato. É *self-describing* (atributos são claros) e apresentam uma estrutura em árvore. São identificados por um ID único.

Geralmente, para o manipular, é necessário carregá-lo por completo e para guardar as alterações re-escreve-lo na totalidade.

A localidade só se torna uma vantagem se manipularmos porções grandes do documento.

Esta topologia não se aplica a modelos *many-to-many*, pois operações não existem operações de *join* de documentos. Deve também ser evitada quando a estrutura do documento é demasiado instável.

Não é desejável demasiada granularidade entre os documentos porque se todos apresentarem características diferentes não são relacionáveis e assim não fará sentido estarem na mesma coleção.

No entanto, é a ideal para *logging* de eventos, sistemas de gestão de conteúdos, blogues, *web analytics*, aplicações *e-commerce*...

Por ser a solução para alguns, mas não para todos os problemas, as bases de dados relacionais começaram a incluir funcionalidades documentais e vice-versa.

Document

- MongoDB, Couchbase, CouchDB,
- RethinkDB, RavenDB,
- Google Cloud Firestore



Multi-model

- MarkLogic, OrientDB, ArangoDB
- Amazon DynamoDB,
- Microsoft Azure Cosmos DB,



– ... many others

No âmbito deste capítulo foi feito um resumo prático sobre MongoDB.

7. Bases de dados orientadas a colunas

Slides teóricos e aula assíncrona

Este tipo de base de dados **armazena e processa os dados em colunas**. Geralmente tem origem em queries agregadores de dados, que permitem gerar dados suscetíveis de serem analisados para fins **estatísticos** ou para **business intelligence**.

Visa então os serviços acima da utilização do armazenamento, permitindo **processamento paralelo** e consequentemente a construção de **aplicações de alto desempenho**.

A falta de normalização faz com os dados sejam esparsos⁵ e que hajam bastantes campos nulos.

Este modelo de base de dados é bastante **vantajoso** em cenários em que são feitos queries com poucas colunas sobre grandes volumes de dados. Destaca-se ainda a maior facilidade de compressão por colunas, uma vez que os dados neste domínio tendem a estar mais relacionados.

No entanto, também apresenta algumas **desvantagens**, como seja o carregamento incremental de dados, o uso de OLTP (OnLine Transaction Processing), ou a realização de queries a linhas (dados individuais).

O **carregamento incremental** de dados caracteriza-se por atualizações constantes. É uma desvantagem porque a cada inserção têm de se editar todos os ficheiros de todas as colunas, pelo que este processo geralmente é realizado periodicamente e para grandes volumes de dados em simultâneo.

Contrariamente às bases de dados relacionais, são **orientadas aos serviços** e não aos dados.

Nasceu com o projeto **Bigtable** da Google, que serviu de inspiração aos restantes SGBD.

Foi criado em resposta ao problema de geração de índices para o seu motor de busca, que levava demasiado tempo. Atualmente é utilizado também no Gmail e Google Maps.

Modelo de dados

As **tabelas** das bases de dados orientadas a colunas são denominadas por **família de colunas** e são compostas por um conjunto de linhas semelhantes.

Cada **linha** é uma coleção de colunas e tem a si associado um ID único.

Uma **coluna** tem um nome e um valor, que pode ser escalar, *set*, lista ou mapa.

⁵ Solto, dispersos, avulso.

Casos de uso

As bases de dados orientadas a colunas são **utilizadas para o armazenamento de dados estruturados com um esquema uniforme** como log de eventos, sistemas de gestão de conteúdo (blogs, ...).

Não garantem **ACID** (atomicidade, consistência, isolamento e durabilidade), **queries complexos** como **joins**, nem **prototipagem**, uma vez que a sua orientação ao serviço a torna altamente dependente dos seus requisitos.

Cassandra

Atualmente é o sistema de bases de dados orientado a colunas mais utilizado.

Foi inicialmente desenvolvido pelo Facebook (2008) mas atualmente está sob gestão da fundação para o *software* Apache e caracteriza-se por ser **open-source**, apresenta **alta disponibilidade**, **tolerância a falhas**, **escalabilidade linear** e modelos **peer-to-peer**. Implementada em Java.

Foi criada para responder ao problema de pesquisa de mensagens por *keyword* e utilizador.
É utilizada pelo Twitter, Netflix, eBay, GitHub, Instagram...

É ainda de destacar o **alto desempenho na escrita, sem prejudicar a eficiência das leituras**.

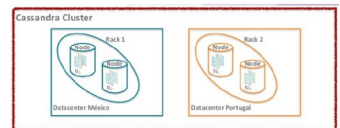
Para dados com uma dimensão superior a 50GB, as escritas e leituras levariam cerca de 300ms. A Cassandra oferece leituras a 0,12ms e escritas a 0,15ms.

Topologia do sistema

A unidade base destes sistemas são os **nós**, que são as **máquinas onde a Cassandra está em execução**.

Estes organizam-se, por sua vez, em **centros de dados**, **conjuntos de nós relacionados entre si**, que fazem **replicação dos dados** de forma a garantir a tolerância a falhas.

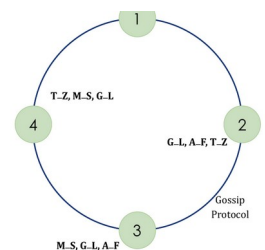
Esta hierarquia termina nos **clusters**, **conjuntos de centros de dados sobre os quais são escritos os mesmos dados**.



Arquitetura do sistema

Tratando-se de um sistema distribuído, apresenta um **protocolo de gestão de membros** que define procedimentos para a junção e gestão de membros (mesmo em caso de falha).

Este define também como é efetivada a **partição**, que é feita de forma democrática (sem *master*) num anel lógico através de redes *peer-to-peer* e com base na chave primária dos dados.



No âmbito deste capítulo foi feito um resumo prático sobre Cassandra.

8. Bases de dados orientadas a grafos

Slides teóricos e aula assíncrona

Apesar de fugirem ao conceito habitual de base de dados a que estamos habituados, a orientação de uma base de dados a grafos pode ser aplicada a diversos domínios, como a organização de sistemas de transportes ou das relações humanas.

Os seus **dados** são um **conjunto de entidades e as suas relações**.

São geralmente modelados por $G=(V, E)$

Seja V um conjunto de vértices e E um conjunto de arestas.

Num grafo, os vértices podem representar vários tipos de entidades, cujas associações (arestas) são também diversas, dependendo do tipo de nós que interligam.

Estes podem ser classificados em vários tipos.

Single-relational Arestas são homogéneas, representando o mesmo tipo de relação

Multi-relational As arestas podem representar vários tipos de relação, pelo que têm a si associado um tipo ou etiqueta. Tanto os vértices como os nós têm a si associado propriedades (um conjunto de pares chave/valor - são como objetos).

Estruturas de dados

Há vários modelos para representar os dados numa base de dados orientada a grafos.

Matriz de adjacências

Esta implementação consiste numa **matriz bi-dimensional nxn com valores booleanos** (0/1), em que cada índice representa um nó.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

$A_{ij} = 1$, se i está ligado a j

Torna bastante simples as tarefas de verificar a ligação entre dois nós e de estabelecer/quebrar ligações (arestas). No entanto, ocupa um espaço quadrático em proporção a n (nxn), as adições de nós são operações complexas (tem de se adicionar uma coluna em cada linha!), tal como a pesquisa por vizinhos, de complexidade O(n).

Podem ser implementadas variações como **grafos direcionados** ou **grafos com peso**.

Lista de adjacências

Esta alternativa apresenta um **conjunto de listas**, uma para cada nó, onde são registados os nós com os quais este se relaciona.

N1 → {N2, N3}
N2 → {N1, N3, N5}
N3 → {N1, N2, N5}
N4 → {N2, N6}
N5 → {N2, N3}
N6 → {N4}

A sua estrutura torna a representação de matrizes esparsas⁶ mais compacta. Destaca-se ainda a simplicidade da pesquisa dos vizinhos de um nó (O(x), x o número de vizinhos) e da adição de nós. No entanto, a verificação da ligação entre dois nós é mais complexa que na estrutura anterior.

⁶ Disperso, solto, avulso.

Matriz de incidência

Tal como na matriz de adjacências consiste numa **matriz bi-dimensional**, mas **nxm**, com n linhas e m colunas, que representam, respetivamente, os nós e as arestas.

$A_{ij} = 1$, se o nó i está conectado pela aresta j

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Esta abordagem é bastante útil para representar **hipergrafos**, onde uma aresta pode conectar vários nós entre si. No entanto, caso a matriz seja esparsa, vamos ter uma matriz demasiado grande para um número reduzido de 1.

Matriz laplaciana

Esta é também uma **matriz bi-dimensional nxn**, que resulta da diferença entre a matriz de graus e a matriz de adjacências. As diagonais indicam o grau do nó e as restantes posições os vértices (-1 se há ligação, 0 caso contrário).

$L = D - A$, D matriz de graus e A matriz de adjacências

$A_{ij} = -1$ se i está ligado a j

$A_{ii} \geq 0$ representa o número de arestas incidentes no nó i

$$\begin{pmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & -1 & 0 \\ -1 & -1 & 3 & 0 & -1 & 0 \\ 0 & -1 & 0 & 2 & 0 & -1 \\ 0 & -1 & -1 & 0 & 2 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$

Os prós e os contras são bastante semelhantes aos da matriz de adjacências, com a vantagem de reter mais informação.

Travessias nos grafos

De forma a conseguirmos determinar o caminho entre dois elementos (vértices ou arestas), as adjacências entre os elementos têm de ser explicitadas.

Vértices(v)/arestas(e) de entrada(_{in})/saída(_{out})

Filtragem das arestas pela sua etiqueta (e_{lab}^{VALUE})

Filtragem pelo valor para uma determinada chave (e_p)

Filtragem dos elementos que são o elemento dado ($\in =$)

Obter os valores das propriedades para uma dada chave (\in^{KEY})

Estas travessias simples podem ser compostas em travessias mais complexas.

Por exemplo, para determinar o nome dos amigos do Alberto (vértice i), começamos por considerar as suas arestas de saída que têm a etiqueta friend, para os quais vamos obter os vértices de entrada, dos quais pretendemos obter os valores da propriedade name.

$$f(i) = (\in^{name} \circ v_{in} \circ e_{lab}^{friend} \circ e_{out})(i)$$

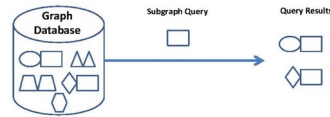
A pesquisa internas aos grafos são feita numa estrutura em árvore, pelo que podem ser aplicadas técnicas de **pesquisa em largura** (BFS) ou **profundidade** (DFS).

Tipos de pesquisas

Os *queries* feitos às bases de dados orientadas as grafos podem ser de vários tipos.

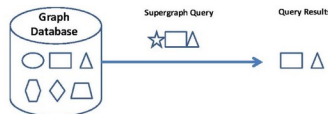
Sub-grafos quando se pretende pesquisar por grafos que apresentem um determinado padrão

A pesquisa é feita a partir de um grafo que deve estar contigo nos resultados.



Super-grafos quando a pesquisa é sobre toda a estrutura de cada membro

A pesquisa é feita a partir de um grafo, que deve ver contido em si os resultados.



Similaridade se pesquisa por grafos semelhantes, mas que não sejam exatamente isomorfos⁷ ao dado

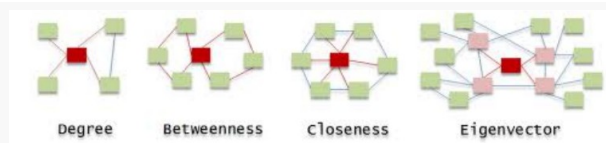
Bases de dados orientadas a grafos

Este tipo de bases de dados aplica-se a cenários em que os **dados estejam altamente conectados entre si**.

Redes sociais, caminhos, serviços baseados em localizações, motores de recomendações, química...

No entanto, não devem ser utilizadas se for necessário fazer operações de modificação de muitos nós com frequência, nem se o volume de dados for demasiado grande, uma vez que a distribuição é praticamente impossível!

Há bastantes algoritmos que tiram partido desta organização dos dados, utilizando algumas das várias propriedades que podem ser associadas a cada nó.



Grau número de ligações (vértices) a outros nós

Betweenness caso a inexistência do nó provocasse uma divisão do grafo em dois

Closeness Centralidade do nó e consequente proximidade aos restantes

Eigenvector Na prática é um grau, mas para além do número de ligações a outros nós, tem em conta também o número de ligações dos vizinhos

Um nó que tenha poucas conexões, mas se essas conexões tiverem muitas conexões, terá uma boa classificação.

⁷ Que tem forma igual ou idêntica.

Por exemplo, uma pessoa pode ter poucos conhecimentos, mas se conhecer o primeiro-ministro e o presidente da república, terá uma boa classificação uma vez que tem bom conhecimentos.

Alguns dos algoritmos mais conhecidos que fazem uso de estas e outras propriedades são o **shortest path** e o **page rank**.

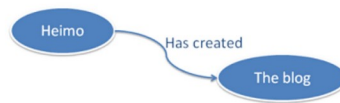
Modelos de base de dados

Vamos abordar bases de dados com o **modelo property graph**, que se caracterizam por ter **vértices** com IDs únicos, um conjunto de arestas de entrada e saída e um conjunto de propriedades. Por sua vez, as **arestas** têm também um ID único e uma coleção de propriedades, aos quais se somam um vértice de partida e um de chegada e ainda uma etiqueta que descreve o tipo de relação entre os vértices.

Vértice	Aresta
<ul style="list-style-type: none"> + ID: String + OutcomingEdges: Collection<> + IncomingEdges: Collection<> + Properties: Map<Key, Value> 	<ul style="list-style-type: none"> + ID: String + Tail: Vertice + Head: Vertice + Label: String + Properties: Map<Key, Value>

Há no entanto outro modelo, o **triple-stores**, bastante semelhante ao anterior, mas com terminologias distintas. Segundo este, a **informação é armazenada em declarações bastante simples, compostas por um sujeito, um predicado e um objeto**.

O sujeito é o equivalente ao vértice/nó do modelo anterior, tal como o predicado à aresta. **O objeto pode no entanto assumir uma de duas possibilidades**: um vértice, tal como no modelo anterior, ou um valor primitivo (como uma *string* ou um número).



Neo4j

Esta é a base de dados orientada a grafos mais utilizada atualmente, destacando-se pelo facto de ser desenvolvida em código aberto, ter facilidade de escalabilidade, alta disponibilidade, tolerância a falhas, transações ACID. É implementada em Java e foi lançada em 2007 pela Neo Technology. Utiliza a linguagem de *queries* **Cypher**.

No âmbito deste capítulo foi feito um resumo prático sobre Neo4j.

9. Bases de dados distribuídas e paralelas

Slides teóricos e aula assíncrona

Os sistemas de gestão de bases de dados (DBMS) centralizados são conhecidos pelas suas propriedades ACID⁸ em todas as transações, pelo controlo da concorrência nos acessos e ainda pelos mecanismos de recuperação a falhas.

No entanto, à medida que os dados aumentam, a **escalabilidade vertical**, que consiste na alocação de recursos à base de dados atual, não vai ser uma boa solução, uma vez que o I/O de dados (em disco) vai tornar-se progressivamente mais lento e exigente.

A escrita/leitura em bases de dados está dependente da velocidade do disco, que é bastante inferior à da RAM e à dos processadores, que vêm a sua velocidade aumentar ao longo dos anos com muito mais frequência.

De forma a aumentar a velocidade de acesso ao disco, pode ser seguida uma de duas abordagens.

Bases de dados paralelas

Estas bases de dados têm a capacidade de no mesmo servidor **utilizar múltiplos CPU e discos, de forma a suportar operações paralelas**. Permitem aumentar a velocidade de resposta aos pedidos.

Uma solução que através da abordagem **divide and conquer** permite dividir um problema grande em pequenas peças que possam ser resolvidas em paralelo. É fundamental que os dados estejam distribuídos pelos vários discos.

São utilizadas para ao armazenamento de grandes volumes de dados, em que as decisões têm de ser tomadas rapidamente.

É fundamental perceber onde vão ser colocados os dados, como vai ser feito o processamento em paralelo de forma transparente para o cliente e ainda a distribuição da carga pelos vários processadores.

Vantagens

Alta performance devido ao alto rendimento permitido pelo paralelismo inter-query, baixos tempos de resposta devido ao paralelismo intra-operação e ainda à distribuição da carga de forma a maximizar a utilização dos recursos.

Alta disponibilidade devido à replicação dos dados.

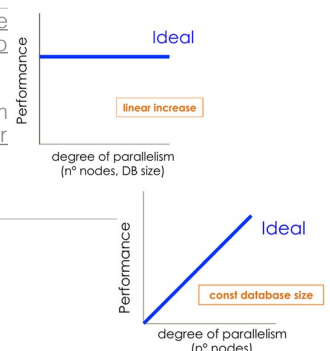
Aumenta o custo da escrita, mas em caso de falhas há uma maior garantia de disponibilidade.

Extensibilidade pode ter como objetivos a velocidade ou a escalabilidade

Caso o objetivo seja a **velocidade**, a abordagem a adotar será a **speed-up**, que para um volume da base de dados constante aumenta o número de nós disponíveis, aumentando a sua performance em proporção do último.

Caso o foco esteja na **escalabilidade** deve ser considerada a **scale-up**, que para uma base de dados com um volume de dados crescente, o número de nós é aumentado na mesma proporção, de forma a manter um desempenho constante.

Entenda-se por aumento do número de nós a adição de poder de processamento e armazenamento.



8 Atomicidade, Consistência, Isolamento e Durabilidade (Persistência)

Desvantagens

Apesar dos benefícios, esta abordagem pode trazer alguns problemas, que devem ser considerados na gestão destas bases de dados de forma a serem evitados.

Inicialização O tempo necessário para paralelizar uma operação deve ser insignificante quando comparado com o seu tempo de computação.

Interferência Quantos mais processos correrem em paralelo, mais lento cada um deles se vai tornar.

Skew Enviesamento dos dados (má distribuição). A partição dos dados deve ser feita de forma equilibrada pelos vários nós.

Arquiteturas

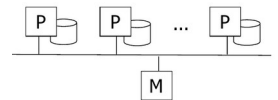
Para responder ao aumento da carga sobre a base de dados, pode optar-se por **arquiteturas multiprocessador**.

Escalabilidade vertical

Memória partilhada Cada processo tem o seu disco, mas partilham a memória RAM.

Permite comunicação entre os nós e previne cópias redundantes.

O **custo de aumento da RAM é super-linear** (duplicar a RAM mais do que duplica o preço, não é linear), para além de que uma máquina com o dobro da capacidade não conseguirá duplicar na totalidade a carga de trabalho que consegue gerir. **Não tem tolerância a falhas de memória.**

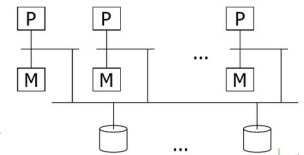


Escalabilidade vertical

Disco partilhado Cada processo tem a sua RAM, mas partilham o disco.

Permite tolerância a falhas de memória devido ao facto de cada processo ter a sua memória, o que aumenta a velocidade de processamento.

No entanto, o facto do disco ser partilhado cria **problemas no controlo de concorrência** das escritas, o que pode levar a tempos de espera elevados caso o sistema seja aplicado a cenários com muitas. Devido a este problema a **escalabilidade é limitada**.

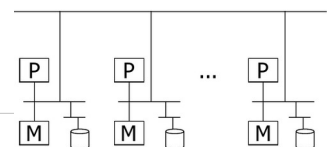


Escalabilidade horizontal

Nada partilhado Cada processo tem os seus recursos independentes.

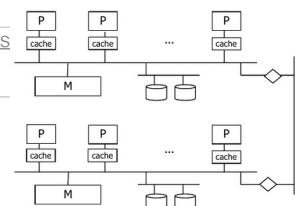
Oferece uma melhor **relação preço/desempenho**, com **fácil extensibilidade**, **alta disponibilidade** e **latência reduzida**.

Tem como desvantagens a **complexidade**, uma vez que é necessário haver uma camada de software a fazer a coordenação dos vários nós, que tem a tarefa **difícil de fazer a gestão da carga de trabalho**.



Há ainda a alternativa das **arquiteturas híbridas**, que tentam tirar partido das arquiteturas anteriores e combiná-las de forma a potenciar a eficiência dos seus sistemas, nomeadamente da **eficiência e simplicidade da memória partilhada** e da **extensibilidade e baixo custo do disco partilhado ou do nada partilhado**.

Um exemplo é a **NUMA** (non-uniform memory access), que mistura os conceitos das **memórias distribuídas** com **memória partilhada**, nomeadamente ao nível do endereçamento.



Para tal oferece **espaço de endereçamento único sobre uma memória distribuída**. Este é possível através do protocolo CC-NUMA, que permite o acesso à memória remota de outro nó de forma altamente eficiente.

Bases de dados distribuídas

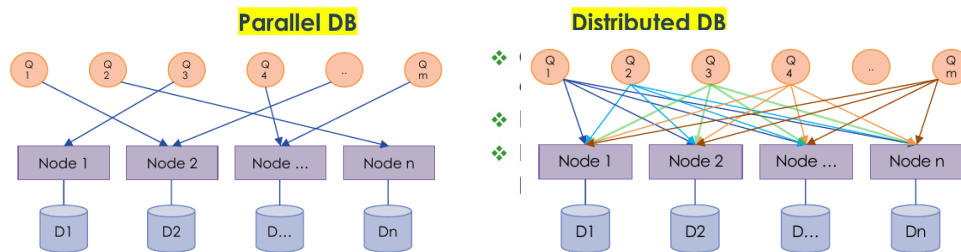
Bases de dados fisicamente separadas, cada uma a correr o seu DBMS. Oferecem...

Escalabilidade ao dividir o volume de dados por várias instâncias da base de dados.

Tolerância a falhas, pois o facto de haverem vários nós permite que haja **redundância** caso um falhe.

Latência mais reduzida, porque a distribuição física do sistema permite que o cliente se conecte à mais próxima de si.

Bases de dados paralelas vs. distribuídas



O foco das **bases de dados paralelas** é a **velocidade**, garantida pela proximidade entre os nós (geralmente na mesma sala de servidores) que permitem comunicações com lag quase inexistente.

Já as **bases de dados distribuídas** têm foco na **partilha** dos dados, que também garante eficiência em grandes volumes, mas que pode ter mais atraso nas operações devido ao lag das comunicações entre os vários pontos geográficos pelos quais os nós são distribuídos.

No entanto, estas abordagens podem ser utilizadas em simultâneo, ou seja, é possível haverem bases de dados distribuídas que em cada nó implementam processamento em paralelo.

Técnicas de bases de dados paralelas e distribuídas

As técnicas por trás das bases de dados que seguem qualquer uma destas abordagens podem variar.

Localização dos dados

A forma de distribuir os dados pode ser feita de acordo com várias técnicas.

Partição Divisão da base de dados em subconjuntos mais pequenos de dados.

Replicação Manter cópia dos dados em vários nós aumenta a redundância e tolerância a falhas.

Estas podem ser utilizadas de forma isolada ou em conjunto, devendo em qualquer um dos casos ser garantida a **transparência para o utilizador**.

No caso da **partição horizontal** do sistema, é necessário implementar técnicas para dividir os dados pelos vários nós da base de dados.

Round robin Consiste em atribuir os tuplos de forma sequencial aos nós de forma sucessiva.

Os tuplos estão bem distribuídos. Não há qualquer tipo de agregação/relação entre os tuplos em cada disco.

Hash partitioning Faz síntese de um ou mais atributos para um tuplo que devolve o número do nó.

Bom para pesquisas sobre a chave de síntese. Não há agregação, logo é mau para queries de range ou pesquisas sobre outros atributos que não o chave.

Range partitioning Associa um intervalo de valores de uma determinada chave a cada nó

Por exemplo para um atributo do tipo ASCII um nó poderia ficar responsável pelo intervalo A-D, outro pelo E-FA, outro pelo FB-Z.

	Round Robin	Hashing	Range
Sequential Scan	Best/good parallelism	Good	Good
Point Query	Difficult	Good for hash key	Good for range vector
Range Query	Difficult	Difficult	Good for range vector

Na figura do lado direito vemos a análise de desempenho para três queries:

- Seleção de todos os tuplos da tabela (SELECT * FROM table);
- Localização de tuplos que tenham determinado atributo com determinado valor (SELECT * FROM table WHERE x=y);
- Localização de tuplos que tenham determinado atributo num determinado intervalo (SELECT * FROM table WHERE x<y AND x>z).

Otimização de queries em paralelo

Há duas técnicas que permitem a execução de queries em paralelo, que podem ser combinadas.

Paralelismo inter-query consiste na divisão do processamento de vários pedidos por vários nós da base de dados.

Permite aumentar a velocidade das transações (compostas por várias queries).

Tira partido da **memória partilhada**. É mais complexa em abordagens com partilha de disco ou nada partilhado.

Paralelismo intra-query consiste na divisão de um query em várias partes, cujo processamento é dividido pelos vários nós.

... **intra-operação** caso cada parte do query seja processada num processador da mesma base de dados

... **inter-operação** caso cada parte do query seja processada por uma partição da base de dados

Permite aumentar a velocidade de execução de queries exigentes.

Processamento de dados em paralelo

O processamento de dados em paralelo pode ser feito de várias formas, tendo em conta o espaço de pesquisa e a estratégia necessária, procurando sempre minimizar o movimento de dados entre as máquinas.

Neste tópico a discussão é feita assumindo apenas queries de leitura (SELECT, SORT, JOIN) sobre uma base de dados com arquitetura sem partilha de recursos e com n processadores.

O **SELECT** por um atributo sobre uma base de dados com **partição horizontal** vai ter uma eficiência dependente do tipo de partição utilizado. Para uma partição **round robin** ou **hash function sobre todo o tuplo**, será necessário executar o query em todas as partições. No entanto, caso tenhamos o **range** ou a **hash function sobre a coluna** a filtrar, apenas as partições que têm registos que potencialmente cumprem os requisitos precisam de ser consultadas.

No caso de **SORT** numa base de dados com **partição horizontal** com partição **range**, como cada nó corresponde a um intervalo de valores, a ordenação é feita individualmente sobre o retorno de cada um, sendo depois o conjunto de tuplos agregado de acordo com a distribuição do intervalo pelos nós, de forma a garantir a ordenação global do resultado. É uma aplicação do algoritmo **merge-sort**.

A mesma técnica é aplicada no caso de **SORT** com **partição vertical do disco** do tipo **range**.

A ideia básica do merge-sort consiste em Dividir (o problema em vários subproblemas e resolver esses subproblemas através da recursividade) e Conquistar (após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas). [±](#)

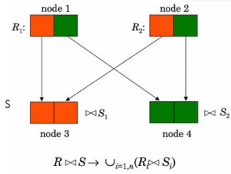
Nas operações de **JOIN** é necessário a análise dos tuplos de duas tabelas tuplo a tuplo para verificar se verificam a condição de junção e nesse caso serem retornados, o que ao nível da complexidade representa fazer o produto cartesiano entre as duas tabelas. A abordagem passa por dividir este processamento da junção das tabelas por cada um dos nós da base de dados. Há vários algoritmos que permitem implementar esta abordagem, todos eles classificados como **intra-operações**.

Neste tópico assumo-se que a operação de JOIN é realizada sobre duas relações R e S que estão particionadas em m e n nós, respetivamente.

Parallel Nested Loop (PNL)

Consiste em enviar a totalidade de R (outer relation) para cada nó, onde será comparada com o fragmento de S (inner relation).

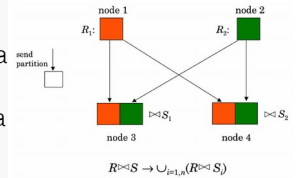
Eficiente quando R é menor que S e ainda mais quando há um índice sobre o valor da junção em S, o que permite filtrar os nós onde é necessário realizar a operação.



Parallel Associative Join (PAJ)

Aplica-se caso a base de dados de S (inner relation) seja **particionada de acordo com hash function**. Só funciona com equijoins (com base na igualdade de dois atributos).

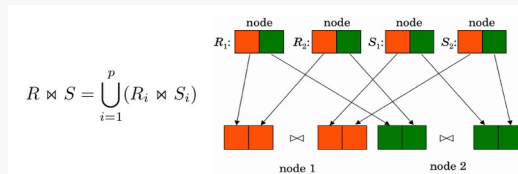
Para cada tuplo de R (outer relation) é determinada a síntese para o seu atributo de junção, sendo cada tuplo enviado para o nó de S correspondente. Localmente em cada nó de S é feita a junção com os tuplos de R recebidos.



Parallel Hash Join (PHJ)

Generalização da da abordagem anterior, para bases de dados **particionada de acordo com hash function** mas cuja chave de partição não é a mesma que a da junção.

Neste cenário, é calculada a síntese para a chave de junção para cada uma das relações S (inner relation) e R (outer relation), sendo depois agrupadas em estruturas auxiliares de acordo com o resultado da síntese e feita a junção sobre cada um destes conjuntos de dados.



Pode tirar partido do paralelismo do processamento em cada nó. É mais eficiente para partições bem distribuídas (sem skew), permitindo no cenário ideal uma velocidade $1/n$.

Na realidade o custo é de $T_{total} = T_{partição} + T_{junçãoResultados} + \max(T_0, T_1, T_n - 1)$, sendo T_i o tempo de processamento da operação em cada processador.

Gestão da carga

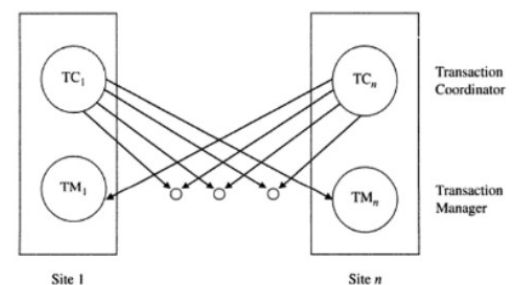
Uma boa **distribuição dos dados** é fundamental para potenciar a eficiência das respostas da base de dados a queries. Para tal, devem ser aplicados **algoritmos eficazes que lidem com desvios** (skew) e eventualmente **alocações dinâmicas em tempo de execução** capazes de lidar com alterações na topologia da base de dados.

Os erros mais comuns são por exemplo quando o conjunto de dados é particionado de acordo com atributo pouco variável.

É ainda fundamental gerir a **distribuição do trabalho** de acordo com algoritmos como o round robin ou o least loaded.

Gestão de transações distribuídas

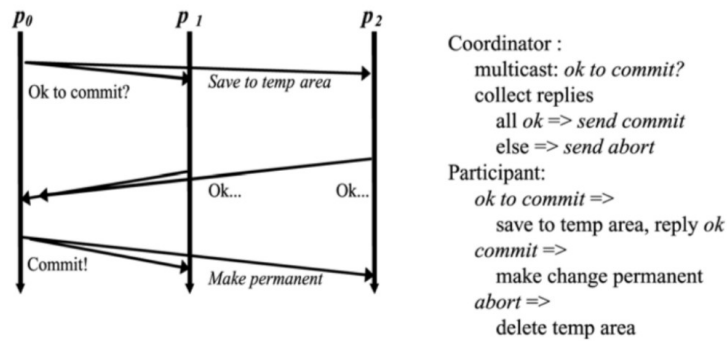
De forma a lidar com transações que têm de aceder a várias máquinas, os sistemas de bases de dados distribuídos apresentam um **coordenador de transações**, responsável por dividir a transação, distribuí-la pelos vários nós e terminá-la, e



um **gestor de transações**, que em cada nó é responsável por executar as transações pedidas pelo anterior e controlo de concorrência do seu nó.

Um dos principais problemas desta gestão são os **commits** que devem atómicos em todas as bases de dados!

Uma solução é a **two-phase commit**, que divide o commit, como o nome sugere, em duas fases: uma primeira em que o coordenador ausculta os participantes sobre o commit ou abort da transação e uma segunda em que recolhidos os votos toda uma decisão e comunica-la, de forma a tornar o commit efetivo em todos os nós ou abortá-lo.



10. Replicação de bases de dados

Slides teóricos

Em bases de dados distribuídas, de forma a **diminuir a latência**, **aumentar a disponibilidade** e **potenciar a escalabilidade** do sistema, é necessário **manter uma cópia dos dados em múltiplas máquinas**, ou seja, **replicá-los**.

Ao distribuirmos um sistema de forma uniforme na geografia que deve servir, em detrimento de uma solução centralizada, vai ser **reduzida a latência** média de comunicação entre os clientes e o sistema.

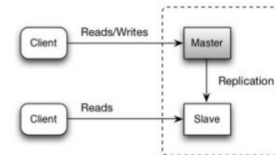
O **aumento da disponibilidade** deve-se à redundância introduzida pela replicação, uma vez que se um nó falhar haverá outros disponíveis para responder aos clientes, pelo que o sistema não fica (totalmente) inoperacional.

Ao aumentar o número de máquinas no sistema com a mesma informação este torna-se capaz de responder a mais pedidos em simultâneo, pelo que se diz **escalável**.

O desafio passa por **gerir alterações nos dados de forma a garantir a sua consistência**, para o qual há várias abordagens possíveis que variam em dois eixos, a **liderança** e o **sincronismo** e podem ainda apresentar variações quanto à **gestão da concorrência** e **replicações falhadas**.

Replicação single-leader

Esta abordagem **distingue os nós em líder e seguidores**, sendo o **primeiro utilizador para os clientes para operações de leitura e escrita** e os segundos apenas para leituras.

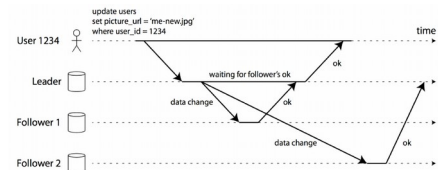


Quando recebe um pedido de escrita, o líder escreve as alterações localmente e comunica-las a todos os seguidores, que por sua vez atualizam o conteúdo da sua base de dados.

Sincronismo

Numa abordagem **síncrona**, o **líder espera pela confirmação de todos os seguidores antes de retornar o sucesso da operação ao cliente**.

Na **assíncrona**, apenas o resultado da sua operação local interessa, pelo que **após a escrita retorna imediatamente sucesso ao cliente**, antes de ter qualquer resposta dos seguidores.



Na figura podemos ver uma abordagem **síncrona** do líder quanto ao Follower 1 (espera pela sua resposta para retornar ao cliente) e uma **assíncrona** quanto ao Follower 2 (do qual não espera qualquer resposta).

Na maioria dos casos a abordagem **síncrona** é exequível porque a **comunicação das alterações aos seguidores demora apenas em frações de segundo**. Garante ainda a **consistência da base de dados**, uma vez que **se o líder falhar, não haverá informação perdida uma vez que os seguidores estão sempre atualizado**.

No entanto, caso algum seguidor esteja com **overhead**, a **recuperar de uma falha** ou com **falhas no acesso à rede**, pode demorar alguns minutos a responder, levando a que seja possível que **escritas sejam impedidas por falhas relativamente comuns nos sistemas**.

Assim, como **na prática** se torna impossível ter todos os nós síncronos, a solução adotada geralmente passa por ter uma réplica síncrona e os restantes assíncronos. Temos então duas abordagens opostas amplamente utilizadas hoje em dia.

Semi-síncrona O líder trabalha com um seguidor síncrono e os restantes assíncronos. Caso o síncrono falhe, outro seguidor será eleito síncrono. Garante-se uma cópia consistente dos dados em pelo menos dois nós.

Totalmente assíncrona Nesta solução todos os seguidores são assíncronos. As escritas não são garantidas, pois mesmo após confirmadas ao cliente pelo líder podem ser perdidas caso este falhe antes de as comunicar aos seguidores.

Novos seguidores

Ao adicionar uma nova réplica à base de dados é necessário que esta obtenha o seu conteúdo. Para tal, o **líder faz um snapshot da base de dados que envia à nova réplica**. Quando a recebe e popula a sua base de dados, o **seguidor faz o pedido ao líder de todas as alterações que foram feitas desde o momento do snapshot**, passando a estar consistente com as restantes réplicas a partir do momento em que as recebe.

Tal não é possível copiando apenas os dados, porque durante a cópia podem ser feitas escritas e torná-la inconsistente. Assim, de forma a evitar o bloqueio da base de dados a escritas, é desencadeado o algoritmo descrito.

Saídas de seguidores

É possível que qualquer uma das réplicas falhe devido a falhas inesperadas ou mesmo manutenções agendadas.

No caso de ser um **seguidor**, este mantém um log de alterações aos dados, que consulta quando voltar ao ativo para **pedir ao líder todas as alterações feitas desde a última que este recebeu**. Este processo é denominado por **catch-up**.

Se for o **líder**, é necessário **eleger um novo, reconfigurar os clientes para escreverem nele e os restantes seguidores para consumirem dele**. Isto é feito pelo processo **failover**, que pode ser desencadeado manualmente ou de forma automática.

Na abordagem automática, é assumido que o líder falhou caso não dê sinais de vida num determinado timeout, após o qual é eleito o novo líder que é geralmente o que tem uma versão da base de dados mais atual. Uma vez eleito, o sistema é reconfigurado e os clientes devem passar a utilizar o novo líder, geralmente através de um sistema de roteamento.

É ainda fundamental que se o líder antigo voltar ao ativo não é utilizado pelos clientes para escrita!

A assunção de que o líder falhou não é 100% segura, uma vez que o timeout se pode dar a falhas de rede ou outras e na realidade este continuar a funcionar. Um timeout muito grande vai levar a que o sistema demore muito tempo a voltar ao ativo. No entanto um muito curto pode despoletar reeleições desnecessárias.

Logs de replicação

Os logs de replicação mantêm a **informação das escritas na base de dados** (operações de inserção, atualização e remoção). Há quatro métodos para os manter.

Statement-based replication O líder, sempre que executa um comando envia-lo para as réplicas, que o executam localmente

Gera problemas na utilização de funções não determinísticas como RAND() ou NOW(). Não há garantia que as instruções sejam executadas pela mesma ordem em todas as réplicas nem dos efeitos de triggers associados às tabelas manipuladas, o que pode gerar problemas de consistência.

Write-ahead log (WAL) Sempre que recebe um query, antes da sua execução escreve-lo para um append-only-log e envia-lo às réplicas.

O log descreve os dados a um baixo nível, o que torna a replicação dos dados e a atualização do sistema altamente dependente da estrutura de armazenamento do sistema de base de dados, que pode diferir de acordo com a versão. Líder e réplicas estão sujeitos a utilizar a mesma versão.

Logical log replication oferece uma alternativa ao anterior, com os dados a serem armazenados de forma independente da representação interna do sistema de armazenamento da base de dados, através de uma representação lógica.

Isto permite que as várias réplicas utilizem diferentes versões do sistema de base de dados e até diferentes estruturas de armazenamento dos dados. Facilita também a atualização dos nós sem necessidade de fazer a conversão dos logs.

Trigger-based replication é uma replicação ao nível da camada da aplicação. Oferece mais flexibilidade. Pode ser implementado através da leitura dos logs ou através de funcionalidades da BD como triggers e stored procedures.

Contrariamente às anteriores, que eram implementadas ao nível do sistema de base de dados, neste caso é implementado por código aplicacional.

Utilizada por exemplo para replicar apenas um subconjunto dos dados, ou fazer conversões ou mapeamentos entre os dados originais e as réplicas.

Lag na replicação

A **replicação** pode parecer a solução para todos os problemas de escalabilidade das bases de dados, mas na realidade traz consigo um novo: o **lag^o na replicação**, que pode levar a que leituras em réplicas assíncronas retornem dados desatualizados.

Por este motivo não se diz haver consistência, mas sim **consistência eventual**, uma vez que as inconsistências da base de dados são possíveis, mas são apenas um estado temporário, uma vez que as réplicas irão estar sincronizadas com o líder eventualmente no tempo.

Há no entanto algumas garantias que devem ser asseguradas.

Read-after-write consistency Um utilizador deve conseguir ver os dados que escreveu na base de dados.

A sua implementação pode consistir no cliente ler do líder todos os dados que pode ter potencialmente alterado e os restantes de qualquer réplica.

No entanto, caso o cliente possa ter alterado potencialmente a totalidade (ou quase), não será uma solução eficiente. Nestes casos pode ser mantido um registo dos dados atualizados nos últimos X minutos e apenas esses são lidos diretamente do líder.

Monoatomic reads Uma vez lido um determinado valor da base de dados, leituras subsequentes nunca devem retornar um valor mais antigo do que o lido anteriormente.

A sua implementação consiste em garantir que cada cliente faz as suas leituras sempre na mesma réplica. Clientes diferentes podem ler de réplicas diferentes.

Consistent prefix reads Escritas ligadas causisticamente devem ser lidas na mesma ordem.

Um exemplo de duas escritas ligadas causisticamente são uma pergunta e a sua resposta.

Replicação multi-leader

Segundo esta abordagem as **escritas podem ser feitas em mais do que uma réplica**, que partilham a liderança. Consiste numa extensão da single-leader, na medida em que cada escrita é encaminhada pelo nó que a fez para os restantes.

Assim, cada réplica vai desempenhar simultâneamente os papéis de líder e seguidor.

Vem responder ao problema do single-leader que era a indisponibilidade para escritas em caso de falha do líder. É utilizado em clientes que operem offline, operações entre data-centers e edição colaborativa. Não é recomendado a data-centers únicos (internamente).

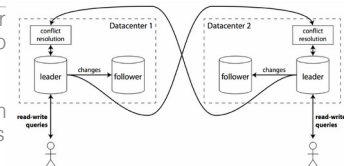
Cenários de utilização

Pode ser utilizado em vários cenários.

Multi-datacenter operation Em datacenters distribuídos pode haver um líder em cada um, que em rede com os restantes líderes aplica uma replicação multi-leader e com as réplicas internas uma replicação single-leader.

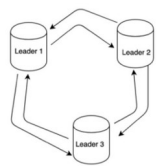
Esta abordagem permite uma boa performance, tolerância a falhas porque um datacenter pode operar de forma independente dos restantes e tolerância a falhas de rede, porque não impede o processamento das escritas.

Pode gerar problemas no caso de alterações concorrentes aos mesmos dados em datacenters diferentes. Funções não determinísticas e triggers e stored procedures podem ser um problema. Esta solução deve ser evitada.



Clientes com operação offline Cada cliente tem uma réplica da base de dados que atua como líder em caso de ficar offline.

É o cenário anterior levado a um extremo. Quando a conexão à rede é retomada, os dados devem ser sincronizados com as restantes réplicas. Utilizado quando os clientes precisam de operar offline ou a sua conexão à rede é muito instável.



Edição colaborativa permite que múltiplos clientes editem o mesmo recurso em simultâneo. As alterações são aplicadas instantâneamente na réplica local e comunicadas assíncronamente às restantes.

Em caso de edição colaborativa de documentos e de forma a evitar conflitos na edição, a aplicação pode obter um lock do documento antes de o alterar.

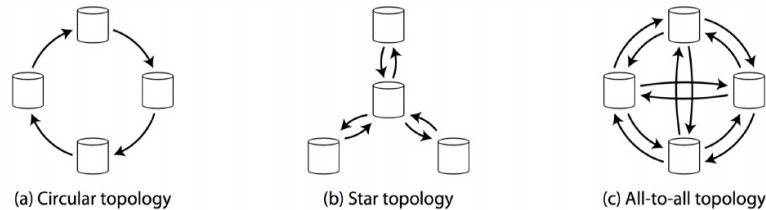
No entanto, este procedimento leva a perdas de tempo. Para uma colaboração mais rápida, as alterações podem ser constantemente comunicadas em unidades mais pequenas (por exemplo em vez de comunicar uma palavra comunicar letra a letra, assim que inserida).

Mesmo assim, podem haver conflitos uma vez que em introduções concorrentes em réplicas diferentes ambas assumirão o papel de líderes e não vão conseguir chegar a um consenso.

A resolução de conflitos pode ter em conta o timestamp em que os dados foram alterados (last write wins), dando preferência aos mais recentes, ou mesmo ao nível aplicacional, questionando o utilizador de o conflito ocorreu e dando-lhe a escolher qual quer manter.

Técnicas de replicação

A topologia da replicação descreve a forma **como as comunicações se propagam entre nós**.



A all-to-all é a mais generalizada, mas a que consome maior largura de banda.

Replicação leaderless

Esta é uma abordagem em que **não há líderes**, sendo as **escritas aceites por qualquer réplica**.

Pode ser implementada de forma a que o cliente envie os queries para as várias réplicas, ou definindo um **coordenador**, que encaminha as escritas para as restantes réplicas.

Contrariamente ao single-leader, este coordenador não estabelece a ordem das escritas, limitando-se a encaminhá-las.

Recuperação de falhas

Devido à ausência de restrições e coordenação, caso um nó falhe, quando voltar ao ativo estará inconsistente com as restantes réplicas. Podem ser utilizados dois mecanismos para o corrigir:

Read repair O cliente lê de várias réplicas e ao detetar que uma está inconsistente com as restantes e tem um valor desatualizado, faz um pedido de escrita para o corrigir.

Anti-entropy¹⁰ process As bases de dados podem ter um processo a correr em segundo plano que está em constante procura por diferenças entre as várias réplicas e corrige-las.

Sem o **anti-entropy process**, valores que são lidos com pouca frequência podem ficar em falta em algumas réplicas e por isso ter pouca durabilidade.

¹⁰ Entropia | Medida de desordem de um sistema.

Quórum para leituras e escritas

De forma a potenciar a **consistência** da base de dados, **deve ser obtido quórum¹¹ antes de ser realizada uma operação de leitura ou escrita**, respetivamente de w e r , de forma a que...

$$w + r > n, n \text{ número de réplicas}$$

Geralmente w e r são números ímpares para poder haver desempate. Devem ser atribuídos de acordo com o contexto para o qual a base de dados vai ser utilizada.

Por exemplo uma base de dados de leitura intensiva poderá ter um $r=1$ e um $w=n$. As leituras serão assim garantidamente consistentes e altamente rápidas. No entanto, basta que um nó falhe para impedir as escritas.

De forma a potenciar a velocidade e disponibilidade do sistema o quórum pode ser ignorado e serem definidos valores de w e r tal que $w + r \leq n$. Neste caso passa a haver a possibilidade de serem lidos valores inconsistentes (tanto maior quanto menores w e r).

Há ainda uma outra abordagem direcionada a bases de dados de grandes dimensões com um grande número de réplicas, o **sloopy¹² quórum**. Esta tem por base o **quórum**, mas dado o elevado número de réplicas e a maior propensão a erros, em vez de exigir um consenso de n nós, sendo n o número total de réplicas, **considera um subconjunto das réplicas como sendo os responsáveis por armazenar determinado valor**, designados por **"home" nodes**.

Assim, sempre que é necessário consenso de w ou r réplicas e caso alguma não esteja a responder, **uma das réplicas que não é "home" node para esse valor pode responder em vez dela**. Uma vez que o nó "home" em falha volte ao ativo, as escritas aceites pelo nó "home temporário" serão transmitidas ao "oficial". A este processo chama-se **hinted handoff**.

Cenários de utilização

A abordagem **leaderless** também pode ser aplicada a datacenters distribuídos, sendo cada escrita comunicada a todas as réplicas, independentemente do datacenter.

No entanto, de forma a garantir uma resposta célere ao cliente, geralmente o quórum limita-se ao datacenter local, sendo as restantes escritas comunicadas de forma assíncrona aos outros datacenters.

¹¹ Número necessário de membros para que uma assembleia possa funcionar.

¹² Descuidado. Desleixado.

Gestão de escritas concorrentes

Caso dois clientes escrevam valores diferentes em réplicas diferentes da base de dados, vai gerar-se uma situação de concorrência. Há quatro abordagens possíveis para a resolver.

Last write wins (LWW) Associar um timestamp a cada escrita, sendo dada preferência aos mais recentes

Keys with version number Manter um número de versão incremental para cada chave. Cada vez que é feito um pedido de escrita deve ser enviado o número de versão e caso este entre em conflito com algum existente a escrita é abortada. O cliente deve fazer uma leitura antes da escrita.

Merging concurrently written values Garante que nenhum valor é descartado silenciosamente. Pode usar uma abordagem como as anteriores, ou marcar o valor como eliminado sem o eliminar (no delete data).

Version vectors Manter um número de versão por réplica e por chave em cada réplica, conjunto denominado por **vetor de versões**. Garante que é seguro escrever se antes for feita uma leitura.

11. Partição de bases de dados

Slides teóricos

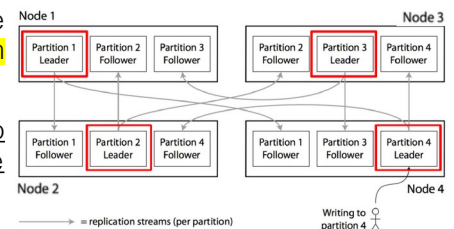
Para trabalhar com grandes conjuntos/fluxos de dados muitas vezes temos de os **partir em subconjuntos mais pequenos**, chamados **partições**, que são **atribuídos a diferentes nós**. Este processo é conhecido por **sharding**¹³.

Geralmente, cada pedaço dos dados é mapeado numa única partição.

A grande vantagem desta abordagem é a **escalabilidade**, uma vez que pode dividir grandes volumes de dados e de carga de trabalho por vários nós e com isto aumentar capacidade de resposta.

Geralmente está associada à **replicação** por todos os benefícios que esta oferece, através do **armazenamento da mesma partição em vários nós**, que por sua vez podem armazenar várias partições.

No modelo de replicação **lider-follower** (single-leader), cada partição terá um nó líder, sendo os restantes nós seguidores, sem prejuízo de diferentes nós poderem ser líderes de diferentes partições.



Tipos de partição

A partição dos dados tem como objeto a **distribuição uniforme** dos dados pelos nós e pode ser feita de forma **horizontal**, caso seja **feita pelas linhas**, ou **vertical**, caso seja **feita pelas colunas**.

No entanto, se não forem aplicados algoritmos adequados para gerir a distribuição dos dados, algumas partições podem ficar **skewed** caso tenham um **excesso de dados ou pedidos em relação às restantes**, ou mesmo tornar-se num **hot spot** caso o **esforço seja concentrado numa única**.

Partição de dados chave-valor

Os dados estruturados desta forma são accedidos pela sua chave primária, podendo a partição ser feita de acordo pelo **intervalo ou pela síntese das chaves primárias**.

No caso da partição por **intervalos**, estes podem ser determinados de forma manual ou automática e permitem determinar facilmente qual o nó responsável pela chave. Pode ainda haver ordenação em cada partição, permitindo queries de range. As chaves podem ser simples ou complexas (concatenação de dados).

As enciclopédias são um exemplo de uma partição de dados estruturados em chave-valor com base em intervalos.

Não deve ser esquecido que as chaves não são necessariamente espaçadas uniformemente no seu intervalo, pelo que uma configuração manual mal feita pode levar facilmente à criação de hot spots.

A partição por **síntese** (hash) da chave primária reduz os riscos de skew e hot spot quando implementado com uma boa função de síntese. No entanto, perde-se eficiência na pesquisa por intervalos, uma vez que deixa de haver ordenação.

¹³ "Shard of glass" é um caco de vidro.

Apesar de parecer uma implementação próxima do ótimo, na realidade há um cenário em que pode criar um hot spot, caso haja um grande fluxo de queries com vista a uma única chave. Por exemplo numa pesquisa por uma celebridade envolvida num escândalo.

Partição e índices secundários

O acesso aos dados através do valor da chave primária revelou-se bastante trivial. No entanto, os índices secundários vêm trazer problemas à organização dos dados. Há duas abordagens principais que respondem a este problema.

Índice particionado em documentos

Esta abordagem mantém um índice por partição, relativa aos dados que armazena localmente.

A **leitura** de dados com filtragem por um índice secundário implica o envio do query a todas as partições e combinar os resultados retornados.

A **escrita**, **remoção** e **atualização** são feitas diretamente e apenas com o nó que tem o documento com ID que queremos manipular.

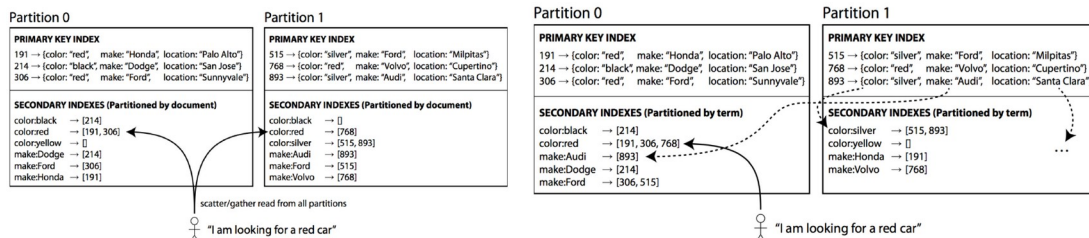
Índice baseado em termos

Contrariamente à anterior, esta solução consiste num índice global que cobre os dados em todas as partições. No entanto, este índice não está armazenado numa única partição, mas sharded (distribuído) pelas várias partições, de acordo com uma chave diferente da chave primária dos dados.

Diz-se partição baseada em termos, porque o termo por que se pretende pesquisar determina a partição na qual o índice é consultado.

A **leitura** com filtragem por um índice secundário é potencialmente mais rápida, uma vez que basta consultar o índice numa partição, em vez de ter de fazer pedidos a todas.

As **escritas** são no entanto mais demoradas, uma vez que é necessário atualizar os índices em várias partições, dependendo do número de atributos que estes cobrem.



Equilíbrio das partições

De forma a garantir resposta ao aumento do número de pedidos é necessário aumentar a capacidade de processamento ou no caso de aumento do volume de dados a capacidade do disco e da RAM e até transferir as responsabilidades de uma máquina em caso de falhas.

Em qualquer um destes casos é necessário **mover dados entre nós**, fazendo a sua **redistribuição** pelas várias partições disponíveis.

Embora algo complexa, esta operação não deve interferir com a disponibilidade do sistema e garantir que com o mínimo de movimentações de dados consegue maximizar a uniformização da sua distribuição.

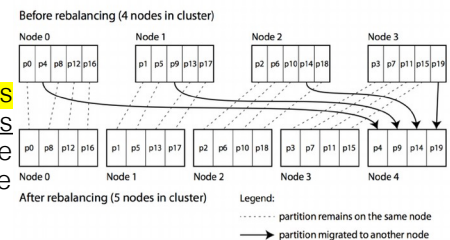
Pode ser feita de forma **automática** ou **manualmente**. A primeira é mais conveniente, uma vez que não necessita de intervenção humana. No entanto, pode ter resultados imprevisíveis em caso de erros que devem ser acompanhados, pelo que deve ser sempre um processo controlado por humanos.

Round-robin

Esta estratégia, que consiste na **distribuição uniforme das chaves pelos nós** **não deve ser utilizada**, uma vez que sempre que o número de nós é alterado há uma grande quantidade de dados que necessita de ser movimentada.

Número fixo de partições

Ao considerar um **número fixo de partições por síntese com mais partições do que nós**. Assim, a gestão consiste em atribuir as partições aos nós, movendo-las integralmente entre eles sempre que necessário. A adição de um novo nó à rede levará à transferência de algumas partições de outros nós para o novo e vice-versa.



Por exemplo para 10 nós podem ser criadas 1000 partições, sendo inicialmente atribuídos 100 a cada um. Assim, mesmo que hajam variações na carga de trabalho ou volume de dados exigido a cada nó, as partições podem ser facilmente redistribuídas por outros nós, sem necessidade de reestruturas as partições, uma vez que estas já são bastante esparsas¹⁴.

Partição dinâmica

Esta é uma abordagem que **cria partições de forma dinâmica para dados organizados em intervalo ou em síntese**, em resposta ao crescimento ou falta dele para cada partição.

Assim, quando uma partição excede um determinado tamanho é dividida em duas e quando o oposto acontece, ou seja, quando muitos dados são removidos de determina partição, esta pode ser fundida (merged) com uma contígua.

Tem como principal **vantagem** a resposta a alterações dinâmicas na topologia dos dados. No entanto, tem a **desvantagem** que caso o volume de dados seja pequeno vai concentrar o trabalho num único nó.

¹⁴ Espalhado. Solto.

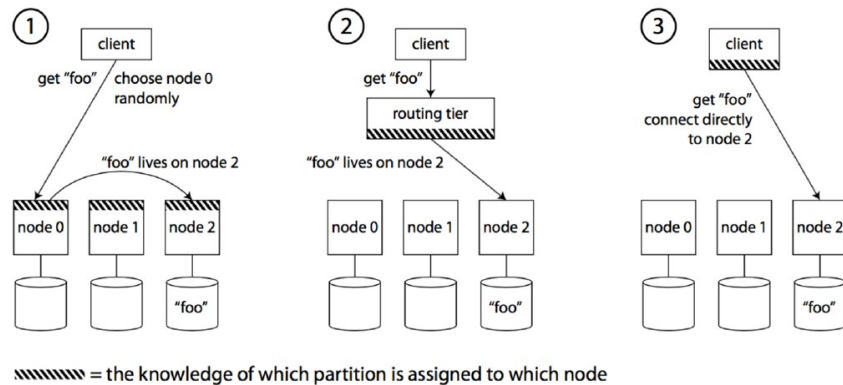
Processo de roteamento

De forma a **saber que nó deve ser contactado para fazer um pedido**, o cliente pode recorrer a três soluções...

Contactar qualquer um dos nós que devem manter um índice global de forma a encaminhá-lo para o nó correto.

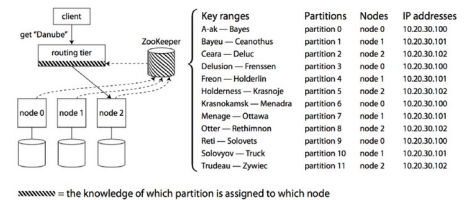
Manter registo da atribuição das partições aos nós conectando-se diretamente ao nó pretendido, sem necessidade de encaminhamento.

Enviar todos os pedidos para um routing tier first que encaminha o pedido para o nó correspondente.



O consenso nestes sistemas distribuídos é difícil de atingir e de forma a **garantir que todos têm a mesma informação quanto à distribuição das partições pelos vários nós** muitos destes sistemas implementam ainda um **Zoo-Keeper**.

Este assume o papel do mapeamento da topologia da base de dados, sendo nele que os nós se registam e que o routing tier subscreve pelas alterações, sendo notificado sempre que estas acontecem, de forma a manter a informação atualizada e poder fazer um encaminhamento correto dos pedidos dos clientes.



O Hbase e o Kafka usam o Apache Zoo-Keeper para manter a informação atualizada quanto à atribuição das partições. O MongoDB utiliza uma arquitetura similar.

A Cassandra já dispensa esta abordagem e recorre a um protocolo de gossip para disseminar e obter consenso nas alterações da distribuição das partições, pelo que os pedidos do cliente podem ser enviados para qualquer nó, uma vez que este será capaz de fazer o encaminhamento.

Há mais complexidade no funcionamento, mas reduz a dependência de um serviço externo.

12. Processamento de dados distribuídos

Slides teóricos e aula assíncrona

Nos últimos anos temos assistido a um crescimento exponencial da quantidade de dados gerada e a tendência não tende para a sua estagnação. No entanto, a capacidade de armazenamento centralizada não acompanhou esta evolução e mesmo apesar do crescimento vertical ser possível, tem limites inferiores à capacidade necessária.

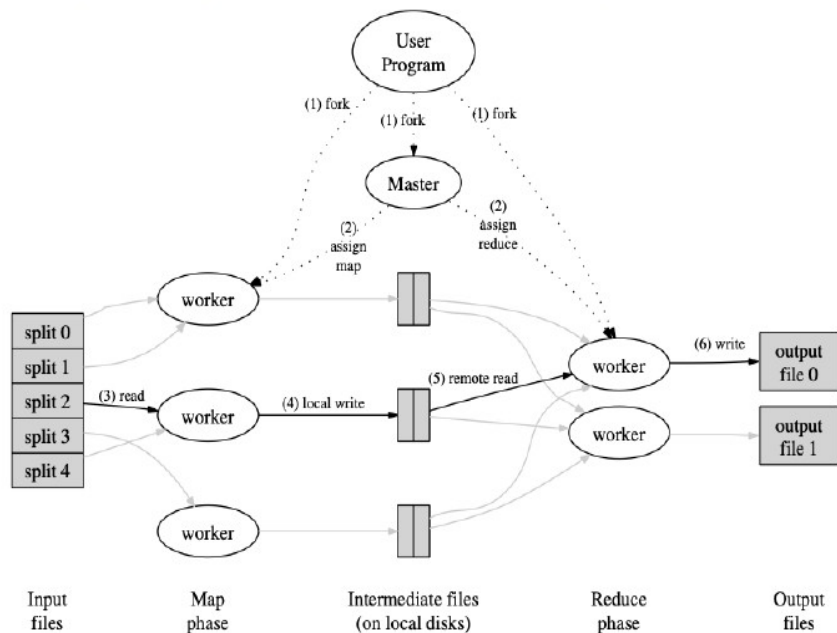
Assim, em cenários que produzem uma quantidade elevada de dados e/ou precisam de uma grande capacidade de processamento recorre-se à **distribuição do processamento dos dados**, onde os dados são distribuídos e os algoritmos vão ter com eles.

Este paradigma foi proposto por engenheiros da Google com o **modelo de programação MapReduce**, uma implementação para **processar e gerir grandes quantidades de dados**.

Segundo esta, o processamento dos dados é dividido por vários workers, que fazem o mapeamento dos dados em cada nó, através do seu carregamento, análise eventuais transformação e ordenação, que são depois agrupados e agregados na redução.

Por exemplo numa lista de números do tipo String podemos mapear o valor de cada um para um inteiro e depois fazer a redução com a soma destes números.

Em Java: `Values.stream.map(num → Integer.parseInt(num)).reduce(0, Integer::sum);`



Frameworks MapReduce

O paradigma de programação deu origem a várias **frameworks**, ferramentas ou **bibliotecas** que **facilitam a escrita de aplicações para o processamento de grandes quantidades de dados distribuídas em paralelo**, agregando um conjunto de soluções amplamente utilizadas na indústria.

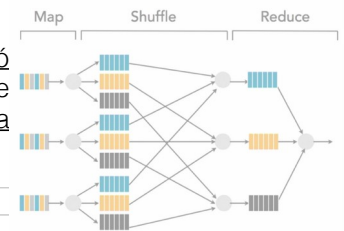
Tal como as restantes frameworks, para além das funcionalidades base oferece suporte para um conjunto de adicionais como por exemplo a gestão de erros e tolerância a falhas, mecanismos automáticos de paralelização e distribuição, agendamento de I/O e monitorização, que permite um acompanhamento do estado das tarefas.

Podem ser utilizadas para contagem de palavras, índices invertidos (número de ocorrências de palavras nas páginas web), pesquisas ou ordenações distribuídas.

As mais conhecidas são a **Hadoop**, que implementa o mapper e o reducer em classes Java que podem ser extendidas e o **Spark**, mais recente e com base no anterior, mas com melhorias no processamento dos dados, que é feito em RAM, pelo que é mais rápido.

O **fluxo dos dados** inicia-se com o **mapeamento** de cada registo para cada nó da base de dados, que são de seguida agregados pela chave e eventualmente ordenados através do processo de **shuffle** e por fim **reduzidos para cada conjunto de valores com a mesma chave**.

A redução pode consistir por exemplo em contar o número de ocorrências para cada chave.



Apache Hadoop

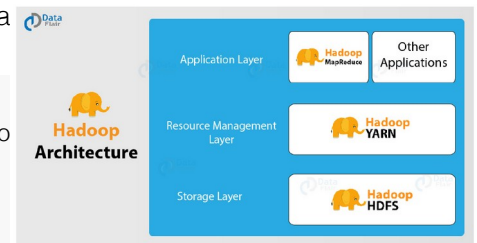
Esta é uma framework **open-source** implementada em **Java**, composta por vários componentes.

Hadoop Distributed File System (HDFS) Sistema de ficheiros distribuídos.

Hadoop Yet Another Resource Navigation (YARN) Sistema operativo sobre o sistema de ficheiros, que controla e aloca os recursos.

Hadoop Common Utilities Utilitários.

Hadoop MapReduce Implementação do modelo de programação.



HDFS

O **HDFS** é um **sistema de ficheiros partilhado** desenvolvido para correr em **hardware básico** de forma a suportar a **escalabilidade horizontal** que implementa uma **arquitetura líder-seguidores**. É também **tolerante a falhas**, armazenando os ficheiros em **blocos de tamanho fixo (128MB)** com **pelo menos 3 réplicas para cada um**. É utilizado para a gestão de ficheiros de grandes dimensões, cuja **atualização é rara**, mas **appends e leituras são frequentes**.

Tem por base um conjunto de permissas:

Falhas do hardware O facto de ser baseado em hardware simples e pouco robusto aumenta a probabilidade de falhas.

Acesso aos dados em streaming Tem foco em processamento de grandes quantidades de dados (batch) em detrimento de processamento transacional.

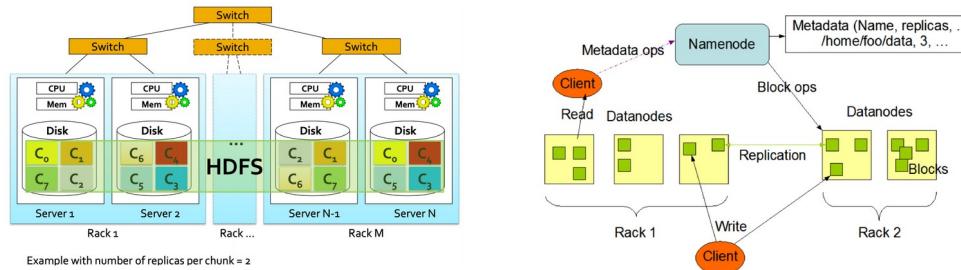
É dado privilégio à capacidade de troca de grandes quantidades de dados em detrimento da baixa latência.

Conjuntos de dados de grande dimensão Na ordem dos GB e mesmo TB.

Modelo de consistência simples Uma vez criado, um ficheiro só é alterado para fazer append e truncate.

Movimentação da computação em vez dos dados Levar os algoritmos até aos dados é mais eficiente do que o contrário, dado o grande volume de que se trata.

Portabilidade sobre um conjunto heterogéneo de plataformas Os nós podem ser heterogéneos quanto ao hardware e software.



A sua **arquitetura líder-seguidores** tem como nó principal o **NameNode**, que guarda os metadados do sistema de ficheiros, **mapeando o nome do ficheiro para a sua localização**, que será num **DataNode**.

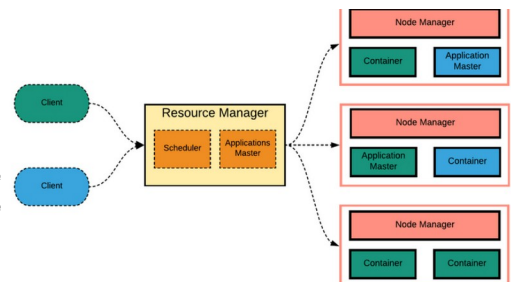
O **NameNode** é como uma FAT (File Allocation Table).

Os clientes consultam o **NameNode** para saber a localização do ficheiro a que querem aceder, ligando-se depois diretamente ao **DataNode** correspondente.

YARN

O **YARN** é um **sistema operativo sobre o HDFS** que gere e controla os seus recursos e a sua alocação.

É responsável pela alocação das tarefas pedidas ao **resource manager** pelos clientes aos vários nós geridos pelo **node manager**. Esta atribuição é mediada pelo **applications master**.



Hadoop MapReduce

O último componente é a camada responsável por fornecer livrarias para processar dados. O processamento é feito sobre partições dos dados em cada nó, de forma a distribuir a carga de trabalho.