

Complementos de Bases de Dados

José Mendes 107188

2023



1 Evolução dos Sistemas de Base de Dados

Sistemas de Dados - Cada vez mais as aplicações de hoje em dia são Data-Intensive, em vez de Compute-Intensive.

Para Data-Intensive, o poder bruto da CPU deixa de ser um fator limitante 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:

1. **Bases de Dados** - armazenam os dados para utilização futura;
2. **Caches** - guardam os resultados de operações dispendiosas, de forma a tornar a leitura mais rápida;
3. **Search Indexes** - permitem aos utilizadores procurarem por palavras-chave ou filtrar os dados;
4. **Message Queues** - permitem a comunicação assíncrona entre processos;
5. **Stream Processing** - permite o processamento de dados em tempo real;
6. **Batch Processing** - permite o processamento de dados acumulados, periodicamente;

Exemplo: 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.

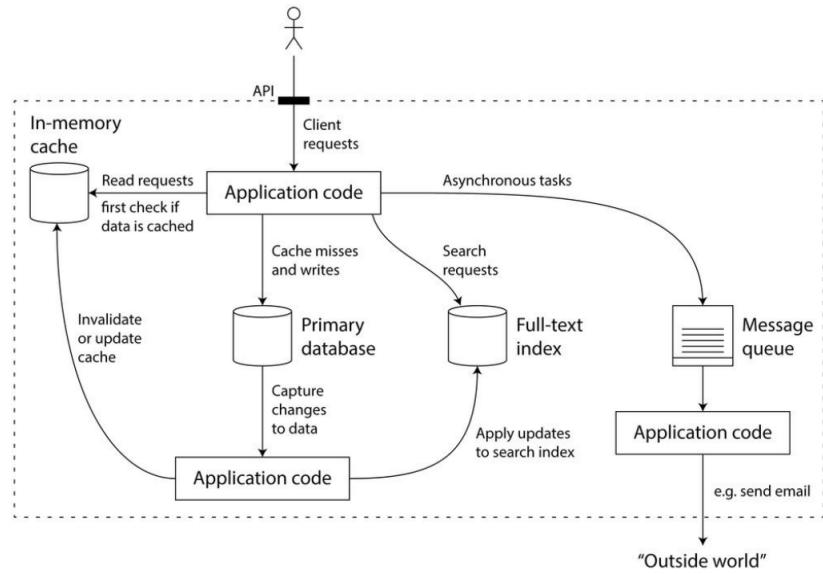
O **batch processing** é visível na faturação dos serviços pós-pagos pelas operadoras de telecomunicações. No final de cada mês, é feita uma consulta às suas bases de dados de forma a identificar todos os consumos do cliente, que são somados e depois gerada a fatura.

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

Cada vez mais as aplicações requerem um **maior wide-range de requisitos**. Muitas das vezes, uma única ferramenta já não consegue satisfazer todas as necessidades de data processing e storage.

Em vez disso, o trabalho é partido em tasks que possam ser realizadas de forma eficiente por uma única ferramenta. As ferramentas individuais utilizadas são depois juntas utilizando código de aplicação.

Exemplo: Podemos ter uma aplicação que utiliza uma Catching Layer (**memcached**), um Full-Text Search (**Elasticsearch**) e uma Base de Dados principal separada (**MySQL**).



1.1 Desafios que os Sistemas de Dados enfrentam

Como garantir que todos os dados se mantêm corretos e consistentes, mesmo quando, internamente, ocorreu algum erro? (ex: persistência de dados)

Como fornecer boa performance para os clientes, mesmo quando partes do sistema estão degradadas?

Como escalar o sistema para ser capaz de aguentar uma load intensiva de trabalho?

Qual a aparência de uma boa API para o serviço?

1.2 Alguns Requisitos

Fiabilidade - O Sistema deve continuar a funcionar corretamente em caso de adversidades (ex: falhas de hardware, software ou mesmo humanas).

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

Manutenibilidade - Deve ser possível que o Sistema sofra alterações ao longo do tempo por várias pessoas diferentes de forma produtiva.

1.3 Bases de Dados

São definidas como um conjunto de dados relacionados entre si e a sua organização. Dividem-se em vários tipos, sendo atualmente os mais comuns: **Relacionais**, seguidas por **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 NoSQL Databases - Key-Value Databases

1. É o mais simples dos tipos NoSQL;
Consiste apenas em chaves únicas e a um "bucket" que contém qualquer tipo de dados que se pretenda;
2. Pares chave-valor:
Chave: (id, identificador, chave primária) Normalmente é uma String;
Valor: Pode ser qualquer tipo de dados, texto, estrutura, imagem ...;
3. O conteúdo do valor ("bucket") pode ser, literalmente, qualquer coisa (mais comum é não estruturado ou semi-estruturado);
4. Os "buckets" podem armazenar entradas pesadas, incluindo BLOBs (Basic Large Objects);
5. Row based systems, utilizados para eficiência;

2.1 Vantagens

1. Tolerância a falhas elevada - sempre disponível;
2. Schemaless, logo, muito flexível. oferece uma grande escalabilidade para mudar os requisitos dos dados;
3. Eficiente a devolver dados de um objeto, com operações de disco minimas;
4. Muito simples, rápido e fácil de dar deploy;
5. Ótimo para escalabilidade horizontal (muitos servidores);
6. Não necessita de queries SQL, indexes, triggers, sp's, views, ...;
7. Data ingest rates muito elevadas (muitos dados a entrar);
Favorece: escreve uma vez, lê muitas vezes;
8. Potente no "offline reporting" com data sets muito grandes;
9. Existem formas avançadas de KVs que apresentam capacidades de document ou column oriented stores;

2.2 Desvantagens

1. Não é apropriado para aplicação complexas;
2. Não é eficiente a atualizar records em que apenas parte do "bucket" é alterado;
3. Não é eficiente em devolver informação limitada de records específicos (ex: returning only records of employees making between \$40K and \$60K);
4. Não é apropriado para queries complexas;
5. Com o aumento do volume de dados, manter chaves únicas pode tornar-se um problema;
6. Geralmente precisa de ler todos os records de um "bucket" ou talvez precise de construir índices secundários;

2.3 Use Cases

1. Session data, user profiles, user preferences, shopping carts, ...;
2. Criar datasets que são raramente acessados mas crescem ao longo do tempo (Caching);
3. Onde a performance de escrita é a prioridade;

2.4 Quando NÃO usar

1. Quando precisamos de ter relações entre entidades;
2. Queries requerem acesso a conteúdos da parte dos valores;
3. Set operations que envolvem múltiplos pares chave-valor;

2.5 Key Management

Como devem as chaves serem produzidas?

Manually assigned keys - Identificadores do mundo real (ex: e-mail, login names, ...);

Automatically generated keys - Auto-increment integers ou chaves mais complexas geradas por algoritmos;

2.6 Query Patterns

1. Basic **CRUD** operations;

Apenas quando a chave for dada;

O conhecimento da chave é essencial;

Ás vezes, pode ser até difícil para uma base de dados dar uma lista com todas as chaves;

2. **No searching by value;**

Mas pode-se instruir à base de dados como dar parse aos valores, para fazer operações;

3. **Batch / sequential processing**

MapReduce;

2.7 Outras funcionalidades

1. Expiração de pares chave-valor;
2. Coleções de valores (We can store not only ordinary values, but also their collections such as ordered lists, unordered sets etc.);
3. Links entre pares chave-valor (podem ser usados quando se usa queries);

2.8 Exemplos

1. **RiakKV**

2. **Redis**

(Ver slides 12-40)

3 NoSQL Databases - Document Databases

As bases de dados de documentos são bastante eficientes em cenários **one-to-many**. Oferecem um esquema flexível (mesmo dentro das mesmas coleções (informação heterogénia)) e melhor performance (devido ao armazenamento da informação junto à entidade a que esta se refere) que são manipulados através de código simples.

Nota: 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 continua** codificada em JSON, XML, ou outro formato binário estruturado. É self-described (atributos são claros) e apresentam uma **estrutura em árvore**. São identificados por um **id único**.

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

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

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

Nota: 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 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.

3.1 Exemplos

1. **MongoDB**
2. **CouchDB**
3. **Couchbase**

(Ver slides 11-51)

4 Modelos de bases de dados

Os modelos de dados com os quais os programas vão trabalhar têm um papel fundamental na sua programação. Têm um efeito enorme na forma como os programas são escritos e na forma como nós pensamos sobre os problemas que estamos a resolver. Todos os modelos de dados têm formas diferentes de representar os dados e de os manipular.

4.1 Bases de Dados Relacionais

Este tipo de base de dados oferece vários benefícios (resolvendo a maior parte dos problemas com dados), entre os quais a **persistência** dos dados (ao guardar dados eles mantêm-se guardados), a **integração** com várias aplicações (com a mesma DB) e a **atomicidade, consistência, isolamento e durabilidade** oferecidas pelas transações (ACID).

4.1.1 Transactions - ACID Properties

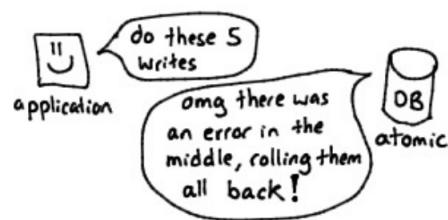
- **Atomicidade** - Numa transação, ou todas as operações são executadas, ou nenhuma é;
- **Consistência** - É garantido que as restrições de integridade antes da transação se mantêm após esta;
- **Isolamento** - As alterações feitas na BD só são visíveis quando a transação termina.
- **Durabilidade** - 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.

ACID is about safety guarantees for database transactions.

Atomicity

not about concurrent writes
(that's "isolation")



Consistency

super overloaded term.
This sense of "consistency" is actually an application property not a DB property.

not linearizability
not as in "eventual consistency"
About preserving application invariants like "every sale gets an invoice".

Isolation



Isolation is about preventing race conditions like this.

Some isolation levels:

- serializable
- snapshot isolation
- read committed

Durability



Perfect durability doesn't exist.

Can involve:

- write-ahead log (usually)
- replication

(Ver slides 7-8, história das bases de dados relacionais)

4.2 Current Trends and Issues

Algumas trends e issues motivaram à mudança nas tecnologias de armazenamento de dados relacionais (em use cases e na tecnologia).

Key Trends include: Aumentar o volume de dados e tráfego. Ligação entre dados mais complexo. **Key Issues include:** O problema → **impedance mismatch**.

4.3 Impedance Mismatch

Nos últimos tempos, tem-se assistindo 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**.

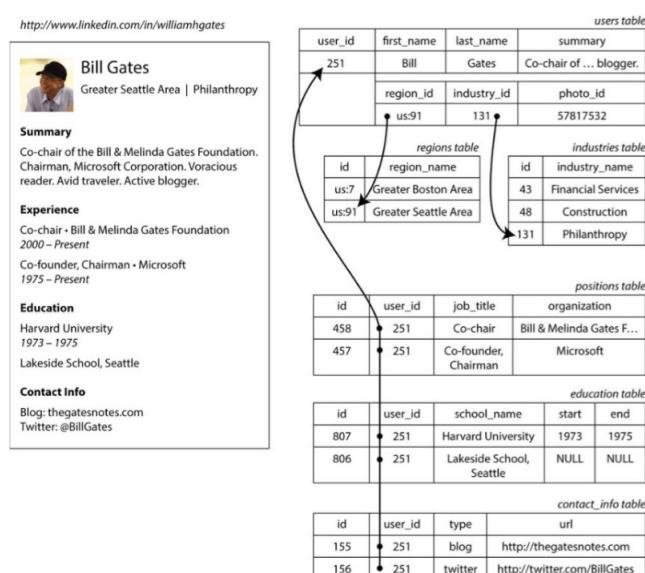
Tem-se também verificado um conflito entre os princípios de engenharia de software, onde o paradigma é orientado a objetos e os princípios relacionais baseados em modelos matemáticos. Este problema é designado por **Impedance** (oposição que um circuito elétrico faz à passagem de corrente elétrica quando é submetido a tensão) **Mismatch** (Disparidade, incompatibilidade).

Atualmente, este verifica-se nas estruturas isoladas, que violam os princípios da **normalização**. Para armazenar informação persistentemente em programas modernos, uma única estrutura lógica tem de ser separada (**normalização**).

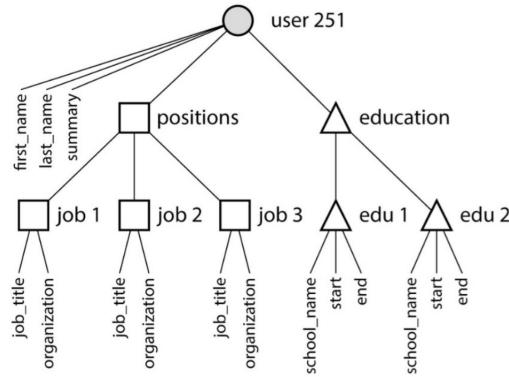
4.3.1 Exemplo

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.



4.3.2 One-to-Many relations

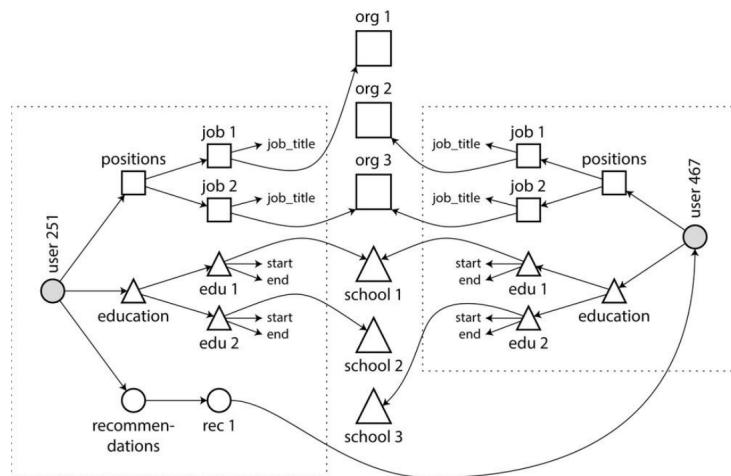


4.4 Normalização

Tem o objetivo de **reduzir a redundância dos dados**. Em DB's este reflete-se na utilização de IDs para identificar entidades, oferecendo uma **consistência de utilização**, ao mesmo tempo que **previne ambiguidades**, caso hajam entidades semelhantes (ex: com o mesmo nome), **facilita alterações** das entidades, uma vez que a sua informação está armazenada numa única tabela, motivo pelo qual também **facilita a tradução**.

Nota: Uma base de dados que verifica estas características diz-se **normalizada**. Uma base de dados na qual as entidades como região e industria estão referidas por ID diz-se **normalizada**. No entanto, uma base de dados que duplica nomes e propriedades de entidades em cada documento diz-se **desnormalizada**.

4.4.1 Many-to-Many relationships



4.5 Responder ao aumento do volume de dados

We are creating, storing, processing more data than ever before!

Existem duas abordagens possíveis:

- Construir bases de dados maiores;
 - Criar um grupo de máquinas mais pequenas que se complementam;
1. A primeira abordagem tem alguns problemas, uma vez que o **custo** de duplicar a capacidade de uma DB é geralmente mais do dobro do custo de uma DB "normal" e mesmo com recursos financeiros, há **limitações físicas** e de engenharia à sua capacidade;
 2. A segunda, apesar de mais exequível, tem também alguns defeitos, uma vez que por ser uma solução **barata**, pode refletir-se em **menos fiabilidade**. É ainda necessário a integração com um DBMS compatível com a tipologia.

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

4.6 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.

1. **Não é relacional** - Podem ser mas não são boas nisso;
2. **API simples** - Sem necessidade de realizar join;
3. **Teorema BASE & CAP** - Viola os princípios ACID;
4. **Schema-free** - Esquema implícito e gerido pela aplicação (sem verificações do lado da DB);
5. **Distribuídas** - Algumas mais do que outras;
6. **Open-source** - Mostly;

4.6.1 Transações BASE

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

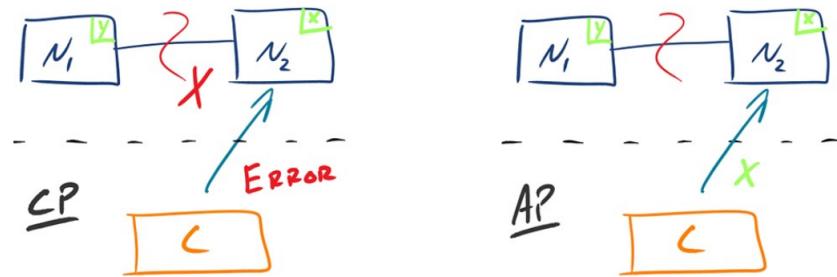
- **Basic Availability** - A DB funciona a maior parte do tempo;
- **Soft-state** - As manipulações dos dados não têm de ser write-consistent, nem diferentes réplicas têm de ser mutualmente consistentes o tempo todo. Escritas num nó da base de dados não têm de ser escritas garantidamente em simultâneo nos restantes nós;
- **Eventual consistency** - O armazenamento de dados eventualmente torna-se consistente, em algum ponto (e.g. lazily at read time);

As bases de dados NoSQL caracterizam-se então por ser **otimistas** e **simples**, o que torna a base de dados mais **rápida, disponibilidade em primeiro lugar, best effort e approximate answers OK**.

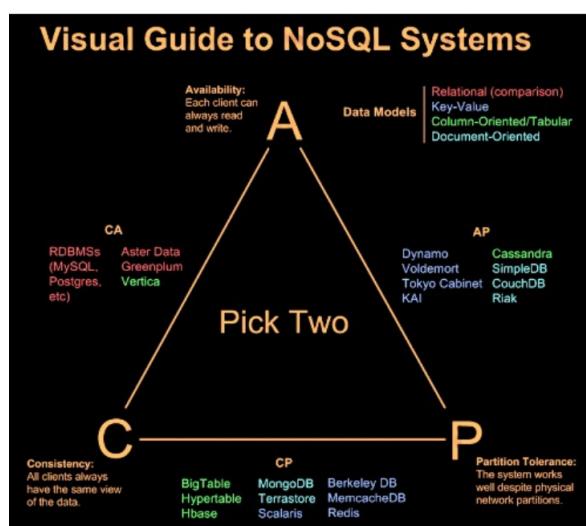
4.6.2 Teorema CAP (Brewer's)

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

- **Consistent** - Escritas atómicas em toda a DB em simultâneo;
- **Available** - A DB responde sempre a pedidos;
- **Partition Tolerant** - O sistema consegue funcionar mesmo que um nó deixe de responder;



1. No primeiro 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.
2. No segundo 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.



4.7 Tipos de Bases de Dados NoSQL

Existem vários tipos de bases de dados NoSQL. Core types:

- Key-value stores
- Document stores
- Column stores
- Graph databases

Non-core types:

- Object databases
- Native XML databases
- RDF stores
- ...

4.7.1 Key-value Databases

Esta base de dados está focada em **armazenamento chave-valor**. É a mais simples e funciona como uma simples hash table, tabela de dispersão (mapping).

Chave - Identificador (chave primária) único, normalmente uma string;

Valor - Pode assumir uma variedade de tipos, desde texto a 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 utilizado, ...

Tipicamente armazenam dados não persistentes. Não permitem relações entre entidades. Não usar quando existem relações entre entidades, ou quando as queries pretendem ter acesso aos valor.

4.7.2 Document Databases

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

Pertinentes queries sobre vários documentos, não só pela sua chave (id), mas também pelo seu valor. É possível construir índices, nos query patterns podemos criar, atualizar e remover documentos, bem como é possível fazer pesquisa usando queries complexas.

Quando comparadas com as bases de dados chave-valor, estas são uma evolução, em que o valor é examinável.

```

{
  "_id": "1",
  "name": "steve",
  "games_owned": [
    {"name": "Super Meat Boy"},
    {"name": "FTL"},
  ],
}

{
  "_id": "2",
  "name": "darren",
  "handle": "zerocool",
  "games_owned": [
    {"name": "FTL"},
    {"name": "Assassin's Creed 3", "dev": "ubisoft"},
  ],
}

```

Usar: Log de eventos, blogs, sistemas de gestão de conteúdos, web analytics, aplicações e-commerce, ... **Documentos estruturados com um schema semelhante.**

Não usar: Operações de **set** que envolva múltiplos documentos, onde o design da estrutura do documento esteja sempre a mudar e em relações **many-to-many**.

4.7.3 Column Databases

Data Model:

- **Família de Colunas (Table)**
 - Table é uma coleção de rows similares (mas não necessariamente identicas);
- **Row**
 - Coleção de colunas - deve compactar um grupo de dados a ser acedidos juntos;
 - Associado a uma chave única de row (row key);
- **Column**
 - Consiste no nome da coluna, valor da coluna (possivelmente outros metadados)
 - Valores escalares, sets flat, listas ou maps;

Query Patterns:

- Criar, atualizar ou remover row de uma dada família de colunas;
- Selecionar rows de acordo com a row key ou outras condições simples;

Use Cases:

- Event logging, content management systems, blogs, ...
- Basicamente tudo o que for structured flat data com um schema parecido
- Batch processing via MapReduce;

Quando NÃO usar:

- Precisamos de ACID transactions;
- Queries complexas (aggregation, joins, ...);
- Protótipos iniciais (quando o design da bd ainda está sujeita a mudanças)

4.7.4 Graph Databases

Data Model:

- Foco na modelação da estrutura e propriedade dos grafos;
- Grafos podem ser ou não direcionados;
- Grafos são coleções de:
 - Nós (vertices) para entidades do mundo real;
 - Relações (edges) entre estes nós;
- Ambos os nós e relações podem ter propriedades;

Query Patterns:

- Criar, atualizar ou remover nós/relações do grafo:
 - Algoritmos de Grafos
 - General Graph Traversal
 - Sub-Graph ou Super-Graph Queries
 - Similarity based Queries

Use Cases:

- Social networks, routing, dispatch, location based services, ...

Quando NÃO usar:

- Quando operações extensivas de batch são necessárias (múltiplos nós/relações são afetadas);
- Vamos dar store de grafos demasiado grandes (a distribuição de grafos ou até mesmo impossível);

4.7.5 Native XML Databases

Data Model:

- Documentos XML;
- Estrutura em árvore com nested elements, atributos e valores de texto;
- Documentos organizados em coleções;

Query Languages:

- Xpath (XML Path Language);
- XQuery (XML Query Language);
- XSLT (Extensible Stylesheet Language Transformations);

4.7.6 RDF Databases

- RDF Triples
 - Subject, Predicate, Object;
 - Cada Triple representa uma statement sobre uma entidade do mundo real;
- Triples podem ser vistas em grafos
 - Vertices representam os subjects e objects;
 - Edges correspondem diretamente às statements individuais;

Query Languages:

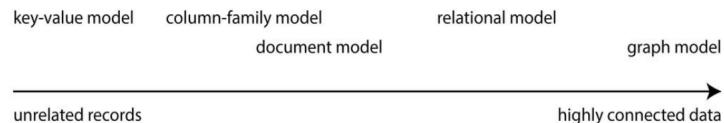
- SPARQL (SPARQL Protocol and RDF Query Language);

4.7.7 Databases and data connectivity

❖ Relational model

❖ NoSQL models

- Key-value stores
- Document stores
- Column stores
- Graph databases



4.7.8 Considerações Finais

As NoSQL DBs **NÃO** são o fim das bases de dados relacionais. Estas continuam a ser preferíveis para 90% dos projetos, para além do facto das pessoas estarem mais familiarizadas, serem mais estáveis, terem mais features e mais documentação e suporte disponível.

Mesmo assim devemos considerar diferentes modelos e sistemas de bases de dados.

Persistência Poliglota - Uso de diferentes data stores em diferentes circunstâncias.

5 Armazenamento e recolha de dados

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.

5.1 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). É um ficheiro de texto simples, em que cada linha é um par chave/valor separados por vírgula. Uma call ao cbd_set acrescenta uma linha ao ficheiro, sendo que versões anteriores não são overwritten.

```
$ cbd_set 42 '{"name":"San Francisco","attractions":["Exploratorium"]}'  
$ cbd_get 42  
{"name":"San Francisco","attractions":["Exploratorium"]}  
  
$ cat database  
123456,{"name":"London","attractions":["Big Ben","London Eye"]}  
42,{"name":"San Francisco","attractions":["Golden Gate Bridge"]}  
42,{"name":"San Francisco","attractions":["Exploratorium"]}
```

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 perfomance**, $O(1)$, 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 $O(n)$. Para um função extremamente simples tem uma boa performance. Muitas bases de dados usam uma append-only data file ou logs. O problema é quando a DB cresce.

Para encontrar eficientemente o valor de uma chave, precisamos de uma estrutura de dados diferente: um índice.

- Um índice é uma estrutura de dados adicional que deriva dos dados primários;
- Um índice bem escolhido aumentam a velocidade das queries, mas cada índice torna as operações de write mais lentas;

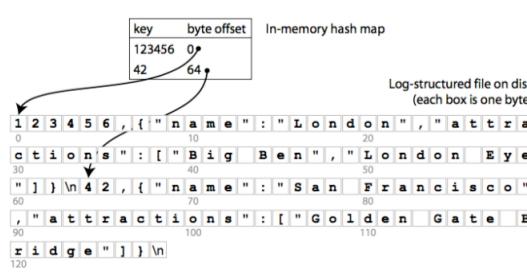
Nota: A não repetição de chaves iria aumentar a eficiência da consulta uma vez que assim que encontrasse uma correspondência não precisava de continuar à procura da chave mais recente. No entanto, neste cenário a inserção iria ser prejudicada, uma vez que a alteração de uma linha implica a leitura e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

5.2 Índices Hash

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.

Key-value stores são tipo um dicionário, normalmente implementados como um hash map. Uma estratégia simples de indexação: manter um in-memory hash map onde todas as keys estão mapeadas para um byte offset no ficheiro de dados.

Isto é o que algumas key-store databases fazem (e.g. Bitcask, Riak, ...). Oferecem boa performance se o hash map for mantido em memória.



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

5.3 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**.

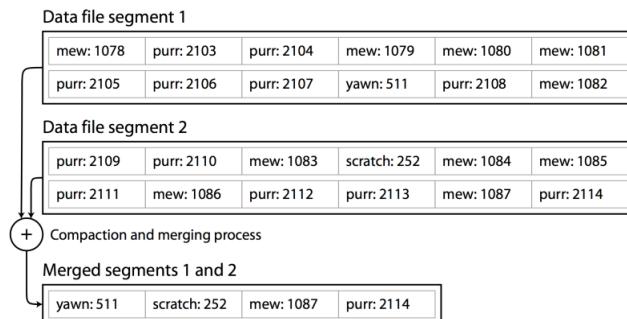
Segmentar: Em vez de utilizar um único ficheiro, utilizar vários (melhor performance). Cada segmento contém todos os valores escritos na DB num período de tempo.

Nota: 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: Remove chaves duplicadas do log.

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

Nota: 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.



5.4 Outros problemas

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). Quando os segmentos do log são combinados, o processo discarta qualquer valor anterior para a chave eliminada.

No que toca à **recuperação de falhas**, se o sistema utilizar estruturas de mapeamento (índices) em memória, o sistema irá demorar algum tempo a reconstruir os mesmos em caso de falha do sistema uma vez que serão perdidos. A solução passa pela criação de snapshots em memória de cada hash map dos seguimentos (índice).

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 que permitem a partes corrompidas do log serem identificadas e ignoradas.

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 leitura pode ser feita em simultâneo por várias threads.

5.5 Append-only log

O design append-only parece discutível. À primeira vista, porque não dar update aos dados do ficheiros dando overwrite aos valor antigo com o novo valor?

Acaba por ser bom por várias razões:

- Dar append e combinar segmentos são operações de escrita sequenciais, que são muito mais rápidas que operações de escrita aleatórias;
- Concorrência e recuperação de falhas são muito mais simples se os ficheiros segmentados forem append-only ou imutáveis;

Mesmo havendo soluções para todos os problemas apresentados, há alguns que não têm resolução, como a necessidade da **hash-table caber em memória** (é difícil fazer um hash map on-disk que tenha boa performance) e a dificuldade em fazer **queries para intervalos de valores**.

5.6 Sorted String Table (SSTable)

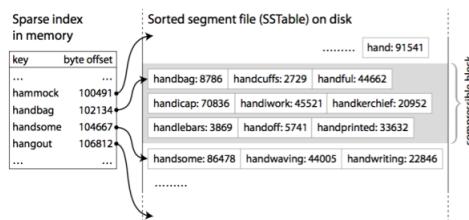
Neste tipo de base de dados os pares chave/valor são ordenados pela chave, que é única (não pode aparecer mais do que uma vez em cada ficheiro segmentado combinado, o processo de combinar já faz isso).

5.6.1 Vantagens

Aqui a **combinação** dos segmentos torna-se ainda mais **simples** e **eficiente**, uma vez que em cada segmento estarão os pares cujas chaves estão próximas em termos de ordenação. Mesmo se os ficheiros são maiores do que (tipo algoritmo mergesort).

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



5.6.2 Combinação de segmentos da SSTable



5.6.3 Ordenação

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

- Pode ser feito eficientemente porque os dados já estão ordenados nos pares key-value;
- A novo file SSTable torna-se o segmento mais recente da DB;
- Quando a noca SSTable está pronta, a memtable pode ser esvaziada;

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**.

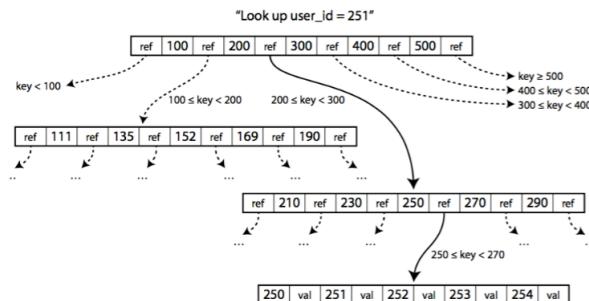
5.6.4 Recuperação a falhas

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 mantém um **append-only log** em memória, que atualizam em cada escrita, de forma a permitir esta recuperação.

5.7 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.

As B-trees partem a Db em blocos/páginas de tamanho fixo, tradicionalmente 4KB, e a leitura e escrita numa página de cada vez. Facilita o seu armazenamento uma vez que estão em sintonia com o hardware.



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

A estrutura começa pela página raiz.

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

Cada filho é responsável por um intervalo contínuo de chaves, e as chaves da página root indicam os intervalos de chaves dos filhos.

5.7.1 Operações

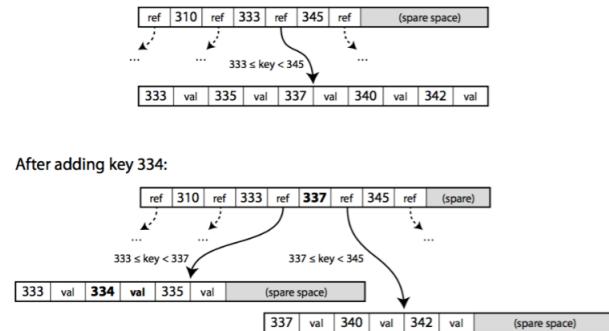
As operações de **pesquisa, inserção e remoção** são de complexidade $O(\log_k n)$.

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.

Atualizar um valor de uma key já existente: Pesquisar pela leaf page que contém a key e atualizar o valor. Escrever a página de volta para o disco (qualquer referência a esta página permanece válida).

A árvore permanece balanceada.

5.7.2 Split numa página



5.7.3 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 semáfros.

5.7.4 Otimização

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

Nota: 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 intacta durante o processo de cópia.

Podem ainda ser aplicadas técnicas de abreviação das chaves.

5.7.5 B-tree versus LSM-trees

LSM-trees são tipicamente mais rápidas para escritas, enquanto que as B-trees são mais rápidas para leituras.

Um downside de um log-structured storage é que o processo compactação pode interferir com a performance de ongoing leitura e escritas.

Nas B-trees cada key existe apenas uma vez no índice, enquanto que num log-structured storage pode ter várias cópias de diferentes segmentos.

- Isto faz com que as B-trees sejam melhores para DB's que querem oferecer semânticas de transações fortes;
- Em muitas DB's relacionais, é implementado usando locks no range das keys, num índice B-tree, estes locks podem ser diretamente aplicados na árvore;

5.8 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 (there might be many rows (documents, vertices) with the same key).

Pode ser feito de duas maneiras:

- Fazer aos valores do índice um matching row identifiers (como um posting list in a full-text index);
- Tornar cada key única adicionando um identificador de row;

Os índices B-trees e log-structured podem ser usados como índices secundários.

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).

5.9 Manter tudo na 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 homólogas (lados opostos), 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.

Nota: 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.

5.10 Processamento e análise transacional

5.10.1 Online Transaction Processing (OLTP)

O **processamento transacional** 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** (padrão de acesso).

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

5.10.2 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.

Nota: 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.

5.10.3 OLTP vs OLAP

O **OLTP** faz leituras de poucos registo 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

RDBMS e SQL trabalham bem com OLTP-type queries e com OLAP-type queries.

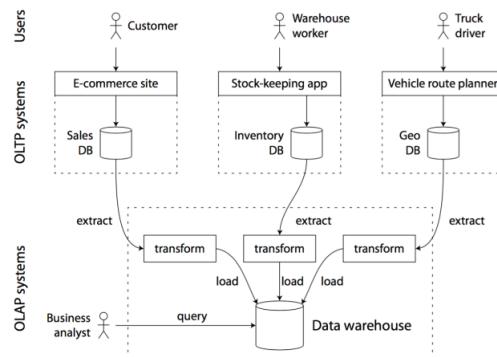
Os sistemas OLAP geralmente ocupam mais espaço que os OLTP porque não apresentam normalização dos dados, ou seja, não tiram partido das características relacionais.

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**.

5.11 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 regtos nulos ou não consistentes.

5.11.1 Porquê separa Data Warehouse

Porquê combinar OLTP e OLAP?

- Data warehouse são normalmente relacionais, porque SQL é bom para analytic queries.
- Muitas ferramentas OLTP (querying, visualization, ...)

Porquê separar?

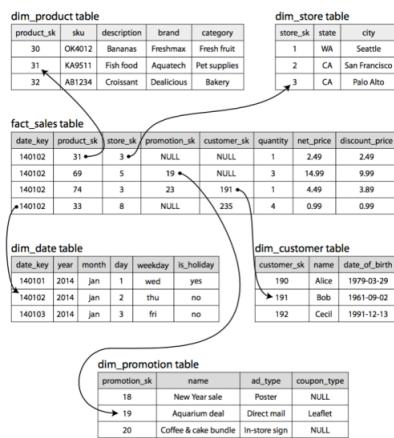
- Diferentes funcções e diferentes dados;

Muitos vendedores suportam ou transaction processing or analytics workloads.

5.12 Esquema da base de dados

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

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.



5.13 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 (fact tables podem ter trilhões de rows e pentabytes de dados, normalmente têm mais de 100 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, OLTP databases são row-oriented: todos os valores de uma row estão próximos. Neste caso, a solução passa pelo **armazenamento orientado às colunas**.

5.13.1 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**. Se cada coluna estiver num ficheiro separado, a query apenas precisa de ler e processar estas colunas.

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

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

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
```

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

Uma técnica que é particularmente efetiva em data warehouses é bitmap encoding.

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.

5.13.2 Escrita

O armazenamento de column-oriented, a compressão e a ordenação ajudam a fazer as queries OLAP mais rápidas.

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**.

- ❖ A good solution: LSM-trees (Log Structured Merge)
- ❖ All writes first go to an in-memory store, where they are added to a sorted structure, and prepared for writing to disk.
- ❖ When enough writes have accumulated, they are merged with the column files on disk, and written to new files in bulk.
- ❖ Queries need to examine both the column data on disk and in memory.

5.14 Materialized views

Data warehouse queries often involve an aggregate function (COUNT, SUM, AVG, MIN, ...). **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 ou OLAP cube**, um tipo de **materialized view**, que consistem numa tabela da base de dados em cache que armazena dados agregados para efeitos de eficiência.

- ❖ Example: Two-dimensional data cube, aggregating data by summing

		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	14967.09	5910.43	7328.85	6885.39	lots

6 Formatos de dados

7 NoSQL Databases - Column Databases

A ideia geral é que vamos **armazenar e processar dados por coluna** (column) em vez de por linha (row). 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 que os dados sejam esparsos e que hajam bastantes campos nulos. É descrita como: “sparse, distributed, persistent multidimensional sorted map”.

❖ Table example:

ID	name	address	zip code	phone	city	country	age
1	Benny Smith	23 Workhaven Lane	52683	14033335568	Lethbridge	Canada	43
2	Keith Page	1411 Lillydale Drive	18529	16172235589	Woodridge	Australia	26
3	John Doe	1936 Paper Blvd.	92512	14082384788	Santa Clara	USA	33

❖ Store by row:

```
1,Benny Smith,23 Workhaven Lane,52683,14033335568,Lethbridge,Canada,43;2,Keith Page,1411 Lillydale Drive,18529,16172235589,Woodridge,Australia,26;3,John Doe,1936 Paper Blvd.,92512,14082384788,Santa Clara,USA,33;
```

❖ Store by column:

```
1,2,3;Benny Smith,Keith Page,John Doe;23 Workhaven Lane,1411 Lillydale Drive,1936 Paper Blvd.;52683,18529,92512;14033335578,16172235589,14082384788;Lethbridge,Woodridge,Santa Clara;Canada,Australia,USA;43,26,33;
```

❖ Store by row:

ID	name	address	zip code	phone
1	Benny Smith	23 Workhaven Lane	52683	14033335568
2	Keith Page	1411 Lillydale Drive	18529	16172235589
3	John Doe	1936 Paper Blvd.	92512	14082384788

❖ Store by column:

ID	name	address	zip code	phone
1	Benny Smith	23 Workhaven Lane	52683	14033335568
2	Keith Page	1411 Lillydale Drive	18529	16172235589
3	John Doe	1936 Paper Blvd.	92512	14082384788

7.1 Vantagens

É 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 estes dados neste domínio tendem a estar mais relacionados.

7.1.1 Explicado

Torna algumas queries mesmo muito rápidas:

- Aggregation queries;
- Funções sobre fields (ex: average salary);

Melhor compressão de dados:

- Ao correr o algoritmo em cada coluna (dados similares);
- Mais notável quando começamos a ter datasets grandes;

7.2 Desvantagens

Possui algumas desvantagens, como o **carregamento incremental de dados**, o uso de **OLTP** (OnLine Transaction Processing), ou a realização de **queries a linhas** (dados individuais).

7.2.1 Explicado

Aggregation é bom, mas algumas aplicações precisam de ser capazes de mostrar dados para cada individual record. BDs colunares são geralmente não muito boas para esses tipos de queries. Escrever novos dados pode demorar tempo, inserir um novo record numa row oriented database é uma simples write operation. Fazer update de muitos values numa column db pode demorar muito tempo.

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's colunares. 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.

7.3 Modelo de Dados

Família de colunas (tabela) - Uma tabela é uma coleção de linhas semelhantes (não necessariamente idênticas).

Linha - Uma linha é um conjunto de colunas (deve abranger um grupo de dados que é acessado em conjunto). Associado com uma chave de linha (row key) única.

Coluna - Uma coluna consiste num nome de coluna (column name) e um valor de coluna (column value), possivelmente outros records de metadados. Valores escalares, mas também sets, listas e mapas.

7.4 Casos de Uso

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

...

Processamento de dados em batch via MapReduce.

Não é recomendado a utilização deste tipo de bases de dados quando **transições ACID** (atomicidade, consistência, isolamento, durabilidade) são necessárias, quando é necessário usar **queries complexas** como joins, nem **prototipagem** (o design da BD pode mudar), uma vez que a sua orientação ao serviço, a torna altamente dependente dos seus requisitos.

7.5 Tecnologias Representativas



Bigtable é a inspiração para as datastores orientadas a colunas. Bases de dados influenciadas pela Bigtable:

- HBase
- HyperTable

Cassandra é uma extensão da Bigtable com aspectos do Dynamo da Amazon.

7.6 Apache Cassandra

7.6.1 Introdução

É uma Cross Platform, **open-source** Column Database, cujas features incluem **alta disponibilidade, escalabilidade linear, fragmentação, replicação P2P configurável, consistência tunable e suporte para MapReduce**.

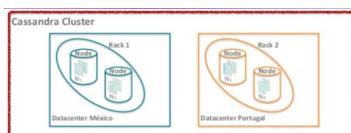
Atualmente é o sistema de bases de dados orientado a colunas mais utilizado. Foi inicialmente desenvolvido pelo Facebook (2008) mas atualmente está sob gestão da fundação para o software Apache.

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

7.6.2 Motivação

- High availability;
- High write throughput, sem sacrificar read efficiency;
- Fault tolerance;
- High and incremental scalability;
- Reliability at massive scale;

7.6.3 Topologia do Sistema



Node - É a unidade base destes sistemas, são máquinas onde a Cassandra está em execução.

Data Center - Os nós são agrupados em data centers. Conjunto de nós relacionados entre si, que fazem replicação dos dados de forma a garantir a **tolerância a falhas**.

Cluster - Conjunto de data centers sobre os quais são escritos os mesmos dados.

7.6.4 Arquitetura do Sistema

Cluster Membership - Como é que os nodes são adicionados ou apagados de um cluster.

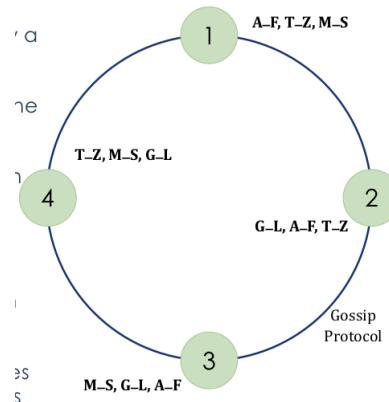
Partitioning

- Como é que os dados são particionados pelos nodes;
- Nodes são logicamente estruturados numa **Ring Topology**;
- Hashed Values e Keys associados a Data Partitions são usadas para lhes dar assign a um node no ring.

Replication

- Como é que os dados são duplicados pelos nós;
- Cada DataItem é replicado por N (replication factor) Nodes.

7.6.5 Topologia do Sistema



- Cluster Data é gerida por um Ring of Nodes;
- Cada Node tem parte da BD;
- Rows são distribuidas baseadas na primary key;
- Rows são distribuidas baseadas na primary key (Row Lookups são rápidos);
- Múltiplos nodes tem os mesmos dados, para garantir availability e durability;
- Não há nenhum master node - Todos os nodes podem realizar todas as operações;

7.6.6 Modelo de dados

Keyspaces > Tables > Rows > Columns

Keyspace:

- É um **namespace** que define data replication nos nodes;
- Um cluster contém **um keyspace por node**;

Table (Column Family):

- Coleção de rows (parecidas);
- Multi-Dimensional Map indexado por chave (row key);
- Tables Schema tem de ser especificado mas pode ser modificado;
- 2 tipos: Simple ou Super (nested Column Families);

Row:

- Coleção de colunas;
- Rows numa table não precisam de ter as mesmas colunas
- Cada Row É **uniquely identified** por uma **primary key**;

Column:

- Name-Value Pair + Dados Adicionais;

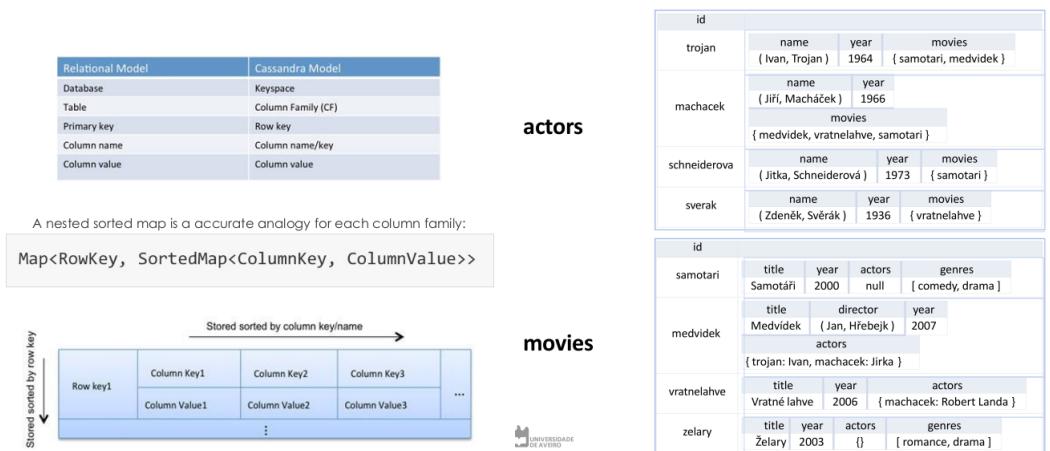
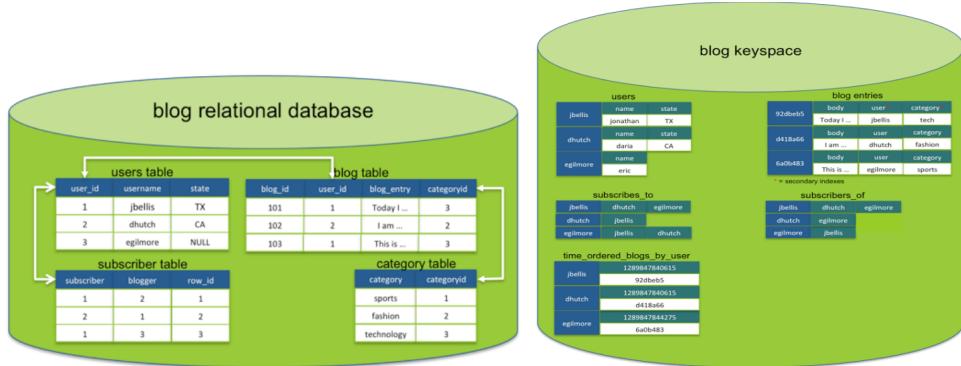


Figure 1: Exemplo de um Data Model

7.6.7 Cassandra vs Relational Example



7.6.8 Column Values

Os value types aceites pelas colunas são:

- **Empty Value**
 - Null
- **Atomic Value**
 - **Native Data Types** (Text, Integers, Dates, ...)
 - **Tuples** (Tuplo de fields anonimos, cada um com o seu Type (podendo cada membro do tuplo ter um type diferente))
 - **User Defined Types (UDT)** (Set de named fields para cada type)
- **Collections**
 - **Lists, Sets e Maps** (Nested Tuples, UDTs ou collections são permitidos mas apenas um Forzen Mode (elementos são serializados quando são guardados))

7.6.9 Data Model - Dados adicionais

Associados a cada coluna no caso dos atomic values ou qualquer element de uma collection.

Time to Live (TTL): Após um certo tempo (segundos) um dado value é automaticamente apagado;

Timestamp: Timestamp de quando a última modificação foi feita. Fornecido automaticamente ou manualmente;

Ambos estes elementos podem ser queried mas não no caso de collections e dos seus elementos.

7.6.10 Cassandra API

- ❖ CQLSH
 - interactive command line shell
 - bin/cqlsh
 - uses CQL (Cassandra Query Language)
- ❖ Client drivers
 - provided by the community
 - available for various languages
 - Java, Python, Ruby, PHP, C++, Scala, Erlang, ...

7.7 Cassandra Query Language (CQL)

Declarative query language inspired by SQL.

- ❖ **DDL statements (Data Definition Lang.)**
 - CREATE KEYSPACE – creates a new keyspace
 - CREATE TABLE – creates a new table
 - ...
- ❖ **DML statements (Data Manipulation Lang.)**
 - SELECT – selects and projects rows from a single table
 - INSERT – inserts rows into a table
 - UPDATE – updates columns of rows in a table
 - DELETE – removes rows from a table
 - ...

7.7.1 Keyspace

- | | |
|--|--|
| ❖ CREATE KEYSPACE
<pre>CREATE KEYSPACE [IF NOT EXISTS] keyspace_name WITH options</pre> | ❖ USE KEYSPACE
<pre>USE KEYSPACE</pre> |
| ❖ Replication option is mandatory <ul style="list-style-type: none">– SimpleStrategy<ul style="list-style-type: none">• one replication factor for the whole cluster– NetworkTopologyStrategy<ul style="list-style-type: none">• individual replication factor for each data center | ❖ DROP KEYSPACE
<pre>DROP KEYSPACE [IF EXISTS]</pre> |
| | ❖ ALTER KEYSPACE
<pre>ALTER KEYSPACE keyspace_name WITH options</pre> |
- ```
CREATE KEYSPACE Excelsior
 WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};

CREATE KEYSPACE Excalibur
 WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1' : 1, 'DC2' : 3}
 AND durable_writes = false;
```
- ```
ALTER KEYSPACE Excelsior
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 4};
```

7.7.2 Table - Create Statement

❖ CREATE TABLE
– creates a new table within the current keyspace
– each table must have one primary key specified

```
CREATE TABLE [ IF NOT EXISTS ] table_name
    '(' column_definition ',' column_definition )*
    [ ',' PRIMARY KEY 'primary_key' ]
    ')'
    [ WITH table_options ]
```

column_definition ::= column_name col_type [STATIC] [PRIMARY KEY]
primary_key ::= partition_key [',' clustering_columns]
partition_key ::= column_name!('column_name', 'column_name')*'
clustering_columns ::= column_name (',' column_name)*'
table_options ::= COMPACT STORAGE [AND table_options] | CLUSTERING
ORDER BY (' clustering_order ') [AND table_options] | options
clustering_order ::= column_name (ASC | DESC) (',' column_name (ASC |
DESC))*

7.7.3 Table - Primary Key

Primary key has 2 parts:

Compulsory Partition Key:

- Uma ou mais colunas;
- Descreve como é que as table rows são distribuidas pelas partitions

Optional Clustering Key (ou Columns):

- Define a Clustering Order, isto é, como é que as tables estão localmente guardadas dentro de uma partição;

```
CREATE TABLE groups (
    groupname text,
    username text,
    email text,
    age int,
    PRIMARY KEY (groupname, username)
```

PRIMARY KEY has two components: **groupname**, which is the **partitioning key**, and **username**, which is called the **clustering key**. This will give us one partition per groupname. Within a particular partition (group), rows will be ordered by username.

7.7.4 Key Roles

Partition Key: Responsável por da Distribution (partitioning) pelos nodes da BD. Pode ter multiplas colunas.

Clustering Key: Responsável oir Data Sorting dentro de uma partition. Pode ter multiplas colunas.

Primary Key: Equivalente a uma Partition Key num Single-Field-Key Table;

Composite Key: Multiple-Column Key

```
create table mytable (
    k_part_one text,
    k_part_two int,
    k_clust_one text,
    k_clust_two int,
    k_clust_three uuid,
    data text,
    PRIMARY KEY((k_part_one,k_part_two), k_clust_one, k_clust_two, k_clust_three)
);
```

Figure 2: Exemplo de uma Composite Key

7.7.5 Table - Other Statements

```
CREATE TABLE postsbyuser (
    userid bigint,
    posttime timestamp,
    postid uuid,
    postcontent text,
    PRIMARY KEY ((userid), posttime)
) WITH CLUSTERING ORDER BY (posttime DESC);

CREATE TABLE timeline (
    userid uuid,
    posted_month int,
    posted_time uuid,
    body text,
    posted_by text,
    PRIMARY KEY (userid, posted_month, posted_time)
) WITH compaction = { 'class' : 'LeveledCompactionStrategy' };

CREATE TABLE movies (
    id TEXT,
    title TEXT,
    director TUPLE<TEXT, TEXT>,
    year SMALLINT,
    actors MAP<TEXT, TEXT>,
    genres LIST<TEXT>,
    countries SET<TEXT>,
    PRIMARY KEY (id)
)
```

7.7.6 Select Statement

❖ DROP TABLE

```
DROP TABLE [ IF EXISTS ] table_name
```

❖ TRUNCATE TABLE

– preserves a table but removes all data it contains

```
TRUNCATE [ TABLE ] table_name
```

❖ ALTER TABLE

```
ALTER [ TABLE ] table_name alter_table_instruction
```

```
alter_table_instruction ::= ADD column_name cql_type ( ',' column_name cql_type )* | DROP column_name ( column_name )* | WITH options
```

```
ALTER TABLE addamsFamily ADD gravesite varchar;
```

7.7.7 Select - FROM Clause

Define uma **única tabela** para a query:

- do keyspace atual/especificado;
- dar join a outras tabelas não é possível;

Suporta:

- **distinct** para eliminar rows duplicadas;
- (user-defined) **aggregate functions**
- ***** para selecionar todas as colunas;
- **AS** para atribuir um alias a uma coluna;
- WRITETIME (timestamp) e TTL (time to live) para selecionar os valores de timestamp e ttl de uma coluna (não pode ser usado na clausula WHERE);



```
SELECT name, occupation FROM users WHERE userid IN (199, 200, 207);
SELECT JSON name, occupation FROM users WHERE userid = 199;
SELECT name AS user_name, occupation AS user_occupation FROM users;

SELECT time, value
FROM events
WHERE event_type = 'myEvent'
AND time > '2011-02-03'
AND time <= '2012-01-01'

select videoname, ttl(videoname), writetime(videoname) from videos;
videoname | ttl(videoname) | writetime(videoname)
Ondas gigantes na barra! | null | 1509294890781000
Aviões de papel! | null | 1509294888607000

SELECT COUNT (*) AS user_count FROM users;
```

| 55

7.7.8 Select - WHERE Clause

Sintaxe parecidas entre CQL e SQL.

As diferenças aparecem devido ao facto da Cassandra estar a lidar com dados distribuidos, logo temos de procurar prevenir queries ineficientes

- Rows estão espalhadas pelo cluster baseado na Hash das suas partition keys;
- Clustering keys são usadas para fazer cluster dos dados de uma partição permitindo um retrieval muito eficiente das rows;

Nota: Partition Key, Clustering e normal columns supostam diferentes sets de restrições dentro da clausula WHERE.

Partition Key Columns:

- Suportam = e IN;
- Todas as primary keys columns tem de ser usadas (restricted), a não ser que tenhasmos secondary indexes;
- Todas as colunas são precisas para computar a hash que vai permitir localizar os nós que contenham os dados partição;

Clustering Key:

- Suportam =, <, >, <=, >;
- IN retorna verdadeiro se o valor for um dos enumerados;
- **CONTAINS*** e **CONTAINS KEY**:
 - Retornam True se a coleção contiver o dado elemento;
 - Quando o query estiver a usar um secondary index;
 - Usado em collections* (lists, sets, maps) / maps**;

Examples:

```

CREATE TABLE numberOfRequests (
    cluster text,
    date text,
    datacenter text,
    hour int,
    minute int,
    numberofRequests int,
    PRIMARY KEY ((cluster, date), datacenter, hour, minute))

/* Data will be stored like this:
(datacenter: US_WEST_COAST {hour: 0 {minute: 0 {numberofRequests: 130}} {minute: 1 {numberofRequests: 125}} ...
{minute: 59 {numberofRequests: 97}}}) ->
{hour: 1 {minute: 0 ...
*/
```

SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND datacenter = 'US_WEST_COAST' AND hour = 14 AND minute = 00; ✗

SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND date = '2015-06-05' AND datacenter = 'US_WEST_COAST' AND hour = 14 AND minute = 00; ✓

SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND date = '2015-06-05' AND datacenter = 'US_WEST_COAST' AND hour = 14 AND minute = 00; ✗ ?

SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND date = '2015-06-05' AND datacenter = 'US_WEST_COAST' AND (hour, minute) IN ((14, 0), (15, 0));

SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND date = '2015-06-05' AND datacenter = 'US_WEST_COAST' AND (hour, minute) IN ((('US_WEST_COAST', 14), ('US_EAST_COAST', 17))) AND minute = 0;

>, >=, <= and < restrictions
Single column slice restrictions are allowed only on the last clustering column being restricted.
Multi-column slice restrictions are allowed on the last set of clustering columns being restricted.

-- OK
SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND date = '2015-06-05' AND datacenter = 'US_WEST_COAST' AND hour = 12 AND minute >= 0 AND minute <= 30;

-- OK
SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND date = '2015-06-05' AND datacenter = 'US_WEST_COAST' AND hour >= 12;

-- OK
SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND date = '2015-06-05' AND datacenter > 'US';

-- NOK
SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND date = '2015-06-05' AND datacenter = 'US_WEST_COAST' AND hour >= 12 AND minute = 0;

**CREATE TABLE numberOfRequests (
 cluster text,
 date text,
 datacenter text,
 hour int,
 minute int,
 numberofRequests int,
 PRIMARY KEY ((cluster, date), datacenter, hour, minute))**

-- OK
SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND date = '2015-06-05' AND datacenter = 'US_WEST_COAST' AND (hour, minute) >= (12, 0) AND (hour, minute) <= (14, 0);

-- OK
SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND date = '2015-06-05' AND datacenter = 'US_WEST_COAST' AND (hour, minute) >= (12, 30) AND (hour) <= (14);

-- NOK: the restrictions must start with the same column
SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' AND date = '2015-06-05' AND datacenter = 'US_WEST_COAST' AND (hour, minute) >= (12, 30) AND (minute) <= (30);

- ❖ Direct queries on secondary indices support only **=**, **CONTAINS** or **CONTAINS KEY** restrictions

```

CREATE TABLE contacts (
    id int PRIMARY KEY,
    firstName text,
    lastName text,
    phones map<text, text>,
    emails set<text>
);

select * from contacts
where firstName = 'Maria'; ✗
```

↓

```

/*
  Solution: Secondary Index
*/
CREATE INDEX ON contacts (firstName);
-- Using the keys function to index the map keys
CREATE INDEX ON contacts (keys(phones));
CREATE INDEX ON contacts (emails);
```

↓

```

SELECT * FROM contacts WHERE firstName = 'Benjamin';
SELECT * FROM contacts WHERE phones CONTAINS KEY 'office';
SELECT * FROM contacts WHERE emails CONTAINS 'Benjamin@oops.com'; ✓
```

7.7.9 Select - GROUP BY, ORDER BY & LIMIT

GROUP BY clause:

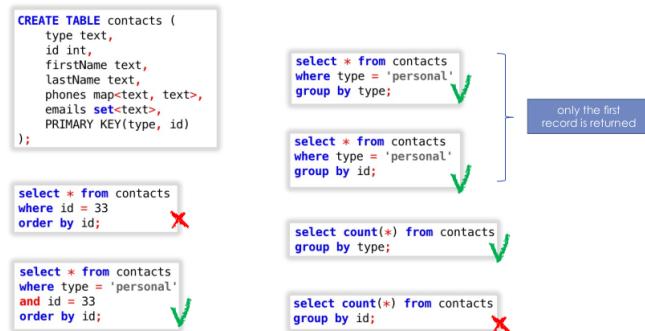
- Agrupar Rows de uma Table de acordo com certas colunas;
- **Apenas grouping induzidos por primary key columns são permitidos;**
- Quando non-grouping columns são selecionadas sem uma aggregate function, o primeiro value encontrado é sempre retornado;

ORDER BY clause:

- Define a ordem (ASC ou DESC) das rows retornadas;
- **Partition Key tem de ser restricted com = ou IN;**
- **APENAS** orderings induzidos por **CLUSTERING COLUMNS** são permitidos;

LIMIT clause:

- Limita o número de rows retornadas numa query result;



7.7.10 Select - User Defined Functions

User-Defined Functions (UDF)

- allow the execution of user-provided code (Java or JavaScript)
- Statements: CREATE (or REPLACE) /DROP FUNCTION

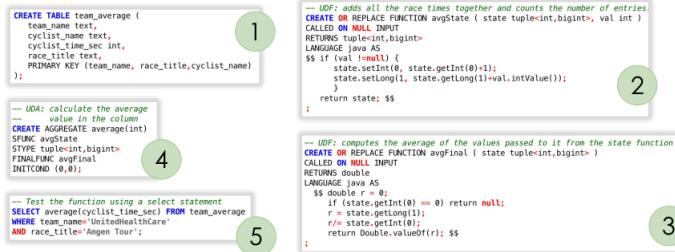
```
CREATE FUNCTION IF NOT EXISTS akeyspace.fname(someArg int)
CALLED ON NULL INPUT
RETURNS text
LANGUAGE java
AS $$  
// some Java code
$$;

CREATE FUNCTION IF NOT EXISTS div (n counter, d counter)
CALLED ON NULL INPUT
RETURNS double
LANGUAGE java AS '
return Double.valueOf(n/d);
';

select rating_counter, div(rating_total, rating_counter)
from ...
```

7.7.11 Select - Aggregates

- ❖ Native
 - COUNT(column), MIN(column), MAX(column),
SUM(column) and AVG(column)
- ❖ User-Defined Aggregate Function (UDA)
 - creation of custom aggregate functions



7.7.12 Select - ALLOW FILTERING

- ❖ Option used to explicitly allow (some) queries that require filtering
- ❖ By default, only non-filtering queries are allowed
 - i.e. queries where the number of rows read ~ the number of rows returned
 - such queries have predictable performance
 - execution time that is proportional to the amount of data returned



7.7.13 Insert Statement

Insere uma nova row numa dada table:

- Se a Primary Key existir, a row é atualizada;
- Existe a condição **IF NOT EXISTS** para apenas inserir caso uma row não exista;

Escreve uma ou mais colunas para uma dada row. Pelo menos as Primeary Key Columns têm de ser especificadas.

```
INSERT INTO NerdMovies (movie, director, main_actor, year)
VALUES ('Serenity', 'Joss Whedon', 'Nathan Fillion', 2005)
USING TTL 86400;

INSERT INTO NerdMovies JSON '{"movie": "Serenity",
"director": "Joss Whedon",
"year": 2005}';

CREATE TABLE movies (
    id TEXT,
    title TEXT,
    director TUPLE<TEXT, TEXT>,
    year SMALLINT,
    actors MAP<TEXT, TEXT>,
    genres LIST<TEXT>,
    countries SET<TEXT>,
    PRIMARY KEY (id)
);

INSERT INTO movies (id, title, director, year, actors, genres)
VALUES (
    'stestil',
    'Štěstí',
    ('Bohdan', 'Sláma'),
    2005,
    { 'vilhelmova': 'Monika', 'liska': 'Tonik' },
    [ 'comedy', 'drama' ]
)
USING TTL 86400
```

7.7.14 Update Statement

Atualiza rows existentes dentro de uma dada table:

- Se a row com a dada primary key não existir, é inserida;

Todas as Primary Key Columns têm de ser especificadas na clausula WHERE.

```
UPDATE movies
SET
    year = 2006,
    director = ('Jan', 'Svérák'),
    actors = { 'machacek': 'Robert Landa', 'sverak': 'Josef Tkaloun' },
    genres = [ 'comedy' ],
    countries = { 'CZ' }
WHERE id = 'vratnelahve';

UPDATE movies
SET
    actors = actors + { 'vilhelmova': 'Helenka' },
    genres = [ 'drama' ] + genres,
    countries = countries + { 'SK' }
WHERE id = 'vratnelahve';

CREATE TABLE movies (
    id TEXT,
    title TEXT,
    director TUPLE<TEXT, TEXT>,
    year SMALLINT,
    actors MAP<TEXT, TEXT>,
    genres LIST<TEXT>,
    countries SET<TEXT>,
    PRIMARY KEY (id)
);

UPDATE NerdMovies USING TTL 400
SET director = 'Joss Whedon',
main_actor = 'Nathan Fillion',
year = 2005
WHERE movie = 'Serenity';

UPDATE UserActions
SET total = total + 2
WHERE user = B70DE1D0-9908-4AE3-BE34-5573E5B09F14
AND action = 'click';
```

7.7.15 Parâmetros Insert & Update

❖ TTL: time-to-live

- 0 or null or simply missing for persistent values
- if set, the inserted values are automatically removed from the database after the specified time.

❖ TIMESTAMP: writetime

- only newly inserted / updated values are really affected
- if not specified, it will be used the current time (in microseconds)

```
UPDATE user USING TTL 3600 SET last_name = 'McDonald'
WHERE first_name = 'Mary';

SELECT first_name, last_name, TTL(last_name)
FROM user WHERE first_name = 'Mary';
first_name | last_name | ttl(last_name)
Mary       | McDonald   |      3588
```



```
INSERT INTO cycling.comments (
    id,
    created_at)
values (
    123456,
    toTimeStamp(now()));
```

7.7.16 Delete Statement

Remove existing rows/columns/collection elements de uma dada tabela.

A cláusula **WHERE** é usada para especificar qual a row que queremos deleted.

Múltiplas Rows podem ser apagadas com uma única query usando o operador **IN**.

Um range de rows pode ser apagado usando um inequality operator (<, >, <=, >=).

```
DELETE FROM NerdMovies USING TIMESTAMP 1240003134
WHERE movie = 'Serenity';

DELETE phone FROM Users
WHERE userId IN (C73DE1D3-AF08-40F3-B124-3FF3E5109F22, B70DE1D8-9908-4AE3-BE34-5573E5B09F14);
```