

Resumo de

CBD - 1

Escrito por Diogo Silva

**adaptando os slides e imagens fornecidos para a
disciplina**

Index

- I. Evolução dos Sistemas de Base de Dados
- II. Modelos de Dados
- III. Storage and Retrieval de Dados I – Data Structures & Indexes
- IV. Storage and Retrieval de Dados II – Transaction Processing & Transaction Analytics
- V. Formatos de Dados
- VI. Key-Value based DB
- VII. Document based DB
- VIII. Column based DB

Evolução dos Sistemas de Base de Dados

I. Sistemas de Dados?

Muitas aplicações hoje em dia são Data-Intensive, em vez de Compute-Intensive

Para Data-Intensive apps o poder bruto dos CPUs é muito raramente um problema. O problema maior é mesmo a quantidade e complexidade dos dados e a velocidade à qual estes mudam

Como tal, sistemas de dados tipicamente precisam de:

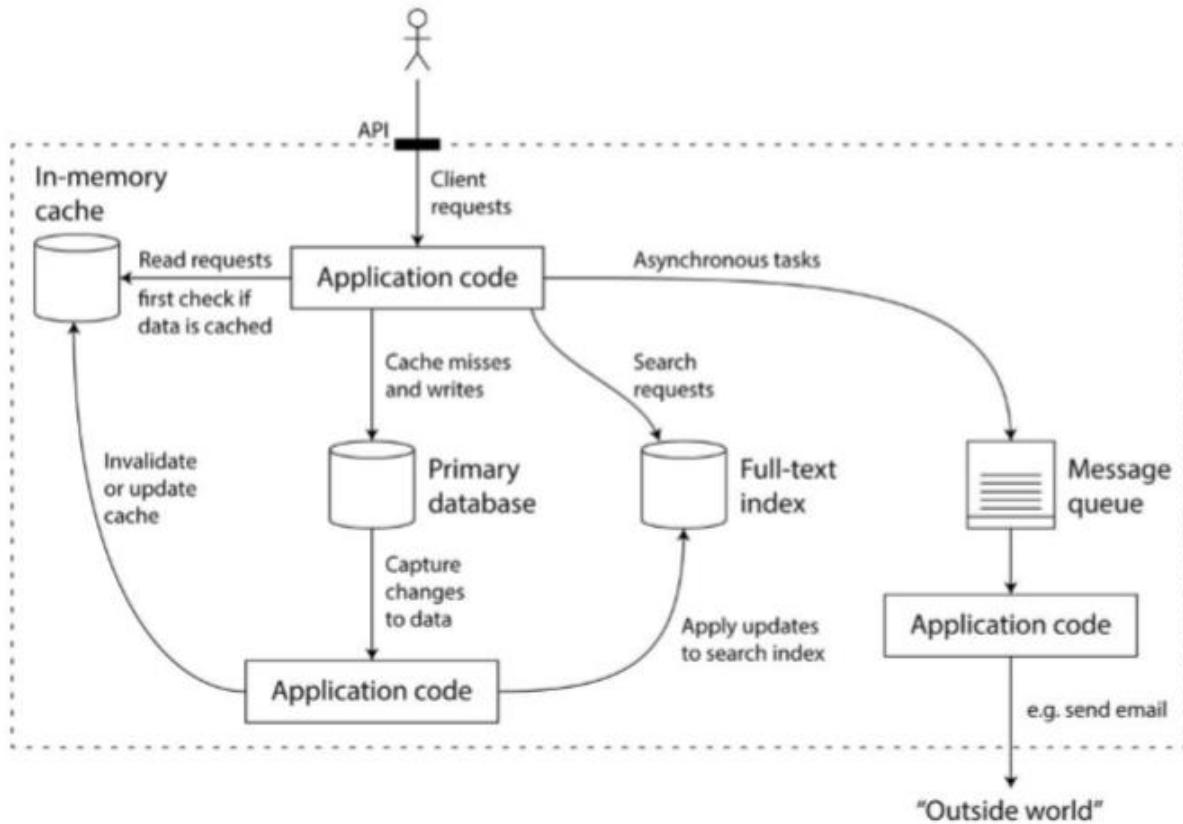
- Guardar dados tal que outra aplicação os possa voltar a encontrar – **Database**
- Lembrar resultados de operações caras, de forma a que as leituras sejam mais rápidas – **Caches**
- Permitir que os users pesquisem por dados usando uma keyword ou filtro de várias formas – **Search Indexes**

- Mandar mensagens para outros processos de forma a serem tratados de forma assíncrona – **Message Queues**
- Observar o que está a acontecer e agir baseado nesses eventos – **Stream Processing**
- Periodicamente compactar uma larga quantia de dados acumulados – **Batch Processing**

Cada vez mais as **aplicações requerem um maior wide-range de requesitos**. Muitas vezes, **uma única ferramente já não consegue ir ao encontro de todas as necessidades de data processing e storage**

Em vez disto, 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

P.ex, podemos ter uma aplicação com uma Caching Layer (e.g memcached), um Full-Text Search server (e.g Elasticsearch) e uma base de dados principal separada (e.g MySQL)



Img 1.1 – Stitching together several tools for our Data System

Alguns dos **desafios** que os Data Systems enfrentam são:

- Como garantir que todos os dados se mantêm corretos e completos, mesmo quando, internamente, ocorrer algum erro? (i.e persistencia de dados)
- Como fornecer bom performance para os clientes, mesmo quando partes do nosso sistema estiverem degradadas?
- Como escalar o sistema para ser capaz de aguentar uma load mais intensiva de trabalho
- Qual a apariencia de uma boa API para o serviço?

Por outro lado, estes são alguns dos **requisitos** dos Data Systems:

- **Reliability**

- O sistema deve continuar a funcionar de forma correta com o performance espectável mesmo na face de adversidades (como hardware ou software faults ou human error)

- **Scalability**

- A medida que um sistema cresce, em volume de dados, transito, ou complexidade, deve haver alguma maneira razoável de lidar com tal crescimento

- **Maintainability**

- Com o passar do tempo, muitas pessoas devem ser capazes de trabalhar no sistema de forma produtiva. Engineering e operations devem manter o comportamento atual e adaptar o sistema para novos use cases sem problema

Algumas **considerações importantes** a ter sobre data-intensive applications são:

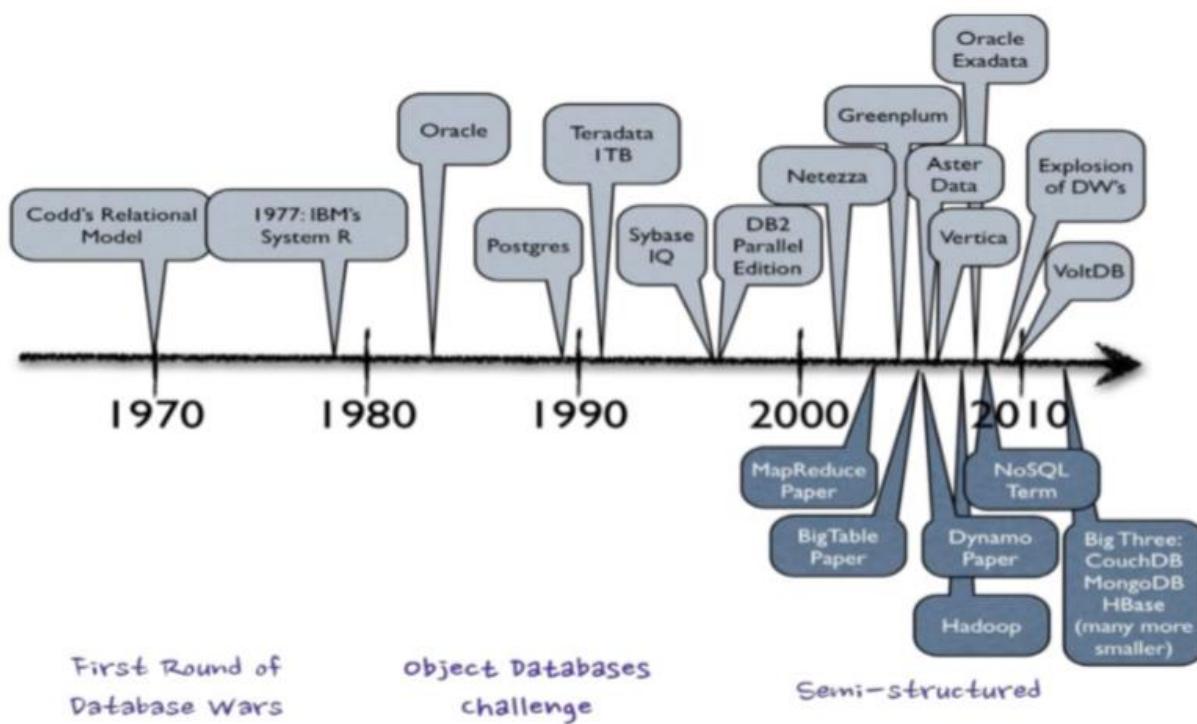
- **Reliability** significa fazer com que o sistema funcione mesmo quando faults ocorrem -> Implica aplicarmos técnicas de **Fault Tolerance** (técnicas que conseguem esconder certos tipos de faults do end user)

- **Scalability** significa termos estratégias para manter o performance do sistema bom, mesmo quando a workload aumenta, pelo que precisamos de **descrever** de forma **quantitativa** a **load** e **performance**

II. Sistemas de Base de Dados

Quando dizemos base de dados, por norma referimo-nos a um **set de dados relacionados entre si**, e a **organização destes mesmos**

Um **Database Management System (DBMS)** controla o acesso a esses dados fornecendo funções que permitem a **escrita, procura, update, retrieval e remoção** de grandes quantidades de informação



Img 1.2 – Timeline da evolução dos sistemas de Base de Dados



Img 1.3 – Exemplos de Sistemas de Bases de Dados

331 systems in ranking, August 2017									
Rank	Aug 2017	Jul 2017	Aug 2016	DBMS	Database Model	Score			Aug 2016
						Aug 2017	Jul 2017	Aug 2016	
1.	1.	1.	1.	Oracle	Relational DBMS	1367.88	-7.00	-59.65	
2.	2.	2.	2.	MySQL	Relational DBMS	1340.30	-8.81	-16.73	
3.	3.	3.	3.	Microsoft SQL Server	Relational DBMS	1225.47	-0.52	+20.43	
4.	4.	5.	5.	PostgreSQL	Relational DBMS	369.76	+0.32	+54.51	
5.	5.	4.	4.	MongoDB	Document store	330.50	-2.27	+12.01	
6.	6.	6.	6.	DB2	Relational DBMS	197.47	+6.22	+11.58	
7.	7.	8.	8.	Microsoft Access	Relational DBMS	127.03	+0.90	+2.98	
8.	8.	7.	7.	Cassandra	Wide column store	126.72	+2.60	-3.52	
9.	9.	10.	10.	Redis	Key-value store	121.90	+0.38	+14.57	
10.	10.	11.	11.	Elasticsearch	Search engine	117.65	+1.67	+25.16	
11.	11.	9.	SQLite	Relational DBMS	110.85	-3.02	+0.99		
12.	12.	12.	Teradata	Relational DBMS	79.23	+0.86	+5.59		
13.	14.	14.	Solr	Search engine	66.96	+0.93	+1.18		
14.	13.	13.	SAP Adaptive Server	Relational DBMS	66.92	+0.00	-4.13		
15.	15.	15.	HBase	Wide column store	63.52	-0.10	+8.01		
16.	16.	17.	Splunk	Search engine	61.46	+1.17	+12.56		
17.	17.	16.	FileMaker	Relational DBMS	59.65	+1.00	+4.64		
18.	18.	20.	MariaDB	Relational DBMS	54.70	+0.33	+17.82		
19.	19.	19.	SAP HANA	Relational DBMS	47.97	+0.03	+5.24		
20.	20.	18.	Hive	Relational DBMS	47.30	+1.10	-0.51		
21.	21.	21.	Neo4j	Graph DBMS	38.00	-0.52	+2.43		
22.	22.	25.	Amazon DynamoDB	Document store	37.62	+1.16	+11.02		
23.	23.	24.	Couchbase	Document store	32.97	-0.05	+5.57		

<https://db-engines.com/en/ranking>

1.4 – Estatísticas de uso e ranking de Sistemas de Bases de Dados atuais

Modelos de Base de Dados

I. THE BATTLE OF THE DATA MODELS (this sounds epic af)

Data Models são uma das partes mais importantes aquando do desenvolvimento de software

Vão ter um efeito profundo em como é que o resto do software vai ser escrito e como é que vamos pensar nos problemas quando os estivermos a resolver

Ha varios tipos diferentes de data models, cada um representativo de assumções que fazemos sobre como será utilizado

Vamos agora então ver um conjunto de general-purpose data models para data storage e querying

II. Relational Databases

Um dos mais bem conhecidos modelos de dados atuais, proposte me 1970.

Os dados estão organizados em relações (SQL Tables) onde cada relação é uma coleção de tuplos (rows) desorganizados

Has Stood the test of time (já estão em uso à mais de 30 anos)

O objetivo deste modelo era o de esconder os detalhes de implementação e representação dos dados internamente atrás de uma interface limpa.

A maioria dos problemas relacionados com dados pode ser resolvido através do uso de **Bases de Dados Relacionais**

Why? Well I'll tell you why! Bases de dados relacionais possuem:

- **Persistencia**

- Podemos guardar dados e termos a segurança que estes ficarão guardados

- **Integration**

- Podemos integrar várias apps diferentes através de uma DB Relacional Central

- **SQL**

- Structured Query Language. Standard, widespread e expressive forma de fazer query à BD

- **Transactions**

- **ACID** Transactions e forte consistencia
- Propriedades **ACID**:

- **Atomic**

- Todo o trabalho numa transação é realizado e completado ou nenhum é
- **Consistent**
 - Uma transação transforma uma base de dados de um estado consistente para outro.
Consistência é definida em termos de constraints
 - **Isolated**
 - Os resultados de qualquer mudança feita durante uma transação não são visíveis até que a transação tenha sido committed. Interações concurrentes comportam-se como se tivessem ocorrido de forma serializada
 - **Durable**
 - Resultados de uma committed transaction sobrevive failures

Ao longo dos anos têm surgido rivais a este modelo porem – Object Databases, XML Databases, ...

Mesmo assim, muito do que vemos usado na web hoje em dia é powered por uma relational database

Mais a frente vamos ver o novo concorrente que procura destronar a dominancia deste modelo – o NoSQL Movement

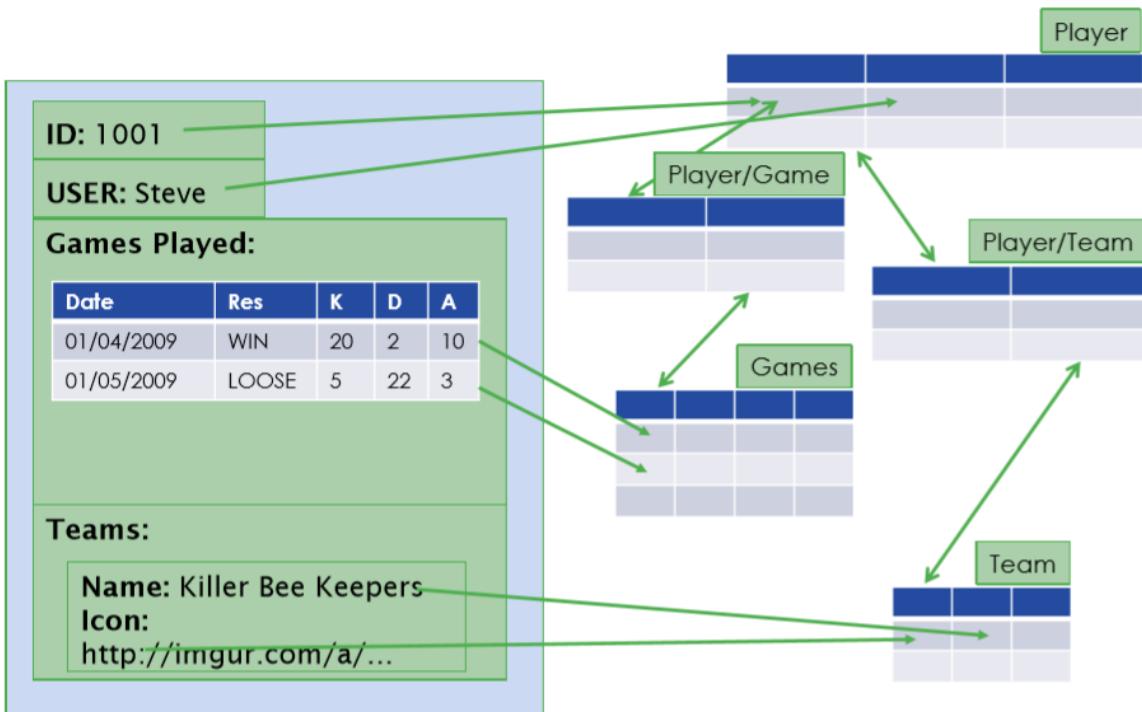
Mas porquê tentar destronar as Relational Databases se são tão widespread? Porque, por muito usadas que sejam, possuem tendencias e problemas chave que as tornam pouco atraentes.

Algumas **Key Trends** que nos levam à procura de modelos alternativos são:

- O aumento no volume de dados e transito
- Complexidade da conexão dos dados

Por outro lado o maior **Key Issue** deste modelo é:

- O problema da **Impedance Mismatch**
 - **Object Orientation** é baseado em princípios de engenharia de software
 - **Relational Paradigms** são baseados em teoria de conjuntos e matemáticas
 - **O mapeamento de um mundo para o outro tem problemas**
 - Para guardar dados de forma persistente num programa moderno com a única estrutura lógica tem de ser partida – i.e **Normalised**



2.1 – Exemplo da complexidade entre fazer a tradução do nosso objeto para o modelo relacional (Impedance Mismatch)

Para remendar o problema da Impedance Mismatch temos de recorrer a **Normalização**. Diz-se que uma base de dados está **normalizada** se todas as entidades (valores de cada row da tabela) são referidas por IDs únicos.

Caso a BD use nomes e propriedades duplicadas para as propriedades das entidades em cada documento então está **desnormalizada**

Mas para quê usar Ids?

- Estilo consistente
- Evita ambiguidades (p.ex se tivermos vários valores, como nomes, iguais mas a identificar pessoas diferentes)

- Nome é guardado apenas num lugar, sendo facil de atualizar
- Simplifica a tradução para outras linguagens

III. Increase in Data Volume

Estamos a cada dia que passa a criar, guardar e processar mais dados do que nunca antes visto

Os dados que produzimos têm vindo a mudar, desde documentos de texto isolados para massive linked open data sets.

Como é que podemos lidar com isto?

Well, temos duas maneiras:

- Construir maquinas de Base de Dados maiores
 - Pode ser caro
 - Existem limites para o quanto podemos expandir uma unica maquina
- Construir clusters de várias máquinas mais pequenas
 - Cada máquina é cara mas potencialmente unreliable
 - Precisamos que o DBMS (database management system) perceba que a BD está partida em clusters

IV. Relational Databases – Fundamental Issues

Os Relational Database Management Systems têm problemas fundamentais relacionados com escalabilidade e Impedance Mismatch.

No que toca a **Escalabilidade Horizontal**:

- Foram desenhados para trabalhar numa unica maquina grande
- São dificeis de distribuir de forma eficiente pelo que clusters são dificeis de implementar

Relativamente à **Impedance Mismatch**:

- Criamos estruturas logicas em memoria e depois partimo-las para as poder colar a uma RDBMS
- O RBDMS Data Model por vezes é disjunto do seu uso pertendido
- Normalization S U C K S
- É inconfortavel para programar (fuck SQL >:()

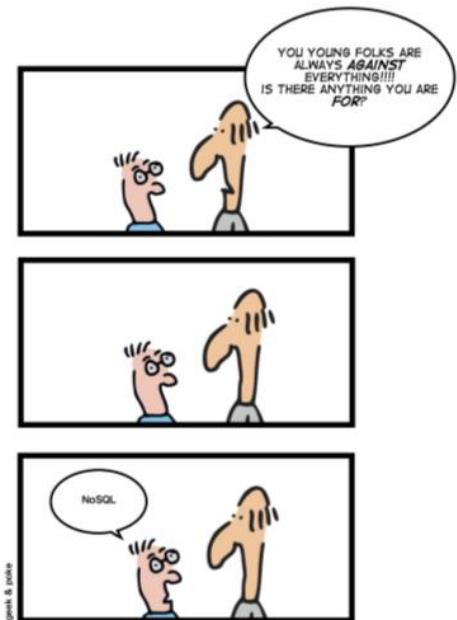
V. O Movimento NoSQL

Titulo kinda misleading visto que NoSQL não se refer a nenhuma tecnologia em especifico. Um titulo melhor seria something like “NotOnlySQL”

De qualquer das formas, varios modelos de base de dados estão agora associados com a #NoSQL visto que esta se espalhou rapidamente.

Atributos chave do NoSQL incluem:

- Non Relation
- Simple API
 - No Join Operations (HURRAY)
- BASE & CAP Theorem
 - Não tem requisitos ACID
 - Mais sobre estes teoremas à frente
- Schema-Free
 - Schema Implicito,
Application side
- Inherently Distributed
 - Alguns mais do
que outros
- Open Source
 - Mostly...



5.1 – Literally Me

Alguns **tipos core** de **NoSQL** DB's incluem:

- **Key-Value** Stores
- **Document** Stores
- **Column** Stores
- **Graph** Stores

Alguns **tipos non-core** de **NoSQL** DB's incluem:

- **Object** Databases
- **Native XML** Databases
- **RDF** Stores
- ...

V.I Transações BASE

Acronimo criado para ser o oposto do ACID

- **Basically Available**
- **Soft State**
- **Eventually Consistent**

Caracteristicas:

- Consistencia Fraca – Stale Data is OK
- Availability vem primeiro
- Best Effort
- Respostas aproximadas são OK

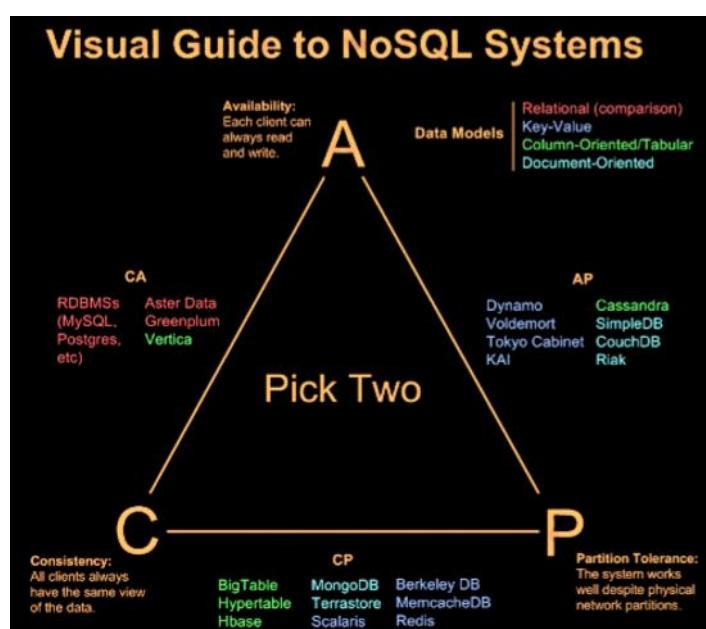
- Agressivo (optimista)
- Simples e rápido

V.II Brewer's CAP Theorem

Teorema que afirma que um Sistema distribuído só pode ser duas das seguintes três coisas:

- **Consistente**
 - Escritas atomicas
 - Todos os pedidos subsequentes vão buscar o valor mais recente
- **Available**
 - Database vai sempre retornar valores desde que o servidor esteja a correr
- **Partition Tolerant**
 - O sistema funciona mesmo que a network cluster esteja particionada (i.e se a cluster perder contacto com partes de si propria)

5.2 – Prioridades do CAP escolhidas por sistemas NoSQL



VI. NoSQL DB – Key-Value Databases

Data Model:

- É o mais simples NoSQL DB Type
- Simples Hash Table (mapping)

Key-Value Pairs:

- **Key** (aka id) – Normalmente é uma String
- **Value** – Pode ser qualquer coisa, texto, estrutura, imagem, lista, ...

Query Patterns:

- Criar, atualizar ou remover valores de uma dada key
- Ir buscar os values de uma dada key

Características:

- Ótimo performance
- Facilemente escalável
- Não é grande coisa para queries ou dados complexos

Use Cases:

- Session data, user profiles, user preferences, shopping carts, ...
- Basically tudo no qual os values saem acessíveis através de keys

Quando NÃO usar:

- Precisamos de ter relações entre entidades
- Queries requerem acesso a conteudos da parte dos values
- Operações do set envolvendo multiplos Key-Value Pairs

Exemplos:

- Redis

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

6.1 – Representação dos Dados numa Key-Value DB

VII. NoSQL DB – Document Databases

Data Model:

- Estrutura complexa auto-descritiva de dados
- Estrutura Hierarquica em Arvore (JSON, XML,...)
 - Valores escalares, mapas, listas, sets, nested documents, ...
- Identificados por um único identifier (key, _id,...)

Document Data stores percebem os seus documentos:

- Queries podem ser corridas contra valores dos document fields
- Indexes podem ser construidos para os document fields

Query Patterns:

- Criar, atualizar ou remover um documento
- Retrieve documentos de acordo com queries complexas

Diferenças dos Key-Value stores:

- São quase como Key-Value stores extendidos
- Mas parte dos values é examinavel ao contrario das key-value db's

Use Cases:

- Event logging, management systems de conteudos, blogs, web analytics, aplicações de e-commerce, ...
- Basically tudo o que apresentar documentos estruturados com schema similares

Quando NÃO usar:

- Operações que envolvam vários documentos
- Design da estrutura dos documentos que esteja sempre a ser mudada
 - i.e quando o nível de granulidade necessário fira um outbalance das vantagens de realizar as agregações

Exemplos:

- MongoDB

Document 1	Document 2	Document 3
{ "id": "1", "name": "John Smith", "isActive" : true, "dob": "1964-30-08" }	{ "id": "2", "fullName" : "Sarah Jones", "isActive" : false, "dob": "2002-02-18" }	{ "id": "3", "fullName" : { "first": "Adam", "last": "Stark" }, "isActive" : true, "dob": "2015-04-19" }

7.1 – Representação dos Dados numa Document DB

VIII. NoSQL DB – 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 row key única
- **Column**
 - Consiste no nome da coluna, valor da coluna (e possivelmente outros metadados)
 - Valores escalares, sets flat, listas ou mapas

Query Patterns:

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

Use Cases:

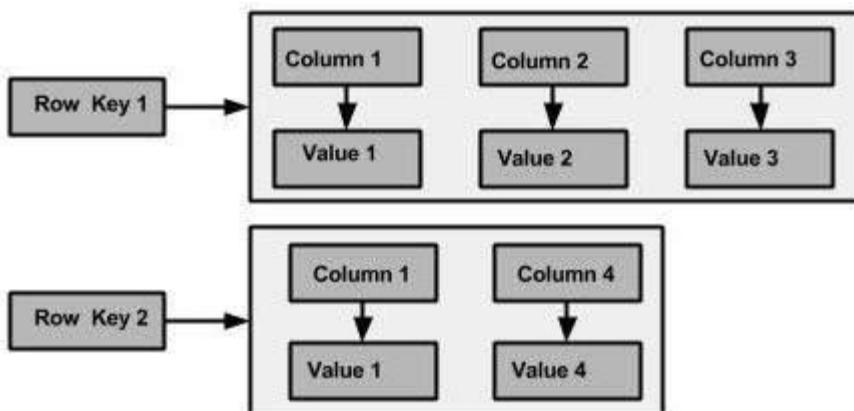
- Event logging, content management systems, blogs, ...
- Basically tudo o que for structured flat data com um schema parecido
- Batch processing via map reduce

Quando NÃO usar:

- Precisamos de ACID transactions
- Queries complexas (aggregation, joining, ...)
- Protótipos iniciais (quando o design da bd ainda está sujeito a mudança)

Exemplos:

- Apache Cassandra



8.1 – Representação dos Dados numa Column DB

IX. NoSQL DB – Graph Database

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 nos e as relações tem propriedades

Query Patterns:

- Criar, atualizar ou remover nos/relações do grafo
 - Algoritmos de Grafos
 - General Graph Traversals
 - Sub-Graph or Super-graph queries
 - Similarity based queries

Use Cases:

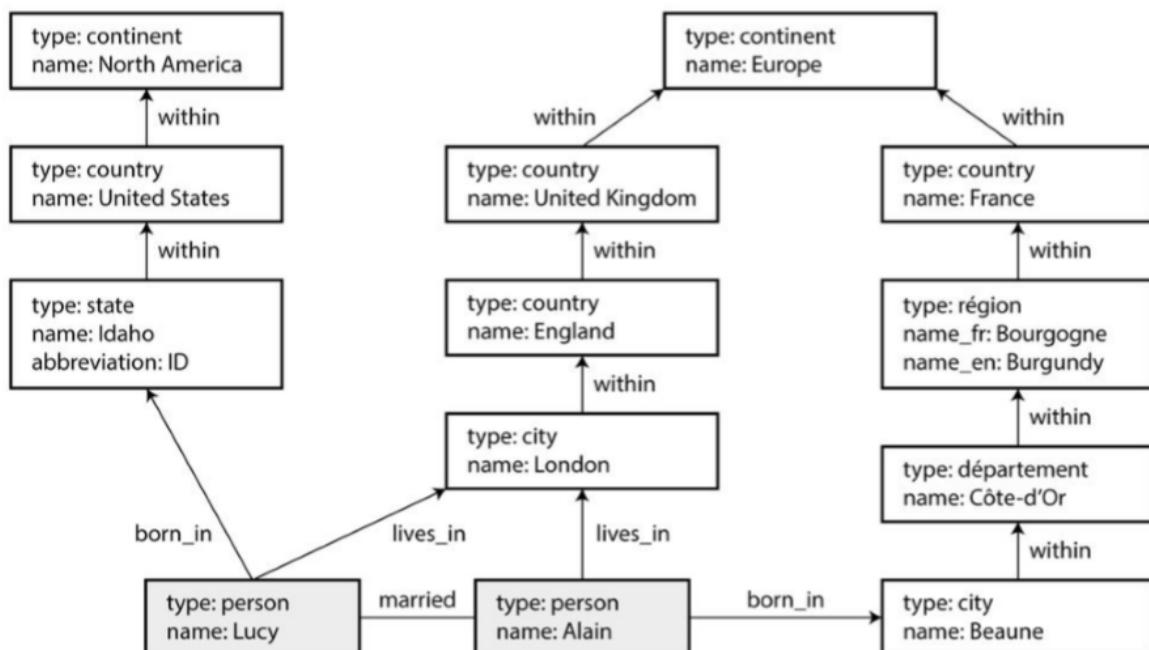
- Social networks, routing, dispatch, location based services, ...

Quando NÃO usar:

- Quando operações extensivas de batch são necessárias (multiplos nos/relações são afetadas)
- Vamos dar store de grafos demasiado grandes (a distribuição de grafos é difícil ou até mesmo impossível)

Exemplos:

- Neo4j



9.1 – Representação dos Dados numa Graph DB

X. NoSQL DB – Native XML Databases

Data Model:

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

Query Languages:

- Xpath
 - XML Path Language
- Xquery
 - XML Query Language
- XSLT
 - XSL Transformations

XI. NoSQL DB – RDF Databases

Data Model:

- RDF Triples
 - Subject, Predicate e Objeto
 - Cada Tripla representa uma statement sobre uma entidade do mundo real
- Triplas podem ser vistas em grafos
 - Vertices representam os subjects e objetos

- Edges correspondem diretamente às estamentes individuais

Query Languages:

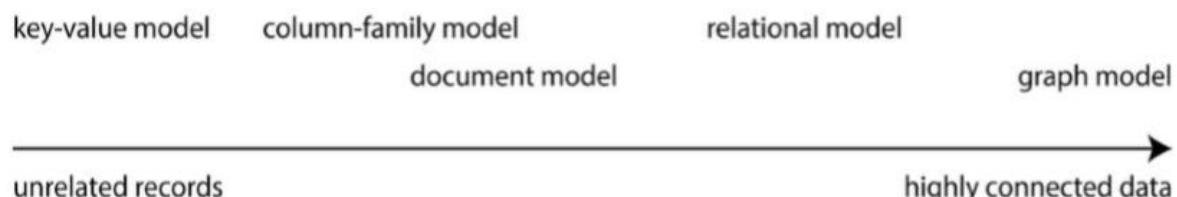
- SPARQL
 - SPARQL Protocol e RDF Query Language

XI. NoSQL DBs e conectividade de dados

· Relational model

· NoSQL models

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



11.1 – Comparaçāo do quāo conectados os dados estāo em cada um dos modelos apresentados

XII. NoSQL DBs – Considerações Finais

As NoSQL DBs NÃO são o fim das bases de dados relacionais (unfortunately <_<)

Estas continuam a ser preferiveis para 90% dos projetos, para alem do facto das pessoas estarem mais familiarizadas, serem mais estaveis, terem mais features e mais documentação e suporte disponivel.

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

Persistencia Poliglota

- Uso de diferentes data stores em diferentes circunstancias

Storage and Retrieval de Dados I – Data Structures & Indexes

I. Objetivos

Neste capítulo vamos abordar tópicos como:

- Como Guardar dados
- Como voltar a encontrar os dados guardados

II. Data Structures - Bash

Vamos, nesta secção criar um data store de key-values muito simples utilizando Bash.

Considerando o seguinte código Bash:

```
#!/bin/bash

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

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

2.1 – Exemplo de um key-value store simples feito em bash

Neste caso, a nossa key e value podem ser quase tudo o que quisermos, por exemplo:

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

2.2 – Exemplo de como usar as funções bash que criamos para adicionar e ir buscar dados

Como podes adivinhar, este formato de storage é muito simples. Basciamente o que estamos a fazer é **criar um ficheiro de texto em que cada linha contém um par key-value, separado por uma vírgula.**

Cada chamada à **cbd_set()** vai fazer append ao **final desse ficheiro**, pelo que dados inseridos previamente não são perdidos

```
$ 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"]}
```

2.3 – Visualização de como o nosso ficheiro de texto fica após as inserções realizadas

No que toca a performance, a nossa cbd_get até é bastante eficiente (para algo que é tão simples quanto 1 linha de código Bash).

Fazer append a um ficheiro é por norma uma operação eficiente, ao ponto de que a maioria das BDs usam append-only data files ou logs (more on that later)

O problema aqui está no facto de que, à **medida que a DB cresce, a cbd_get function tem um performance não muito desejável** – $O(n)$

Para **encontrar eficientemente** o valor da chave precisamos de uma **data structure** diferente – um **Index**

Um Index é uma estrutura adicional derivada da estrutura de dados primaria

Indexes bem escolhidos podem **diminuir drasticamente o tempo das Read Queries**, mas é de

notar que **por cada Index criado, estamos a tornar os nossos Writes mais lentos.**

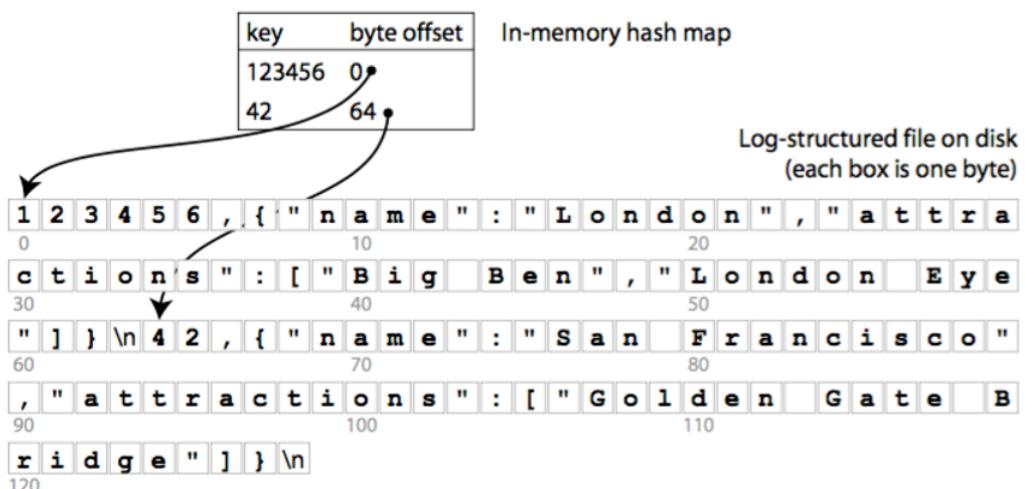
III. Hash Indexes

Podemos olhar para os Key-Value stores como um dicionario que mapeia uma Key a um Value.

Normalmente estes são implementados utilizando Hash Maps

Uma estratégia de indexação bastante simples é:
Manter um in-memory Hash Map na qual cada chave está mapeada a um byte offset no data file

Esta estratégia é adotada por algumas key-value databases visto vir a oferecer high performance reads E writes, caso o hash map seja mantido, na sua totalidade, em memoria



3.1 – Como é que Hash Indexes funcionam

IV. Gestão do espaço do disco

Como é que evitamos ficar sem espaço no disco?

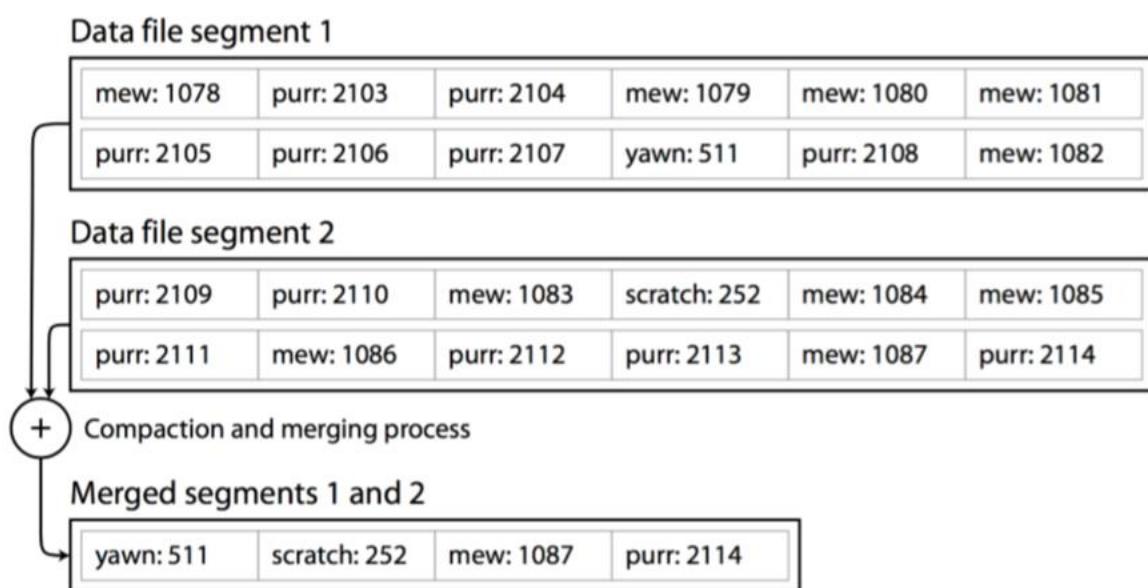
Well podemos:

- **1) Criar segmentação**

- Para um performance melhor
- Cada segmento vai conter todos os valores escritos na BD durante um periodo de tempo

- **2) Realizar Compactação**

- Apagar chaves duplicadas nos logs
- Compactação e Merging podem ser feitos numa thread em background enquanto continuamos a servir read e write requests como normal, utilizando old segment files



4.1 – Merging e Compactação

V. Append-Only Logs

À primeira vista, append-only designs podem parecer um desperdicio – Porque não atualizar o ficheiro em vez de fazer overwriting do valor antigo com um novo valor?

Após uma analise mais educada podemos encontrar varias razões pelos quais é uma boa decisão usar append-only logs, porem:

- Appending e Segment Merging são operações de write sequenciais, pelo que são geralmente mais rapidos que random writes
- Concorrencia e Crash Recovery são muito mais simples se segment files forem append-only e imutaveis

Still, também tem os seus downsides:

- A Hash-Table tem de conseguir caber em memoria
- É dificil fazer com que uma on-disk hash map tenha bom performacne
- Range queries não são eficientes (e.g procurar por todas as chaves de 01 até 99)

VI. Outros problemas com esta Data Structure

- **File Format**
 - CSV não é o melhor formato para um log
 - Um formato Binário pode ser utilizado como alternativa
- **Apagar Records**
 - Para apagar a chave precisamos de fazer append de um Special Deletion Record no data file (tombstone)
 - Quando log segments são merged o processo descarta quaisquer valores antigos para a deleted key
- **Crash Recovery**
 - Se o sistema for reiniciado os in-memory hash maps são perdidos
 - Para acelerar a velocidade de recuperação podemos criar snapshots do hash map de cada segmento no disco
- **Partially Written Records**
 - A BD pode crashar a qualquer momento, inclusive a meio do appending de um record ao log
 - Checksums permitem que partes corrompidas do log possam ser detetadas e ignoradas

- **Controlo de Concorrencia**

- Visto que os writes são appended de forma sequencial, não podemos ter mais do que uma Writer Thread
- Data File Segments são append only e imutaveis, pelo que podem ser concorrentemente lidos por varias threads

VII. Data Structures – Sorted String Table

Nos exemplos que temos usado, os key-value pairs aparecem na ordem em que são escritos (i.e appended)

Podemos, porem, assegurar que uma sequencia de key-value pairs está **sorted pela key**

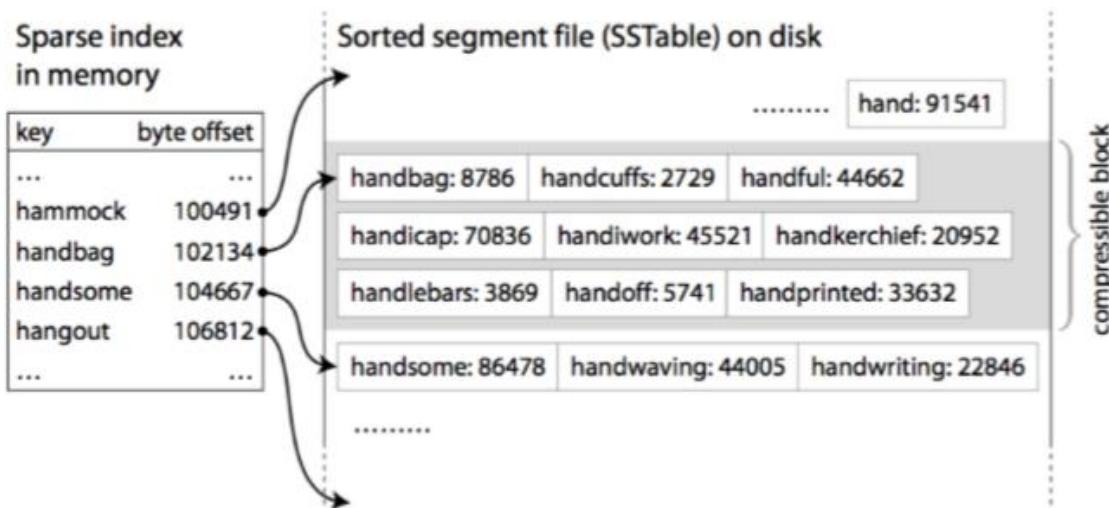
Chamamos a este formato **Sorted String Table**, ou **SSTable**

Este formato requer que cada chave apareca apenas uma vez por cada merged segment file, mas o processo de merging já nos garante isso sooo its tranquilo.

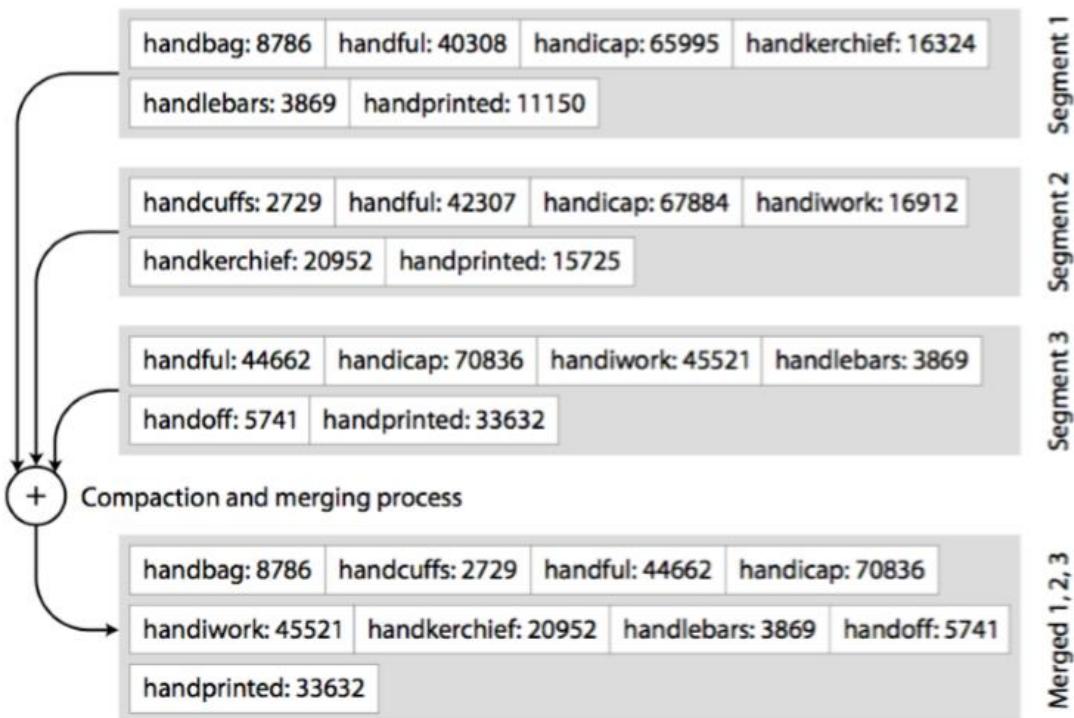
À primeira vista a habilidade das keys estarem sorted aparenta ir contra o facto dos writes serem sequenciais, but we'll get to that in a moment

SSTables têm várias vantagens sobre Log-Segments com Hash Indexes:

- Merging de segments é mais simples e eficiente, mesmo com ficheiros maiores do que os disponíveis em memoria (mergesort algorithm)
- Não precisamos de manter um index de todas as chaves em memória



7.1 – SSTable Index em memória e em disco



7.2 – Merging de SSTable Segments

Levanta-se agora a pergunta: **Como é que fazemos sort dos dados por Key sabendo que os writes podem ocorrer em qualquer ordem e sequencialmente?**

R: Podemos manter uma estrutura sorted em memória. Usando uma Tree Data-Structure (B-Trees, AVL, ...) podemos inserir Keys em qualquer ordem mas le-las de forma sorted.

Chamamos a esta in-memory tree uma **Memtable**

Quando a MemTable fica grande demais e ultrapassa o nosso threshold de tamanho maximo, escrevemos-la em disco como um ficheiro SSTable.

Isto pode ser feito de forma eficiente porque a Tree já contem os Key-Value pairs sorted por Key

O novo SSTable File torna-se o mais recente Segment da BD

Quando a nova SSTable for escrita e estiver pronta, a MemTable pode ser esvaziada

Para servir um Read Request, primeiro procuramos pela Key na Memtable, depois no mais recente on-disk segment e de seguida no próximo mais recente segment, and so on.

De tempo a tempo, corremos um processo de Merging e Compaction em background para combinar SSTable Segment Files e descartar Overwritten ou Deleted Values

Este esquema sofre de **um problema**:

- Se a BD Crashar, os Writes mais recentes (que estão na memtable mas ainda não foram escritos para o disco) são perdidos :(

Podemos evitar este problema mantendo um log separado no disco no qual todos os writes são imediatamente appended. Sempre que uma memtable é escrita para uma SSTable, o log correspondente pode ser descartado

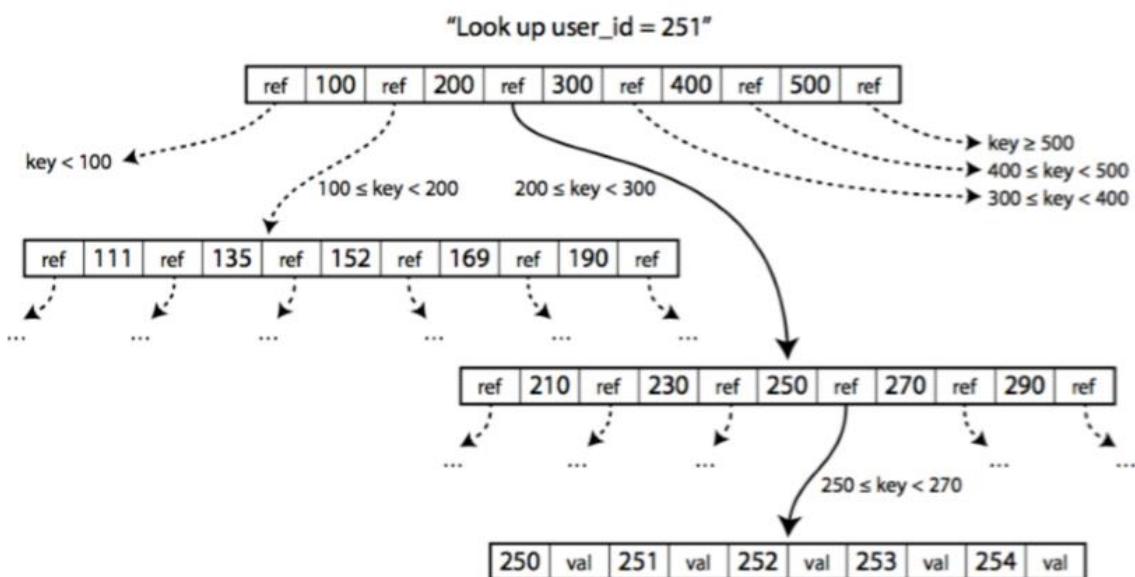
VIII. Data Structures - B-Trees

São uma implementação standard para Indexes em quase todas as Relational e non-relational DBs

Tal como SSTables, B-Trees mantem os Key-Value Pairs sorted por Key, permitindo key-value lookups e range queries eficientes

B-Trees **partem a BD num tamanho fixo de blocos** (ou **pages**), tradicionalmente de 4kB, e escrevem/leem uma page de cada vez

Para quem já teve a cadeira de Sistemas Operativos, sabe que isto é bastante parecido com o que acontece em Hardware, visto que os discos também estão organizados por blocos de tamanho fixo



8.1 – B-Tree Structure e Lookup example

Em termos de organização da estrutura:

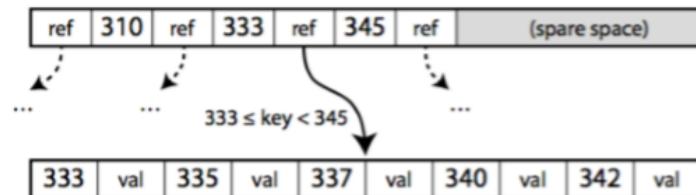
- A estrutura começa numa **Root Page**
- Cada Page contem **K keys e K+1 references para Child Pages**
- Cada Child corresponde a um range continuo de chaves
- As Keys na Root Page indicam quais os limites desse range

Em termos de operações sobre a estrutura:

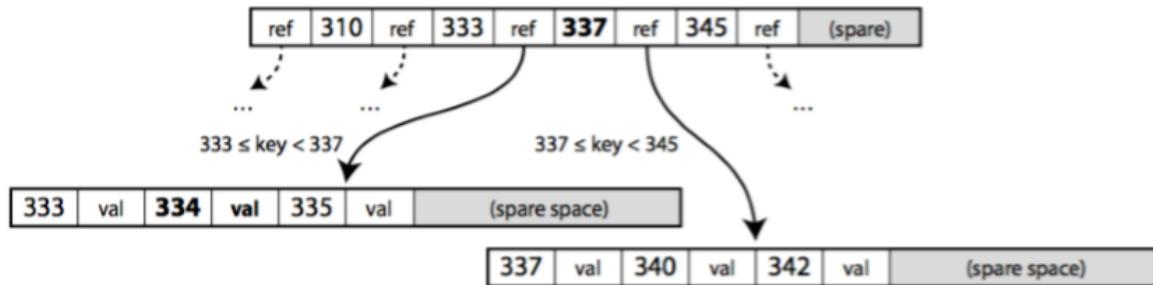
- **Adicionar uma nova chave**
 1. Encontramos a Page cujo range englobe o valor da nova Key (as in o id da chave e não o seu value correspondente) e adicionamos essa key à Page
 2. **Caso não exista espaço suficiente** nessa page, esta **full Page será partida em duas Half Full Pages** e a Parent Page será atualizada para acomodar a nova subdivisão dos Key Ranges
- **Atualizar o Value de uma Existing Key**
 1. Procuramos pela Leaf Page que contenha a chave e mudamos o valor dessa pagina

2. Escrevemos a pagina de volta ao disco
 (quaisquer referencias a essa pagina
 mantem-se validas)

Os algoritmos de qualquer operação assegura que a B-Tree se mantém **Balanced** – i.e Uma B-Tree com n keys terá sempre uma altura de $O(\log n)$



After adding key 334:



8.2 – B-Tree Page Splitting Example

IX. B-Trees – Update in Place vs Append Only Log

A underlying write operation base de uma B-Tree é de fazer overwrite de uma Page em disco com novos dados

Por outro lado, log-structured indexes so fazem append a files mas nunca modificam files in place (são imutaveis)

Algumas operações requerem que varias Pages sejam overwritten.

Se houver um crash aquando da escrita causando que apenas algumas das Pages sejam escritas, vamos acabar com um **Corrupted Index** – e.g uma Orphan Page que não é child de nenhum parent

Para lidarmos com crashes é normal incluirmos um Write-Ahead Log (WAL) – Um **Append-Only file** para o qual todas as modificações da B-Tree são escritas antes de serem applied à Tree

Uma complicação adicional no que toca ao update de pages in-place é que precisamos de um **controlo de concorrencia cuidado**. Se multiplas threads vao aceder a

B-Tree ao mesmo tempo, a Thread pode acabar por ver uma tree num estado inconsistente

Este controlo é tipicamente realizado protegendo a estrutura de dados da Tree com **Latches** (lightweight locks).

Neste caso, uma log-structured approach é mais simples: O merging e swapping ocorrem em background sem interferir com incoming queries

X. B-Trees – Optimizations

- **Copy-on-Write Scheme**
 - Para lidar com crash recovery uma modified page é escrita num local diferente e uma nova versão das parent pages na tree são criadas, apontando para esse novo local
- **Guardar espaço nas pages abreviando a Key em vez de a guardar na sua totalidade**
 - Incluir mais keys por page permite que a tree tenham um maior branching factor levando a menos níveis
- **Variantes da B-Tree**
 - Tais como Fractal Trees podem dar uso das mesmas log-structured ideas para reduzir disk seeks

XI. B-Trees vs SSTable (LSM-trees)

LSM Trees têm tipicamente writes mais rápidos

B-Trees tendem a ter reads mais rápidos

Log-Structured Storage (LSM-Trees) tem um processo de compactação que pode por vezes interferir com o performance de ongoing reads e writes

Em **B-Trees** cada key existe em exatamente um local no index.

Em **Log-Structured Storage** podemos ter varias copias em varios segment

Isto torna as **B-Trees** mais atraentes em DB nas quais queremos oferecer **fortes semanticas transacionais**

Em muitas BD Relacionais, isolação de transações é implementada utilizando locks em ranges de keys. Numa B-Tree Index, esses locks podem ser diretamente attached à tree

XII. Outras estruturas de Indexing

É muito comum a utilização de **Secondary Indexes**

A principal diferença entre estes e os que vimos até agora é o facto das Keys não serem unicas, i.e, pode haver mais do que uma row com a mesma key

Isto pode ser tratado de duas formas:

- Fazer com que cada value no index uma lista de matching row identifiers
- Tornar cada key unica adicionando-lhe um row identifier

Tanto B-Trees como Log Structured Indexes podem ser usados como secondary indexes

XIII. Guardar Values num Index

As Keys são os ids pelos as queries procuram, mas o Value pode ser uma de duas coisas:

- A Row em questão
- Uma referencia para a Row, guardada noutro local qualquer
 - Neste caso é conhecido como **Heap File**

- Data Duplication é avoided quando multiplos secondary indexes estão presentes
- Ao atualizar um value, a Heap File approach pode ser muito eficiente
- O record pode ser overwritten in-place, caso o novo value não seja maior que o old value

Por vezes pode ser desejavel guardar a indexed row diretamente num index. Chamamos a isto **Clustered Index**

A chave primaria de uma table pode ser uma clustered index e os secondary indexes podem referir-se à primary key (em vez de à localização do Heap File)

Existe um compromise entre Clustered e Non-Clustered Index conhecidos como **Covering Index**.

Estes guardam algumas das colunas da tabela dentro do index

Estes indexes podem aumentar a velocidade dos read mas:

- Requerem storage adicional e overhead nos writes

- BDs precisam de esforços adicionais para enforçar Transactional Guarantess por causa da duplicação

XIV. Multi-Column Indexes

Os indexes que temos vindo a discor mapeam uma unica Key a um Value

Isto por vezes não é suficiente, especialmente se quisermos fazer uma query a varias colunas de uma table (ou multiplos fields num documento) simultaneamente

O mais comum tipo de **Multi-Column Index** chama-se um **Concatenated Index**

Este combina varios fileds numa unica Key fazendo appending de uma coluna noutra

Assemelha-se a uma lista telefonica que fornece um index de (lastname, firstname) para um phone number

Este index pode ser usado para encontrar todas as pessoas com um lastname especifico, u todas as pessoas com uma combinação lastname-firstname especifica

XV. Fuzzy Indexes

Todos os indexes que temos vindo a discutir assumem que queremos fazer queries para valores exatos de keys ou para um range de valores de uma key com uma sort order

Mas e se quisermos procurar por keys parecidas a um certo valor? P.ex procurarmos por palavras mesmo que estas estejam misspelled?

Existem data structures que permitem pesquisar por texto dentro de uma certa edit distance – E.g Lucene

O lucene usa uma SSTable-like structure para o seu dicionario de termos.

Esta estrutura diz as queries qual o offset num sorted file no qual tem de procurar pela key.

Este é um automato de estados finito sobre os caracteres nas chaves que suporta pesquisa de palavras de forma eficiente dentro de uma dada edit distance

XVI. Manter tudo em memória

Para muitos datasets é fazivel manter todos os dados em memoria, até mesmo potencialmente distribuidos por várias maquinas

Isto levou-nos ao desenvolvimento de **in-memory databases**

Alguns in-memory key-values stores, tais como Memcached, foram **feitos para Caching Use Only – Quando é aceitável que dados sejam perdidos ao reiniciar a máquina**

Outros porém, procuram **Durability**.

Ao escrever um log of changes em disco, usando snapshots do disco periódicos ou fazendo replicação do in-memory state para outras máquinas

VoltDB, MemSQL and Oracle TimesTen are in-memory databases with a relational model.

RAMCloud is an open-source in-memory key-value store with durability (using a log-structured approach for the data in memory as well as the data on disk).

Redis and Couchbase provide weak durability by writing to disk asynchronously.

16.1 – Algumas soluções in-memory

No que toca a vantagens, as in-memory DBs ganham performance, não pelo facto de não terem que ler do disco (até mesmo disk-based storage engines

podem nunca precisar de ler dos discos se tiverem memoria suficiente), mas pelo facto de poderem ser mais rapidas devido a evitarem overheads no encoding de in-memory data structures de forma a poderem escrever no disco.

Para além de performance, oferecem data models que são mais dificeis de implementar com disk-based indexing

Mantendo todos os dados em memoria a implementação é comparativamente simples.

Convém também dizer que **escrever no disco também tem vantagens operacionais** – Ficheiros em disco podem ser backed up facilmente, inspecionados ou analizados por utilities externas

Storage and Retrieval de Dados II – Transaction Processing & Transaction Analytics

I. Objetivos

Neste capítulo vamos abordar tópicos como:

- Como selecionar um storage engine apropriado para uma aplicação
- Diferença entre Storage Engines otimizados para transactional workloads e otimizados para analytics

II. Online Transaction Processing (OLTP)

Quando BDs começaram a surgir, escrever para uma DB costumava corresponder a uma **Transação** comercial (i.e fazer uma compra, criar um pedido a um supplier, pagar o ordenada a um empregado, ...)

Mesmo com a expansão das BDs para outras áreas, o termo **Transação** manteve-se.

Transaction Processing significa permitir a um cliente fazer low-latency reads e writes

Por outro lado, **Batch Processing Jobs** correm periodicamente (p.ex uma vez por dia)

Aplicações são, por norma, interativas (inserts, updates, searches, ...). Este padrão de acesso tornou-se naquilo que chamamos **Online Transaction Processing (OLTP)**.

III. Data Analytics

BDs também começaram a ser utilizadas cada vez mais para **Data Analytics** (que apresentam padrões de acesso completamente diferentes dos OLTP)

Normalmente, uma **Analytic Query** precisa de scannar um grande numero de records e calcular aggregate statistics.

Alguns exemplos incluem queries como:

- Quanto foi o total revenue em cada uma das nossas lojas no mes de Janeiro
- Quantas mais Playstations é que vendemos durante a ultima promoção, comparativamente às vendas normais
- Qual a brand de baby food que mais normalmente é comprada com a brand X de fraldas

IV. Online Analytic Processing (OLAP)

Analytic Queries são normalmente escritas por Business Analysts and alimentam relatórios que ajudam equipas de management de uma empresa a fazer melhores e mais informadas decisões – Business Intelligence

Para diferenciar este padrão de utilização de BDs do padrão de Transaction Processing, passamos a chamar a este **Online Analytic Processing (OLAP)**

V. OLTP vs OLAP

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

5.1 – Diferenças entre OLTP e OLAP

RDBMS e SQL funcionam bem para ambas OLT-type e OLAP-type queries

Porém as OLAP tendem a usar BDs separadas das OLTP – **Data Warehouse**

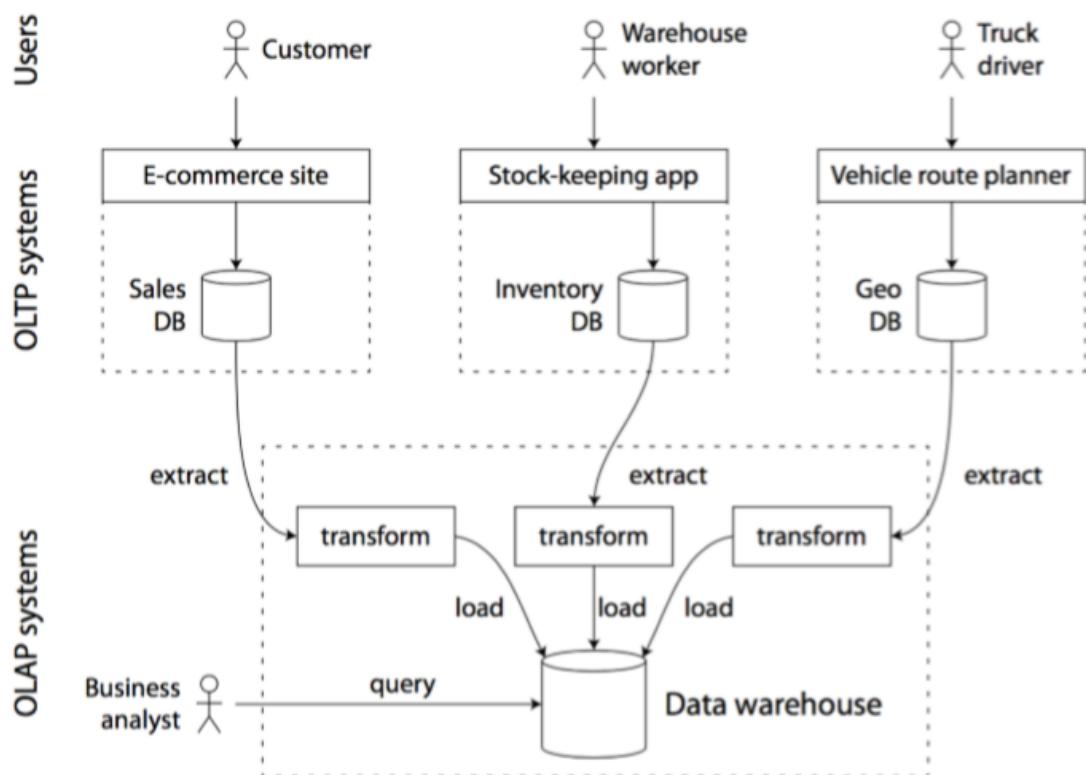
VI. Data Warehousing

Espera-se que os OLTP Systems ofereçam **High Availability** e que processem transactions com **Low Latency**

Uma **Data Warehouse** é uma BD separada sobre a qual analistas podem realizar queries sem after operações de OLTP

Tipicamente contem uma **copia read-only dos dados** em todos os sistemas OLTP da empresa

Dados são extraídos das BDs OLTP e transformados, limpos e carregados para uma **Data Warehouse**. A este processo chamamos **Extract-Transform-Load (ETL)**



6.1 – Exemplo de Data-Warehousing e do processo ETL

Tambem podemos argumentar que se devem combinar o OLTP e o OLAP:

- SQL pode ser usando tanto para normal queries como para analytic queries
- Existem muitas ferramentas para OLTP (querying, visualization, ...) que também podem ser usadas para OLAP

Mas então, porque que devemos separar o OLAP do OLTP em Data Warehouses?:

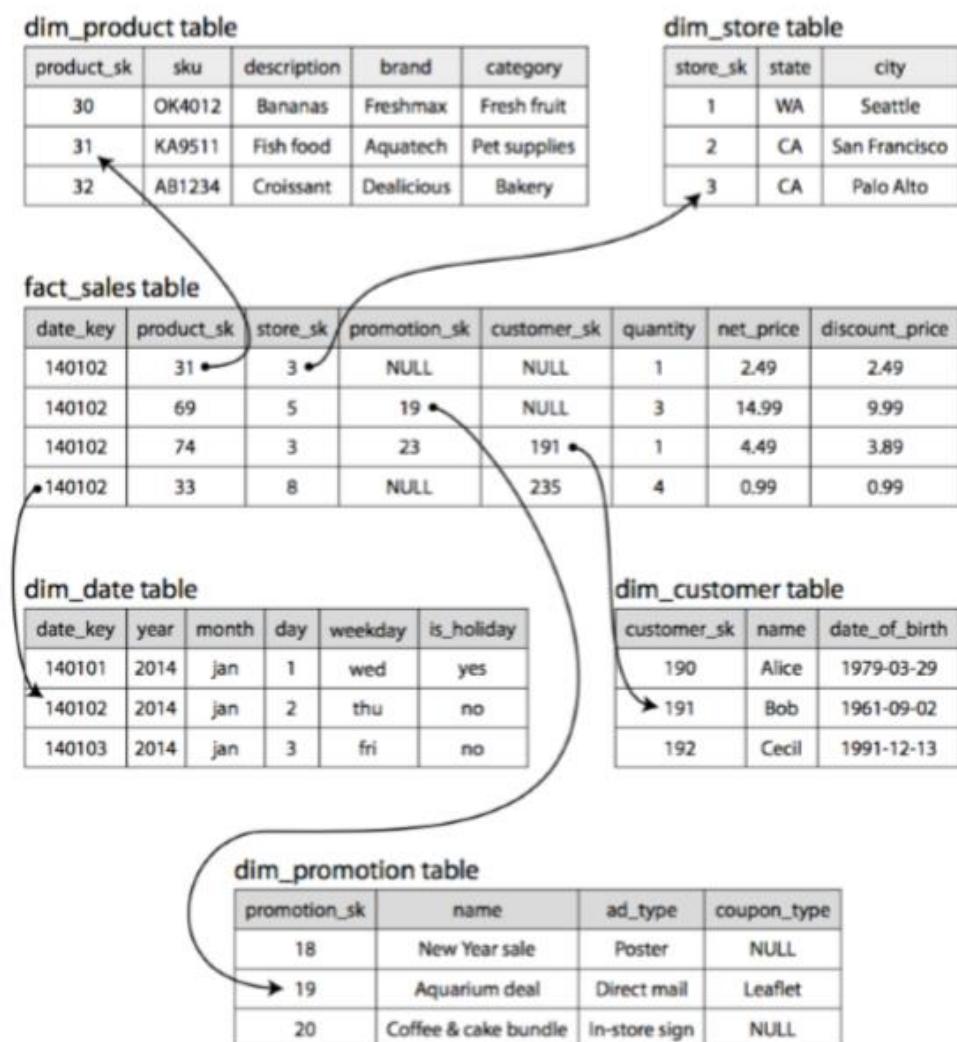
- **Different Functions e Different Data**
 - **Missing Data**
 - Suporte de decisões requer um historico de Dados que as BDs operacionais por norma não mantem
 - **Data Consolidation**
 - Suporte de decisões requere consolidação (aggregation, summarization) de dados de sources heterogeneas
 - **Data Quality**
 - Sources diferentes tipicamente usam representações de dados inconsistentes, codigos e formatos que têm de ser reconsolidados

VII. Schemas para Analytics

Muitas Data Warehouses usam o **Star Schema** (aka Dimensional Modeling)

A entidade principal é a **Fact Table**

- Cada Row da Fact Table representa um evento que ocorreu num timestamp particular (uma compra, uma visualização de uma pagina, etc)
- Algumas colunas são atributos (e.g preço da compra), e outras são referencias (foreign keys) para outras tabelas (dimensional tables)



Existe uma variação do Star Template conhecida por **Snowflake Schema**

Neste, as dimensions são partidas em sub-dimensions (criando um schema mais normalizado mas mais complexo com que trabalhar)

Um problema com as **Fact Tables** é o facto de serem, tipicamente, ENORMES, com trilions de rows e petabytes de dados e centenas de colunas.

Storing e Querying torna-se um grande problema

Porém, uma **Data Warehouse Query** por norma só costuma a aceder a alguns desses dados de cada vez

Problem:

Precisamos de ler a tabela inteira para processar uma unica coluna que seja

OLTP DBs são tipicamente row-oriented – Todos os values de uma row de uma table são guardados lado a lado

Solution:

Column-Oriented Storage

VIII. Column-Oriented Storage

A ideia deste tipo de storage é:

- Não guardar todos os values de uma row juntos
- Guardar os values de cada coluna juntos instead
- Se cada coluna for guardada num ficheiro separado, uma query a uma coluna passa a só precisar de ler e parsear todas essas colunas

O Column-Oriented Storage Layout precisa que cada coluna contenha as rows na mesma ordem

Para fazer reassemble de uma row inteira, n, precisamos de pegar em todas as entradas n de cada ficheiro individual que contenha cada uma das colunas

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

8.1 – Exemplo do uso de Column Storage Layout

Uma boa característica do Column-Oriented Storage é o facto de se ceder muito bem a **compressão**

As sequências de valores para cada coluna são muitas vezes repetitivas

Dependendo dos dados de cada coluna, diferentes técnicas de compressão podem ser usadas

Uma das técnicas particularmente eficiente em data warehouses é o **Bitmap encoding** (more on compression no próximo capítulo)

IX. Column-Oriented Storage – Column Sorting

Normalmente, as colunas são guardadas por **ordem de inserção**

Porém podemos escolher outra ordenação utilizando um dos valores das colunas – Isto serve como **mecanismo de indexação**

Um segunda coluna pode determinar a ordem de ordenação de qualquer row que tenha o mesmo value na primeira sorting column

P.ex se uma date_key for a primeira sort key, e product_sk for a segunda, todas as sales com o mesmo produto no mesmo dia serão agrupadas juntas

Outra vantagem de usar sorted order é que pode ajudar com a compressão das colunas

X. Column Oriented Storage – Writing Problem

Como já referimos, **Column-Oriented Storage**, **Compression** e **Sorting** ajudam a tornar as **OLAP Queries** mais rápidas

Mas então..e os Writes?

- As rows são identificadas pela sua posição dentro da coluna, a inserção tem de atualizar TODAS as colunas de forma consistente...

Qual é a melhor estrutura?

- Uma **Update-In-Place** approach, tais como as que B-Trees usam, **não é possível em compressed columns**
- Se quisessemos inserir uma row no meio de uma osrted table, teríamos que reescrever todos os column files...

XI. Column Oriented Storage – Writing Problem Solution

Uma boa solução é o uso de **LSM-Trees** (Log Structured Merge)

1. Todos os Writes primeiro vão para uma in-memory store, onde serão adicionados a uma sorted structure e preparados para serem escritos em disco
2. Quando writes suficientes tiverem sido acumulados, são merged com os column files em disco e escritos para novos ficheiros em bulk
3. Queries examinam ambos os column data em disco e memória

XII. Materialized Views

Data Warehouse Queries envolvem uma **aggregate function** (count, sum, avg, min, ...) na maioria dos casos

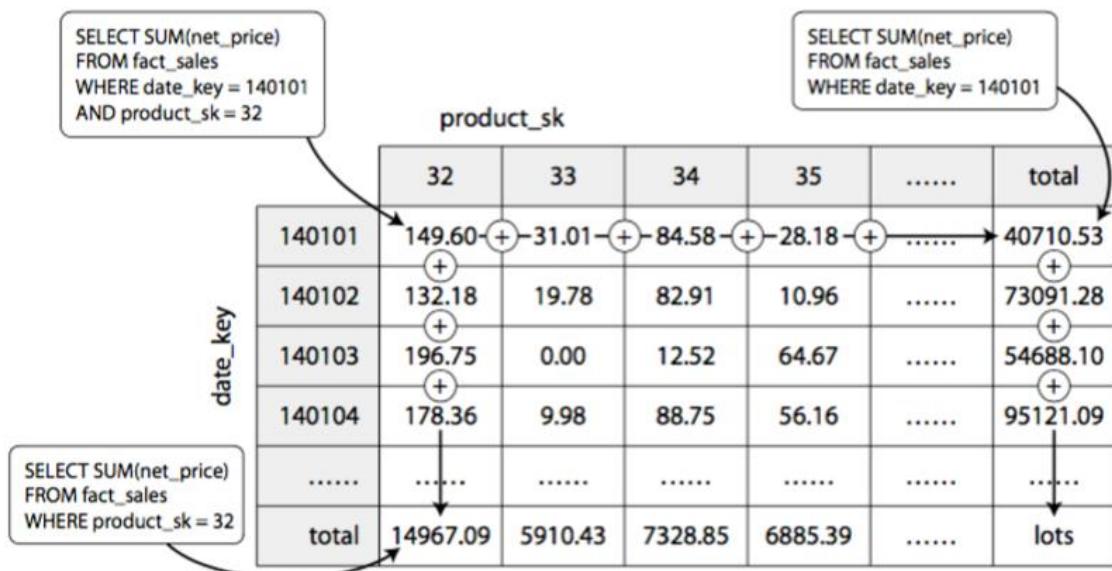
Aggregates podem ser usados por muitas queries, o que nos leva a repetir o procesamento de dados..

Solution?

Criar uma **Cache com os resultados destes aggregates** naquilo que se chama uma **Materialized View**

Quando a underlying data mudar, a materialized view tem, obviamente de ser atualizada

Um **Data Cube** (aka OLAP Cube) é um caso específico de Materialized Views. Este consiste numa grid de aggregates agrupados por diferentes dimensões



12.1 – Exemplo de um Two-Dimensional Data cube, aggregation data by summing

Data Formats

I. Objetivos

Neste capítulo vamos abordar os seguintes formatos de encoding de dados:

- CSV
- XML
- JSON
- BSON
- RDF
- Protocol Buffers

II. Data Encoding

Software applications inevitavelmente mudam ao longo do tempo

Estas mudanças muitas vezes requerem que os dados sejam também mudados.

Existe a possibilidade que versões antigas e novas do código e dos formatos de dados coexistam num sistema ao mesmo tempo

Para que o sistema continue a funcionar mesmo com esta evolução e coexistencia, é preciso que compatibilidade seja mantida em ambas as direções:

- **Backwards Compatibility**

- Código novo consegue ler dados escritos por código mais antigo

- **Forwards Compatibility**

- Código antigo consegue ler dados escritos por código mais recente
- Requer que o código antigo consiga ignorar additions feitas por novas versões do código

Os programas tendem a trabalhar com dados:

- **In-Memory**

- Dados mantidos em objetos, estruturas, listas, arrays, trees, hash tables, etc

- **Out of Memory**

- Escritas de dados em ficheiros ou envio de dados por network

A translação de representação in-memory para uma sequencia de bytes chama-se **Encoding** (aka Serialização ou Marshalling)

O oposto diz-se **Decoding** (aka Deserialization, unmarshalling ou parsing)

III. Language Specific Formats

Muitas linguagens de programação vem com built-in support para fazer encoding de in-memory objects para byte sequences – Java: Serialization ; Python: Pickle; ...

Estas bibliotecas de encoding são bastante convenientes, mas torna a leitura de dados encoded numa linguagem dificeis de realizar numa linguagem diferente.

Ou seja, a utilização destas bibliotecas prende-nos a estarmos comprometidos a utilizar a mesma linguagem de programação

Por esta razão, não é recomendavel usar estes built-in encoding libraries para tudo o que não sejam **transient purposes** (i.e, quando queremos um encoding temporario por alguma razão, mas sabemos que inevitavelmente vamos ter que usar outro tipo de encoding)

IV. Encoding Formats

IV.I Textual

Têm a vantagem de ser **human-readable**

Exemplos incluem: CSV, JSON, XML e RDF

Alguns problemas incluem:

- **Ambiguidade entre um numero e uma string**
 - JSON kinda resolve isto mas não distingue integers de floating points, i.e falta-lhe precisão
- **Não existencia de Schema**
 - O CSV sofre desta falha
 - Cabe à aplicação definir o significado de cada row/column

IV.II Binary

Têm a vantagem de ser mais compacto e com um parse (decoding) mais rápido

Exemplos incluem: BSON, BJSON, MessagePack, ...

Por muito mais eficientes que sejam, para datasets mais pequenos os ganhos são nulos – Apenas são visíveis quando começamos a ter terabytes de dados

Mesmo assim não são tão widespread quanto as formatos textuais

IV. CSV – Comma Separated Values

Content Type: text/csv

File Extension: *.csv

Infelizmente não está completamente standardizado

- Existem muitos field separators possíveis (e.g , \t ;)
- Existem diferentes escaping sequences
- Não há encoding de informação

```
firstname,lastname,year
Ana,Katrina,1974
Paul,Machado,1956
Luis,Morais,1974
Sofia,Silvasky,1986
Maria,Marinova,1976
```

4.1 – Exemplo do conteúdo de um CSV file

V. XML – Extensible Markup Language

Content Type: text/csv

File Extension: *.xml

Representação de dados semi-estruturados

Design Goals incluem

- Simplicidade
- Generalibilidade
- Usabilidade pela internet

```
<?xml version="1.1" encoding="UTF-8"?>
<movie year="2007">
    <title>The Great Marnoto</title>
    <actors>
        <actor>
            <firstname>Jakim</firstname>
            <lastname>Dalmeida</lastname>
        </actor>
        <actor>
            <firstname>Sofia</firstname>
            <lastname>Ravara</lastname>
        </actor>
    </actors>
    <director>
        <firstname>Paulo</firstname>
        <lastname>Castanho</lastname>
    </director>
</movie>
```

5.1 – Exemplo do conteúdo de um XML file

XML constructs incluem:

- **Elements**

- Marcados com o formato
 - <element_name> ... </element_name>
- Cada elemento pode estar associado a um set de atributos
- É precisos que estejam bem formados
- Alguns dos tipos de conteudos incluem:
 - Empty Content
 - Text Content
 - Element Content (sequence of nested elements)
 - Mixed Content (elements arbitrarily interleaved with text)

- **Attributes**

- Par Name-Value
- Ex: <person **gender="female"**> ...
- Existem também sequencias de Escape (entidades predefinidas) que podem ser usadas com values de attributes ou conteudos de texto dos elementos
 - E.g <, <, >, ...

Existem outros como comments, processing instructions, etc. Mas estes dois (mais Text) são os principais

VI. JSON – Javascript Object Notation

Content Type: application/json

File Extension: *.json

Standard aberto para intercambio de dados
derivado do Javascript (mas sendo language
independent)

Design Goals incluem

- Simplicidade
 - text-based, facil de ler e escrever
- Universabilidade
 - Suportado pela maioria das linguagens de programação modernas

6.1 – Exemplo do
conteudo de um Json
file

```
{  
    "title": "The Great Marnoto",  
    "year": 2007,  
    "actors": [  
        {  
            "firstname": "Jakim",  
            "lastname": "Dalmeida"  
        },  
        {  
            "firstname": "Sofia",  
            "lastname": "Ravara"  
        }  
    ],  
    "director": {  
        "firstname": "Paulo",  
        "lastname": "Castanho"  
    }  
}
```

Um JSON é construído de duas estruturas:

- **Coleção de pares Name/Values (i.e Object)**
 - Na maioria das linguagens isto é realizado como um objeto, record, struct, dictionary, hash table ou associative array
 - O objeto é portanto uma **coleção não ordenada de name-value pairs**

```
{ "name" : "Manuel Sliav", "year" : 2000 }  
{ }
```

6.2 – Exemplo de dois objetos

- **Ordered Collection of Values (i.e Array)**
 - Na maioria das linguagens isto é realizado como um array, vector, list ou sequence
 - Values podem ser de diferentes tipos e valores duplicados são permitidos

```
[ 2, 7, 7, 5 ]  
[ "Some person", 1979, 77 ]  
[ ]
```

6.3 – Exemplo de 3 arrays

- **Value**

- Pode tomar a forma de:
 - **String Unicode enclosed por “ “**
 - Permite uso de escaping sequences
 - E.g \n \t \" \\, ...
 - E.g “Ola\nComo estás :) ?”
 - **Numero**
 - Integers decimais ou floats
 - E.g 1, -0.5, 1.5e3
 - **Nested Array**
 - **Nested Object**
 - **Boolean Value**
 - True, False
 - **Missing Information**
 - null

```
{  
  "stuff": {  
    "onetype": [  
      {"id":1,"name":"John"},  
      {"id":2,"name":"Don"}  
    ],  
    "othertype":  
      {"id":2,"company":"ACME"}  
  },  
  "otherstuff": {  
    "thing": [[1,42],[2,2]]  
  }  
}
```

6.4 – Exemplo de values dentro de um object

VII. BSON – Binary Json

File Extension: *.bson

Binary-Encoded serialização de documentos JSON.

Usado pelo MongoDB!

Design Characteristics incluem

- Lightweight, traversable, eficiente
- Storage conveniente de informação binaria
(suitable para exchanging de imagens e attachments)
- Designed para manipulação rápida in-memory
- Mais data types que JSON
 - Double, Date, Byte array, JS Code, ...

JSON

```
{  
  "title" : "Marnoto",  
  "year" : 2007  
}
```

BSON

```
          t i t l e           M  
2200 0000 0274 6974 6c65 0008 0000 004d  
a r n o t o      y e a r    2007    // = 0x07d7  
6172 6e6f 746f 0010 7965 6172 00d7 0700  
0000
```

7.1 – Exemplo do conteúdo de um BSON criado a partir de um JSON

No que toca a Document Structure temos

- **Element**

- Serialização de uma propriedade do JSON

- **Structure**

- **Type Selector**

02 (string), 03 (object), 04 (array)

01 (double), 10 (32-bit integer), 12 (64-bit integer)

08 (boolean), 09 (datetime), 11 (timestamp)

0A (null)

t i t l e
0274 6974 6c65

...

7.2 – Type Selectors e exemplo utilizando o documento anterior (o tipo da propriedade identificada pelo nome ‘title’ é String)

- **Property Name**

- Unicode String terminado em 00

y e a r :
0010 7965 6172 00

7.3 – O tipo da propriedade identificada por ‘year’ é 32-bit int, e o nome da propriedade é identificado por 7965 6172 (está entre o type selector 10 e o 00)

- **Property Value**

- Vai desde o 00 até ao fim do ficheiro ou até a proxima vez que encontremos um Type Selector

VIII. RDF – Resource Description Framework

Linguagem usada para representar informação sobre resources na WorldWideWeb

Usada em Graph DBs e no contexto de Semantic Web, Linked Data, ...

Baseado no conceito de **cada resource ter diferentes propriedades que em si tem values**

- **Resource** – Qualquer real-world entity
 - **Referents**
 - Resource identificado por um IRI (Internationalized Resource Identifier)
 - **Values**
 - Resources para literais

Uma **RDF Statement** é uma **TRIPLE** que contem um:

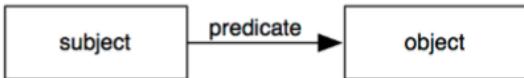
- **Resource**
 - Sujeito da Statement
- **Property**
 - Predicado da statement
- **Value**
 - Objeto da statement

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

<http://www.example.org/index.html> has a **language** which is **English**.

<http://www.example.org/index.html> has a **title** which is **Example_Title**.

8.1 – Exemplo de RDF Statements sobre o Resource <http://www.example.org/index.html>

Graphical form	
Triple	subject predicate object
Relational form	predicate(subject,object)
RDF/XML	<pre><rdf:Description rdf:about="subject"> <ex:predicate> <rdf:Description rdf:about="object"/> </ex:predicate> </rdf:Description></pre>
Turtle	subject ex:predicate object.

8.2 – RDF Triples

```
<?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>
```

8.3 – Exemplo de um RDF File

IX. Turtle Notation

Content Type: text/turtle

File Extension: *.ttl

Formato de texto compacto com varias abreviações utilizado para padrões de uso comuns

```
@prefix i: <http://db.com/terms#>
@prefix m: <http://db.com/movies/>
@prefix a: <http://db.com/actors/>
m:Marnoto
i:actor a:Dalmeida , a.Jakim ;
i:year "2007" ;
i:director [ i:firstname "Paulo" ; i:lastname "Castanho" ]
```

9.1 – Exemplo do conteudo de um Turtle Notation File

X. Approaches a serialização

RDF/XML notation

- XML syntax for RDF (.rdf, .rdfs, .owl, .xml)
- <https://www.w3.org/TR/rdf-syntax-grammar/>

10.1 – Tipos de
serialização
para cada
notação
apresentada
até agora

Turtle notation (Terse RDF Triple Language)

- .ttl extension
- <https://www.w3.org/TR/turtle/>

N-Triples notation

- .nt extension
- <https://www.w3.org/TR/n-triples/>

JSON-LD notation

- JSON-based serialization for Linked Data
- <https://www.w3.org/TR/json-ld/>

XI. Protocol Buffers

File Extension: *.proto

Biblioteca de encoding binario que requere um Schema para qualquer dado que for encoded

Mecanismo extensivel para serialização de dados estruturados – Usado em protocolos de comunicação, data storage, ...

Design Goals incluem

- Language Neutral
- Platform Neutral
- Small, fast, simple

```
message Person {  
    required string user_name = 1;  
    optional int64 favorite_number = 2;  
    repeated string interests = 3;  
}
```

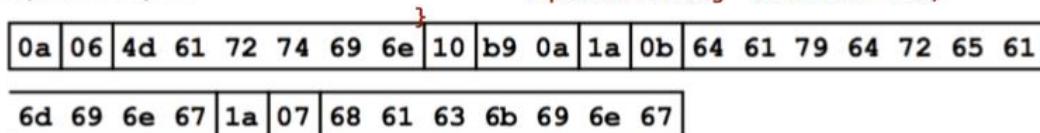
```
syntax = "proto3";  
message Actor {  
    string firstname = 1;  
    string lastname = 2;  
}  
message Movie {  
    string title = 1;  
    int32 year = 16;  
    repeated Actor actors = 17;  
    enum Genre {  
        UNKNOWN = 0;  
        COMEDY = 1;  
    }  
    repeated Genre genres = 2048;  
}
```

11.1 – Exemplos de Protocol Buffer Schemas

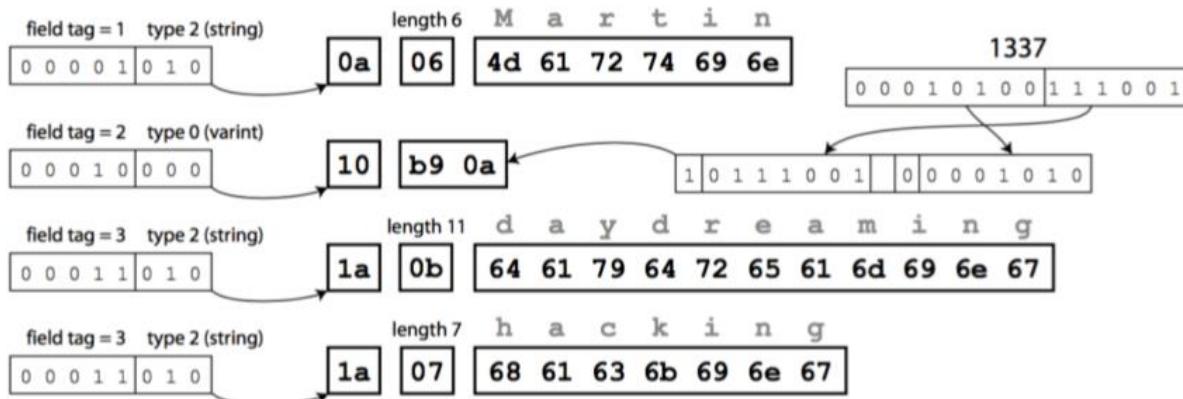
Protocol Buffers

```
message Person {  
    required string user_name = 1;  
    optional int64 favorite_number = 2;  
    repeated string interests = 3;  
}
```

Byte sequence (33 bytes):



Breakdown:



11.2 – Encoding Examples

Os protocol buffers foram criados pela Google com a intenção de serem usados em **Schema Creation -> Automatic source code generation -> Envio de mensagens entre aplicações**

Em termos de componentes, protocol buffers possuem:

- Interface description language
- Source Code Generator (protoc compiler)
- Supported languages
- Binary Serialization format
- Compact, not self describing

Protocol Buffers tratam de schema changes enquanto mantem backwards e forwards compatibility

Um encoded record é apenas a concatenação dos seus encoded fields

Cada field é identificado pelo seu **tag number** e é annotated com um datatype (string, integer, ...)

Se um field value não estiver set, é simplesmente omitido do encoded record

Podemos adicionar novos fields ao schema (Ao le-los, código antigo pode simplesmente ignorar esses fields, mantendo a forward compatibility)

Para garantir a backward compatibility não podemos adicionar required fields:

- Old code não pode escrever novos fields
- Para manter backward compatibility ada new field do schema tem de ser opcional ou ter um valor default

Remover um field funciona tal como adicionar, sendo que desta vez as preocupações de backward e forward compatibility estão trocadas

- So podemos remover optional fields
- Nao podemos voltar a usar o mesmo tag number

Mais concretamente, um **Field** descreve um data value

- **Rule**
 - Numero permitido de value occurrences
 - Default = 0 ou 1
- **Name**
 - Nome dado a um given field
- **Type**
 - **Atomic**: int32, int64, double, bool, bytes, ...
 - Mappings para data types de uma particular linguagem de programação bem como valores default são introduced
 - **Composed**: messages, enumerations, ...
- **Tag**
 - Internal Integer Identifier
 - Usado para identificar fields dentro de uma mensagem num formato binario
 - Frequentemente fields usados são dados tags mais baixas (visto que lower numbers de bytes são necessarios)

XII. Conceitos aprendidos

Programming-language-specific

- restricted to a single programming language.

Textual formats

- widespread, and its compatibility depends on the use.
- Somewhat vague about datatypes, namely numbers and binary strings.

Binary schema-driven formats

- More compact and efficient encoding, with clearly defined forward and backward compatibility semantics.
- The schema can be useful for documentation and code generation in statically typed languages.

12.1 – Conceitos aprendidos neste capítulo

Key-Value Databases

I. O que são – Explicação e Considerações

Key Value Stores são o tipo de NoSQL DBs mais simples que existem

Consistem de uma **unique key** e um **bucket** que contém todos os dados que lá queremos guardar

Os pares Key-Values consistem de:

- **Key**
 - Normalmente uma String
 - Também chamada de Id, Identifier ou Primary Key
- **Value**
 - Pode ser quase tudo – Text, structure, image, etc...

Os Key-Values são **Row based Systems designed para eficiencia**

1.1 – Representação dos Dados numa Key-Value DB

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

O conteúdo do bucket pode ser literalmente tudo, mas o mais comum é serem unstructured ou semi-structured data

Os buckets podem guardar entidades grandes, incluindo BLOBs (Basic Large Objects)

VANTAGENS:

- Altamente Fault Tolerante
 - Sempre available
- Schema-less oferece upgrades path mais fáceis para mudar os requisitos dos dados
 - Document stores fornecem ainda maior flexibilidade
- Eficiente a ir buscar informação sobre um objeto (**bucket**) particular com poucas disc operations
- Modelo de dados muito simples, rápido de fazer set up e de dar deploy
- Ótimo para escalabilidade horizontal (por várias centenas e milhares de servidores)
- Não necessita de SQL Queries, Triggers, indexes, stored procedures, temporary tables, forms, views ou outros overheads técnicos dos RDBMS
- Muito rápido a ingerir dados
 - Favorece write-once, read-many apps

- Offline reporting poderoso com data sets muito largos
- Alguns vendedores estão agora a oferecer formas avançadas de KeValues que possuem capacidades parecidas com a dos document ou column oriented stores

DESVANTAGENS:

- Não é ótimo para aplicações complexas
- Não é eficiente a atualizar records quando apenas uma porção do bucket tem de ser updated
- Não é eficiente a ir buscar limited data de records específicos
 - P.ex ir buscar a uma employee database apenas os records dos empregados que fazem entre §69K e §420K
- À medida que o volume de dados aumenta, torna-se difícil manter unique values como keys
 - É necessário maior complexidade na geração de character strings que se mantenham únicos mesmo para largos sets de chaves
- Geralmente precisamos de ler todos os records num bucket ou então somos obrigados a construir secondary indexes

USE CASES:

- Session data, user profiles, user preferences, shopping carts, ...
- Criar ever-growing datasets que raramente são acedidos mas crescem overtime (caching)
- **Quando o performance dos writes são a nossa maior prioridade**

NOT USE CASES:

- Precisamos de representar relações entre entidades
- Queries requerem acesso a conteúdo através da value part
- Set Operations que envolvam múltiplos key-value pairs



1.2 – Exemplos de KeyValue DBs

II. Key Management

Como é que as Keys devem ser designed e geradas?

Atribuição Manual de Keys

- Real-world natural identifiers
- E.g endereço de email, nomes de login, ...

Geração Automatica de Keys

- Auto-Increment Integers
 - Não é adequado a P2P Architectures though
- Para chaves mais complexas, necessarias em datasets maiores, utilizamos algoritmos
 - Keys compostas por multiplos componentes como timestamps, cluster node ids, etc
 - Muito usado na pratica

III. Query Patterns

Suporta **operações CRUD** Básicas (Create, Read, Update, Delete), com as seguintes considerações:

- Apenas uma chave deve ser fornecida

- O conhecimento das chaves é essencial
- Pode ser difícil para um particular DB System fornecer uma lista com todas as available keys

Não existe pesquisa por valor!

- Podemos, porém, instruir uma DB a fazer parse de valores para podermos ir buscar o intended search criteria e guardar references dentro de index structures

Podemos fazer Batch/Sequential Processing

- Fixe para fazer MapReduce

IV. Outras funcionalidades

- **Expiração dos Key-Value Pairs**
 - Após um certo intervalo de tempo, key-value pairs são automaticamente removidos da BD
 - Util para use cases como user sessions, shopping carts, etc
- **Collections de valores**
 - Podemos guardar não apenas valores ordinários mas também coleções (como ordered lists, unordered sets, etc)

- **Links entre key-value pairs**
 - Valores podem estar mutuamente interligados por links
 - Estes links podem ser traversed ao realizarmos uma query

Outras possíveis funcionalidades dependem do store utilizado

V. Rias Key-Value Store

V.I O que é

Exemplo de uma KVDB

Open source, escalabilidade incremental, alta availability, simplicidade operacional, design decentralizado, distribuição automática de dados, replicação avançada e fault tolerance

Inclui uma general purpose, concurrente, garbage-collected programming language e runtime system



5.1 – Logotipo da Rias Key Value Store

V.II Modelo de dados

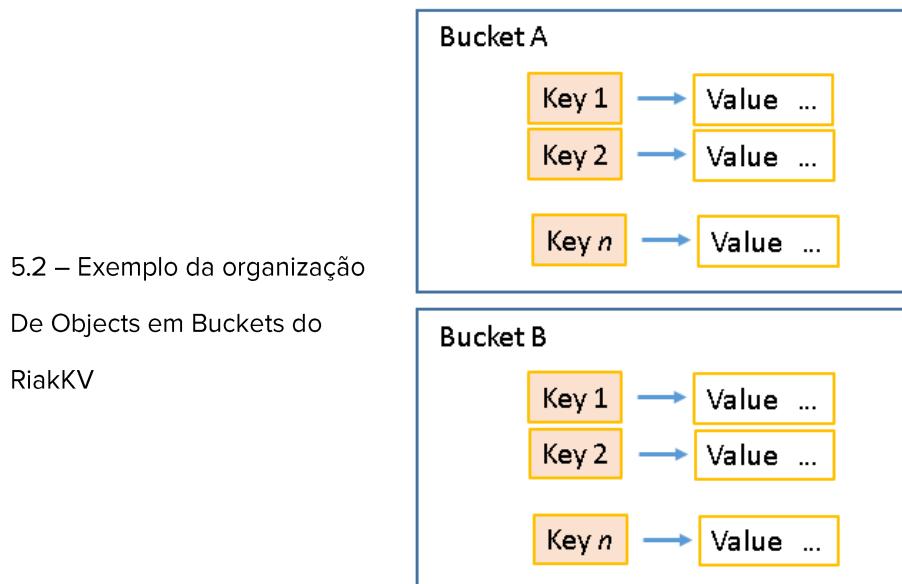
Instance (> Bucket Types) > **Buckets** > **Objects**

Bucket:

- Coleção de lógica (e não física) de objetos
- Cada objeto deve ter uma unique key
- Varias propriedades são set ao nível dos buckets
 - E.g Default replication factor, read/write quora, ...

Object:

- Key-Value Pair
- A Key é um Unicode String
- Values podem ser qualquer coisa
 - Text, Binary Object, Image, ...
- Cada objeto está associado a metadados
 - E.g Content Type, ...



Como é que buckets, keys e values devem estar desenhados? Well tres opções são:

Objetos complexos que contenham vários tipos de dados

- E.g um key-value pair a segurar informação sobre todos os atores e filmes ao mesmo tempo (movies -> wall_street, valerian, spiderverse, ... ; actors -> DS, Pedro, Escaleira, ...)

Buckets com diferentes tipos de objetos

- E.g objetos distintos para atores e filmes, mas todos no mesmo bucket
- Convenções de nomenclatura estruturadas para chaves pode ajudar
- E.g actor_trojan, movie_medvidek

Buckets separados para diferentes tipos de objetos

- E.g um bucket para atores e um para filmes

V.III Operações CRUD

Temos operações CRUD Basicas:

Create: POST or PUT methods

- Inserts a key-value pair into a given bucket
- Key is specified manually, or will be generated automatically

Read: GET method

- Retrieves a key-value pair from a given bucket

Update: PUT method

- Updates a key-value pair in a given bucket

Delete: DELETE method

- Removes a key-value pair from a given bucket

5.3 – Operações CRUD no Riak kv

Bem como outras funcionalidades:

- **Links**
 - Relações entre objetos e a sua transversão
- **Search 2.0 (!!!!)**
 - **Full-text queries que acedem a values dos objetos**
- **MapReduce**

V.IV Utilizações – API

- **HTTP API**
 - Todos os user requests são submetidos como HTTP Requests com o método propriamente selecionado e com o URL, Headers e Dados especificamente construidos
 - A cURL tool permite a transferencia de dados de e para um server usando HTTP
- **Protocol Bufers API**
- **Erlang API**
- **Client libraries para várias linguagens de programação**

VI. Redis – Remote Dictionary Service

V.I O que é

É uma **in-memory** key-value store

Open source, master-slave replication architecture, sharding, alta availability, various persistence levels, ...



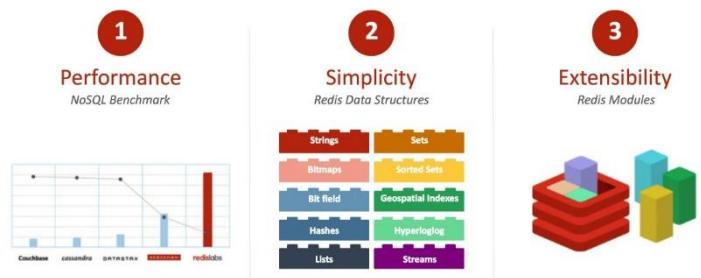
5.1 – Logotipo do Redis

Em termos de funcionalidade oferece:

- Standard Key-Value Store
- Suporte para Structured Values (listas, sets, ...)
- Time-to-live
- Transactions

O Redis não é apenas um Key-Value Store, mas sim um **Data Structures Server** que suporta varios tipos de values

Redis Top Differentiators



V.II Modelo de dados

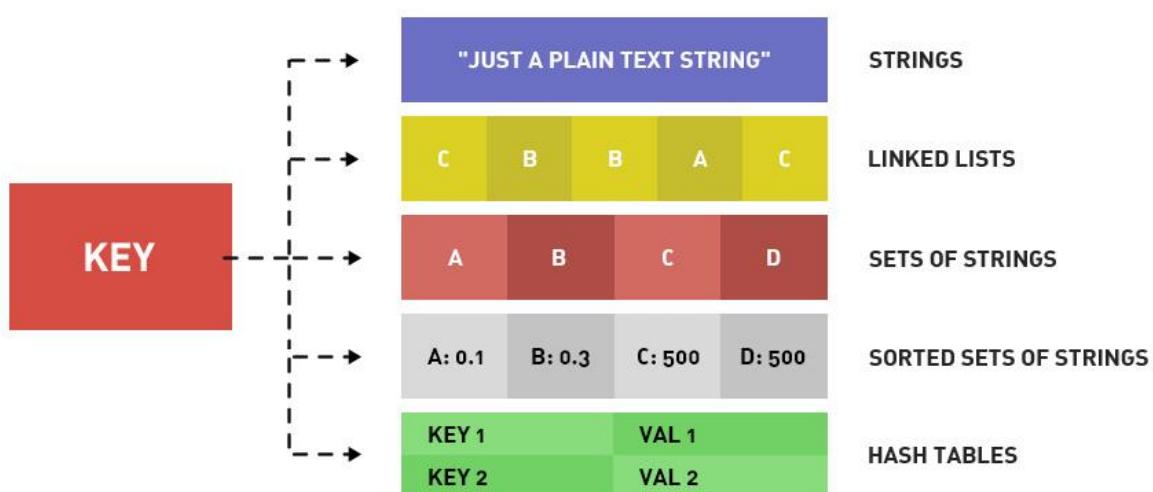
Instance > Databases > Objects

Database:

- Coleção de objetos
- Não tem nomes mas sim Integer Identifiers

Object:

- Key-Value Pair
- A Key (i.e qualquer binary data)
- Values podem ser
 - Atómicos
 - String
 - Structured
 - List, set, ordered set, hash



V.III Data Types



5.4 – Redis Data Types

V.III.I String

- Único data type **Atomico**
- Pode conter qualquer binary data (string, integer counter, PNG Image, ...)
- Maximo allowed size é de 512 MB

V.III.II List

- Coleção ordenada de Strings
- Elementos devem ser preferencialmente lidos ou escritos à cabeça/cauda

V.III.III Set

- Coleção não ordenada de Strings
- Valores duplicados não são permitidos

V.III.IV Sorted Set

- Coleção ordenada de Strings
- A ordem é dada por um score (floating number value) associado a cada elemento (do com menos para o com maior score)
- Valores são garantidamente únicos (não ha valores duplicados)

V.III.V Hash

- Mapa associativo entre String fields e String Values
- Field Names têm de ser mutuamente distintos

V.IV Interface

Temos um **Command Line Client – redis-cli**

Este possui dois modos...

- ...**Basic**

- Comandos são passados como standard command line arguments
 - E.g redis-cli PING
- Permite Batch Processing
 - E.g cat script.txt | redis-cli

- ...**Interativos**

- User escreve DataBase commands na prompt shell da redis-cli

RESP (REdis Serialization Protocol)

V.V Comandos Básicos

SET key value

- inserts / replaces a given string

GET key

- returns a given string

HELP command

- Provides basic information about Redis commands

CLEAR

- Clears the terminal screen

FLUSHDB

- Deletes all the keys of the currently selected database

BGSAVE

- Saves the current dataset (on background)

V.VI String Operations

STRLEN key

- returns a string length

APPEND key value

- appends a value at the end of a string

GETRANGE key start end

- returns a substring Both the boundaries are considered to be inclusive
- Positions start at 0;
- Negative offsets for positions starting at the end

SETRANGE key offset value

- replaces a substring
- Binary 0 are padded when the original string is not long enough

5.6 - Redis String Commands

V.VII Counter Operations

INCR key

DECR key

- Increments / decrements a value by 1

INCRBY key increment

DECRBY key increment

- Increments / decrements a value by a given amount

5.7 - Redis Counter Commands

V.VIII Handling Keys

EXISTS key

- determines whether a key exists

KEYS pattern

- finds all the keys matching a pattern (*, ?, ...)
- E.g. KEYS *

DEL key ...

- removes a given object / objects

RENAME key newkey

- changes the key of a given object

TYPE key – determines the type of a given object

- Types: integer, string, list, set, zset and hash

5.8 - Redis Key Handling Commands

V.IX Volatile Keys

Chaves que possuem um limited Time-To-Live

Quando esse tempo passa, o objeto é removido

Funciona com qualquer data type

EXPIRE key seconds

- Sets a timeout for a given object, i.e. makes the object volatile
- Can be called repeatedly to change the timeout

TTL key

- Returns the remaining time to live for a key

PERSIST key

- Removes the existing timeout

5.9 - Redis Volatile Key Commands

V.X DataTypes Complexos

Uma das razões para a popularidade do Redis vem do facto de suportar valores complexos – Listas, Hashes, Sets e Sorted Sets

Estes podem conter 2^{32} elementos por chave
(jesus that's a lot)

Comandos seguem sempre o mesmo padrão:

- Set commands começam por S
- Hash commands começam por H
- Sorted Set commands começam por Z
- List commands começam ou por um L (Left) ou R (right) dependendo da direção da operação

V.XI List Commands

LPUSH key value

R PUSH key value

- Adds a new element to the head / tail (Left / Right)

LINSERT key BEFORE | AFTER pivot value

- Inserts an element before / after another one

LPOP key

RPOP key

- Removes and returns the first / last element (Left / Right)

V.XII Set Commands

SADD key value ...

- Adds an element / elements into a set

SREM key value ...

- Removes an element / elements from a set

SISMEMBER key value

- Determines whether a set contains a given element

SMEMBERS key

- gets all the elements of a set

SCARD key

- gets the number of elements in a set

SUNION / SINTER / SDIFF key ...

- Calculates and returns a set union / intersection / difference of two or more sets

5.11 - Redis Set Commands

V.XIII Hashes Commands

HSET key field value

- sets the value of a hash field

HGET key field

- gets the value of a hash field

5.12 - Redis Hash Commands

Existem também as alternativas Batch:

HMSET key field value

- Sets values of multiple fields of a given hash

HMGET key field ...

- Gets values of multiple fields of a given hash

5.13 - Redis Hash Batch Commands

HEXISTS key field

- determines whether a given field exists

HGETALL key

- gets all the fields and values

HKEYS key

- gets all the fields in a given hash

HVALS key

- gets all the values in a given hash

HDEL key field

- Removes a given field / fields from a hash

HLEN key

- returns the number of fields in a given hash

5.12 – More Redis Hash Commands

V.XIV Sorted Set Commands

ZADD key score value

- Inserts one element / multiple elements into a sorted set

ZREM key value ...

- Removes one element / multiple elements from sorted set

ZSCORE key value

- Gets the score associated with a given element

ZINCRBY key increment value

- Increments the score of a given element

ZRANGE key start stop

- Returns all the elements within a given range based on positions

ZRANGEBYSCORE key min max

- Returns the elements within a given range based on scores

ZCARD key

- Gets the overall number of all elements

ZCOUNT key min max

- Counts elements within a given range based on score

V.XV Geospatial Field Operations

GEOADD key longitude latitude member ...

- Adds the specified geospatial items (latitude, longitude, name) to the specified key.

GEODIST key member1 member2 ...

- Return the distance between two members.

GEOHASH key member ...

- Return Geohash string (compatible with geohash.org)

GEOPOS key member ...

- Return the positions (longitude,latitude) of all the specified members.

GEORADIUS key longitude latitude radius ...

- Return the members which are within the radius of the location.

5.14 - Redis Geospatial Field Commands

Document Databases

I. O que são – Explicação e Considerações

Tipo de NoSQL DBs nos quais utilizamos documentos para guardar os nossos dados

Em termos de Data Model temos:

- **Documents**
 - Auto Descritivos
 - Organizados de forma hierarquica em estruturas em arvore (JSON, XML, ...)
 - Os valores são escalares, maps, lists, sets, nested documents, ...
 - Identificadas por u unique identifier
- **Collections**
 - Conjunto de Documentos

1.1 –
Representação
dos Dados numa
Document DB

Document 1	Document 2	Document 3
<pre>{ "id": "1", "name": "John Smith", "isActive": true, "dob": "1964-30-08" }</pre>	<pre>{ "id": "2", "fullName": "Sarah Jones", "isActive": false, "dob": "2002-02-18" }</pre>	<pre>{ "id": "3", "fullName": { "first": "Adam", "last": "Stark" }, "isActive": true, "dob": "2015-04-19" }</pre>

Basicamente são quase como **Extended Key-Value Stores** nos quais a parte do Value pode ser examinada

USE CASES:

- Event logging, management systems de conteudos, blogs, web analytics, aplicações de e-commerce, ...
- Basically tudo o que apresentar documentos estruturados com schema similares

NOT USE CASES:

- Operações que envolvam vários documentos
- Design da estrutura dos documentos que esteja sempre a ser mudada
 - i.e quando o nível de granulidade necessário fira um outbalance das vantagens de realizar as agregações
- Se a aplicação tiver muitas **Many-To-Many Relationships** o document model fica menos appealing (não oferece joins)
 - Podemos desnormalizar a BD ou JOINS podem ser emulados no código da aplicação fazendo varios pedidos a DB...
 - Mas problemas de gerir a desnormalização e JOINS podem ser maiores do que o problema de Object-Relational Mismatch das Relational DB

Query Patterns:

- Create, update ou remove de documentos (CRUD operations)
- Retrive documentos de acordo com queries complexas (com aggregates and shit)



1.2 – Exemplos de Document DBs

II. Document VS Relational Databases

<http://www.linkedin.com/in/williamhgates>



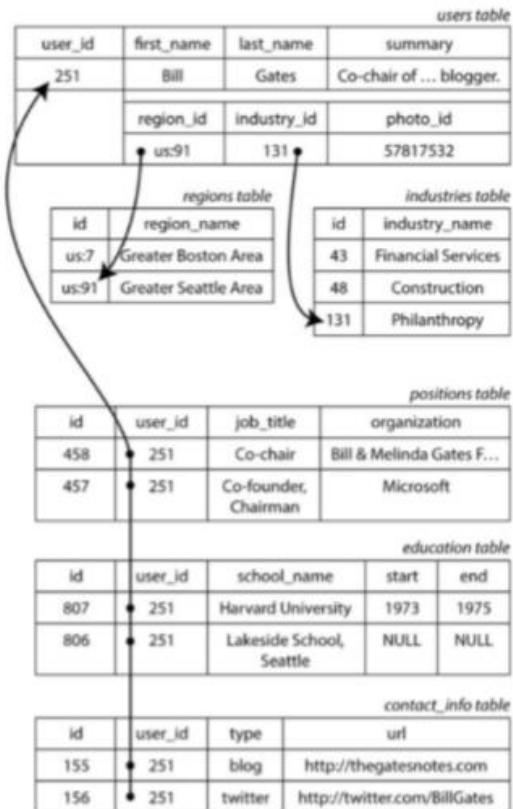
Bill Gates
Greater Seattle Area | Philanthropy

Summary
Co-chair of the Bill & Melinda Gates Foundation. Chairman, Microsoft Corporation. Voracious reader. Avid traveler. Active blogger.

Experience
Co-chair • Bill & Melinda Gates Foundation
2000 – Present
Co-founder, Chairman • Microsoft
1975 – Present

Education
Harvard University
1973 – 1975
Lakeside School, Seattle

Contact Info
Blog: thegatesnotes.com
Twitter: @BillGates



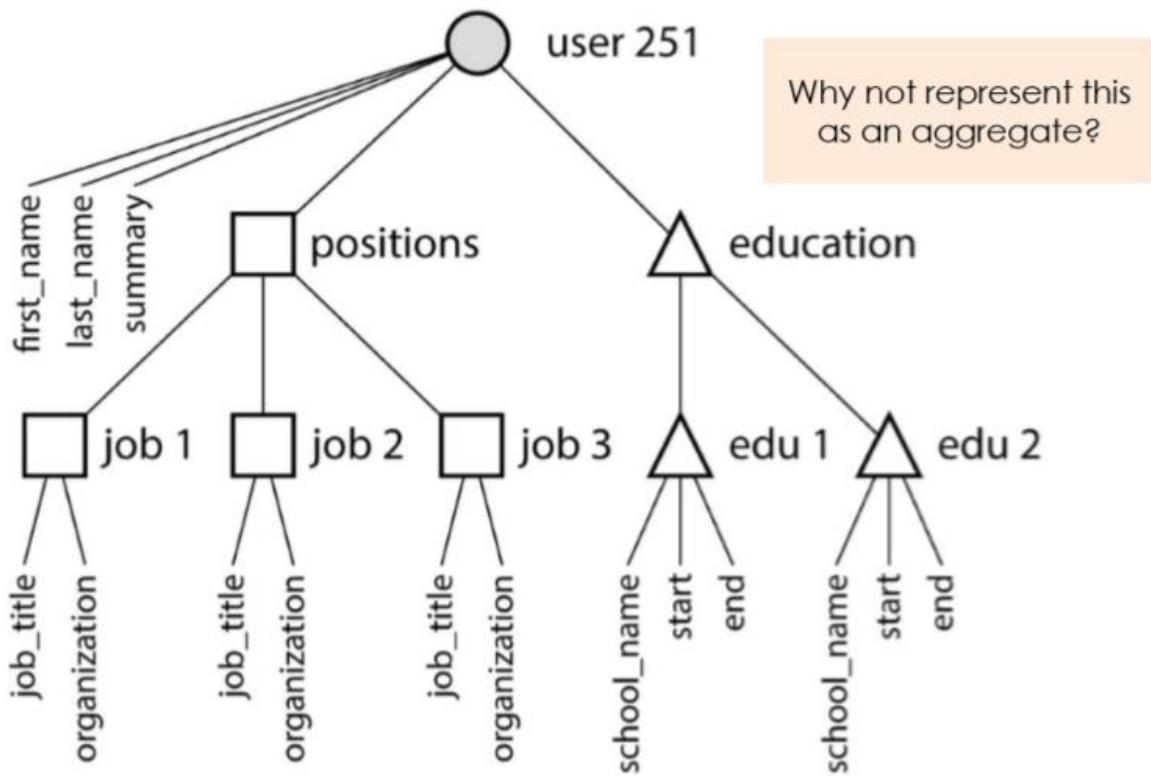
```
{
  "user_id": 251,
  "first_name": "Bill",
  "last_name": "Gates",
  "summary": "Co-chair of the Bill & Melinda Gates... Active blogger.",
  "region_id": "us:91",
  "photo_url": "/p/7/000/253/05b/308dd6e.jpg",
  "positions": [
    {"job_title": "Co-chair", "organization": "B&M Gates Foundation"},
    {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}
  ],
  "education": [
    {"school_name": "Harvard University", "start": 1973, "end": 1975},
    {"school_name": "Lakeside School, Seattle", "start": null, "end": null}
  ]
}
```

2.1 – Exemplo de Relational VS Document DB

O maior argumento de quem suporta Document Data Models é o facto de:

- **Simplificar o application code**
- **O schema é mais flexivel**
- **O performance é melhor devido a locality**

Pode (e deve) ser usada para dados com uma document-like structure (i.e uma tree de one-to-many relationships onde tipicamente toda a tree é carregada de uma vez)



2.2 – Exemplo de uma relação One to Many – Favorece Document Databases

Quando fazemos splitting de uma document-like structure para multiplas tabelas (i.e Relational Database)

estamos a complicar desnecessariamente o application code

III. Document para Schema Flexibility

O facto das Document Based DBs não terem Schema é vantajosa se os **dados forem heterogeneos** (ou seja, se os items numa coleção não tiverem todos a mesma estrutura)

Isto porque, há casos em que o Schema pode ser prejudicial, como por exemplo:

- Há muitos tipos diferentes de objetos e não é pratico meter cada tipo de objeto na sua propria tabela
- Data Structure é determinada por sistemas externos, sobre os quais não temos controlo, e que pode mudar a qualquer altura

O Document é, normalmente, guardado como um **único string continuo**, encoded como JSON, XML ou uma variavel binária (como BSON, no caso do MongoDB)

Se for necessario aceder a todo o documento, há uma vantagem de performance devido a esta **Storage Locality**

A Locality Advantage só se aplica se for precisas largas partes de um documento ao mesmo tempo

A BD normalmente precisa de carregar todo o documento, mesmo que apenas precisemos de aceder a uma pequena porção deste... o que pode ser wasteful

Em Updates a Documents, todo o Document precisa de ser re-written...also not that good

Por isso mesmo é recomendável que se mantenham os documentos pequenos

IV. Convergence de Document e Relational Databases

A maioria dos Relational Database Systems suportam XML

Relational databases têm vindo a adicionar JSON Type support e functions para criar/atualizar XML Documents, permitindo a aplicações usarem os data models de forma muito parecida à dos Document DBs

Por outro lado, as Document Databases têm vindo a adotar práticas Relational como por exemplo o facto de

haverem MongoDB drivers que realizem Client-Side Joins

V. MongoDB Document Database

V.I O que é

É uma Open Source JSON Document Database que fornece High Availability, Eventual Consistency, Automatic Sharding, Master-Slave Replication, Automatic Failover, Secondary Indexing, ...



5.1 – Logotipo da MongoDB Database

V.II Modelo de dados

Instance > Databases > Collections > Documents

Database:

- Set de Collections

Collection:

- Set de Documents, normalmente com uma estrutura parecida (mas não necessariamente igual)

Document:

- Um MongoDB Document corresponde a um JSON Object
- Internamente guardado como um BSON
- Cada documento pertence a exatamente uma coleção
- Cada documento tem um unique identifier **_id**

❖ Collection redwine

```
{  
  _id: "1",  
  name: "Cartuxa",  
  year: 2012  
}  
  
{  
  _id: "2",  
  name: "Evel",  
  year: 2010  
}  
  
{  
  _id: "3",  
  name: "EA",  
  year: 2016  
}
```

❖ Query statement

Wines older than 2012 and later,
sorted by these titles in descending
order

```
db.redwine.find(  
  { year: { $lt: 2014 } },  
  { _id: false, name: true } )  
.sort({ name: -1 })
```

❖ Query result

```
{ "name" : "Evel" }  
{ "name" : "Cartuxa" }
```

5.2 – Exemplo de uma collection com 3 documentos e uma Find Query

V.III Modelo de dados – Primary Key

_id:

- Nome reservado para a Primary Key
- Unique dentro de uma collection
- Imutável (não pode ser mudado depois de assigned)
- Pode ser de um qualquer type exceto arrays
- Valores Possíveis incluem
 - Natural Identifier (Unique String)
 - UUID (Universlaly Unique Identifier)
 - **ObjectID** (default e mais comum)
 - 12-Byte BSON Type
 - Pequeno
 - Provavelmente unico
 - Rapido a gerar
 - Ordenado
 - Baseado no timestamp, id da maquina, id do processo e process-local counter

V.IV Modelo de dados – Denormalizado

Embedded Documents:

- “Subdocumento dentro de um documento”

- Util para One-to-one ou One-to-many relationships
- Traz aabilidade de ler/escrever dados relacionados numa unica operação
 - Melhor performance
 - Menos queries necessarias

```
> db.redwine.insert( {  
    winepack: "Dinner",  
    bottles: [  
        { name: "Cartuxa", year: 2012 },  
        { name: "Evel", year: 2010 },  
        { name: "EA", year: 2016 }  
    ]  
})
```

5.3 – Exemplo da inserção de um documento com 3 subdocumentos

V.V Modelo de dados – Normalizado

References:

- Links direcionados entre documentos
- Expressos via Identifiers
- Parecido com as Foreign Keys das Relational DBs
- Suitable para Many-To-Many Relationships
- Embedding neste caso pode resultar de data duplication

- Fornecem maior flexibilidade do que embedding
- Mas não tem as vantagens mencionadas no embedded document
- Follow up queries são necessárias

```
> db.redwine.insert( {
  winepack: "Dinner",
  bottles: [
    { "$id" : "1" },
    { "$id" : "2" }
  ]
})
```

5.3 – Exemplo da inserção de um documento com 2 References a outros documentos

V.VI Ferramentas

mongod

- the main application

```
$ mongod --dbpath <path to data directory>
```

mongo

- interactive JavaScript interface to MongoDB.

```
$ mongo
```

```
$ mongo --username user --password pass --host host --port 28015
```

Other tools

- bsondump, dump, mongodump
- mongoexport, mongofiles, mongoimport
- mongooplog, mongoperf, mongoreplay
- mongorestore, mongos, mongostat, mongotop

5.4 – Ferramentas incluídas no MongoDB e comandos para as chamar

V.V Query Language

Javascript Commands:

- Cada commando individual é avaliado sobre exatamente uma coleção
- Queries retornam um **cursor** que nos permite iterar sobre os documents selecionados

Query Patterns:

- Operações CRUD Básicas
 - Aceder a documentos via identifiers ou conditions nos fields
- Aggregations
 - MapReduce, Pipelines, Grouping

V.VI CRUD Operations

Create

- db.collection.insertOne()
- db.collection.insertMany()

Read

- db.collection.find()
 - Finds documents based on filtering/projection/sorting conditions

Update

- db.collection.updateOne()
- db.collection.updateMany()

Delete

- db.collection.deleteOne()
- db.collection.deleteMany()

5.5 – Comandos para CRUD Operations

V.VII Selection Operators

Comparison

- **\$eq, \$ne**
 - Tests the actual field value for equality / inequality
- **\$lt, \$lte, \$gte, \$gt**
 - Less than / less than or equal / greater than or equal / greater
- **\$in**
 - Equal to at least one of the provided values
- **\$nin**
 - Negation of \$in

Logical

- **\$and, \$or**
- **\$nor**
 - returns all documents that fail to match both clauses.
- **\$not**

Element operators

- **\$exists**
 - tests whether a given field exists / not exists
- **\$type**
 - selects documents if a field is of the specified type.

Evaluation operators

- **\$regex**
 - tests whether the field value matches a regular expression (PCRE)
- **\$text**
 - performs text search (text index must exists)

Array query operators

- **\$all**
 - Matches arrays that contain all elements specified in the query.
- **\$elemMatch**
 - Selects documents if an element in the array field matches all the specified \$elemMatch conditions.
- **\$size**
 - Selects documents if the array field is a specified size.

```

> db.inventory.find( { status: "D" } )
    // SELECT * FROM inventory WHERE status = "D"

> db.inventory.find( { status: { $in: [ "A", "D" ] } } )
    // SELECT * FROM inventory WHERE status in ("A", "D")

> db.inventory.find( { status: "A", qty: { $lt: 30 } } )
    // SELECT * FROM inventory WHERE status = "A" AND qty < 30

> db.inventory.find( { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )
    // SELECT * FROM inventory WHERE status = "A" OR qty < 30

> db.inventory.find( {
    status: "A",
    $or: [ { qty: { $lt: 30 } }, { item: /%p/ } ]
} )    // SELECT * FROM inventory WHERE status = "A" AND ( qty < 30 OR item LIKE "p%")

```

5.6 – Selection Operators e exemplos de utilização

V.VIII Projection

```

// SELECT _id, item, status FROM inventory
> db.inventory.find( { } , { item: 1, status: 1 } )
{ "_id" : ObjectId("59b1bd23ed835ca4380da8b2"), "item" : "journal", "status" : "A" }
{ "_id" : ObjectId("59b1bd23ed835ca4380da8b3"), "item" : "notebook", "status" : "A" }
{ "_id" : ObjectId("59b1bd23ed835ca4380da8b4"), "item" : "paper", "status" : "D" }
{ "_id" : ObjectId("59b1bd23ed835ca4380da8b5"), "item" : "planner", "status" : "D" }
{ "_id" : ObjectId("59b1bd23ed835ca4380da8b6"), "item" : "postcard", "status" : "A" }

// SELECT item, status FROM inventory
> db.inventory.find( { } , { _id: 0, item: 1, status: 1 } ) // true or 1 is included
{ "item" : "journal", "status" : "A" }
{ "item" : "notebook", "status" : "A" }
{ "item" : "paper", "status" : "D" }
{ "item" : "planner", "status" : "D" }
{ "item" : "postcard", "status" : "A" }

> db.inventory.find( {} , { _id: 0, qty: 0, size: 0 } ) // false or 0 is excluded
{ "item" : "journal", "status" : "A" }
{ "item" : "notebook", "status" : "A" }
{ "item" : "paper", "status" : "D" }
{ "item" : "planner", "status" : "D" }
{ "item" : "postcard", "status" : "A" }

```

5.5 – Exemplos do uso de Projections

V.IX Modifiers – Sort, Limit, Skip

```
// SELECT _id, item, status FROM inventory ORDER BY status ASC
> db.inventory.find( {} , { _id: 0, item: 1, status:1 }).sort({ status: 1 })
{ "item" : "journal", "status" : "A" }
{ "item" : "notebook", "status" : "A" }
{ "item" : "postcard", "status" : "A" }
{ "item" : "paper", "status" : "D" }
{ "item" : "planner", "status" : "D" }

> db.inventory.find( {} , { _id: 0, item: 1, status:1 }).sort({ status: -1 })
{ "item" : "paper", "status" : "D" }
{ "item" : "planner", "status" : "D" }
{ "item" : "journal", "status" : "A" }
{ "item" : "notebook", "status" : "A" }
{ "item" : "postcard", "status" : "A" }
> db.inventory.find( {} , { _id: 0, item: 1, status:1 }).limit(3)
{ "item" : "journal", "status" : "A" }
{ "item" : "notebook", "status" : "A" }
{ "item" : "paper", "status" : "D" }
> db.inventory.find( {} , { _id: 0, item: 1, status:1 }).skip(3)
{ "item" : "planner", "status" : "D" }
{ "item" : "postcard", "status" : "A" }
```

5.6 – Exemplos do uso de Modifiers

V.X Update Operation

Syntax

```
db.collection.updateOne(filter, update, options)
db.collection.updateMany(filter, update, options)

db.collection.updateOne(
    <filter>,    // = selectors in find()
    <update>,    // modification to apply
    {
        // optional ...
        upsert: <boolean>,           // if no doc -> insert
        writeConcern: <document>,   // ack of num of replicas
        collation: <document>       // language-specific rules
    }
)
```

Update operators

```
$set, $unset, $rename
```

5.7 – Sintaxe da CRUD Update Operation

V.X Delete Operation

Syntax

```
db.collection.deleteOne(filter, options)
db.collection.deleteMany(filter, options)

db.collection.deleteOne(
  <filter>,    // = selectors in find()
  {
    // optional ...
    writeConcern: <document>, // ack of num of replicas
    collation: <document>     // language-specific rules
  }
)
```

5.8 – Sintaxe da CRUD Delete Operation

V.VI Indexes

Motivação:

- Precisamos de realizar scans a toda uma coleção para pesquisar por um documento (lento)
- O uso de indexes previne isso

Primary Index:

- O MongoDB cria um unique index com o **_id** field aquando da criação da coleção !!!

Secondary Index:

- Criados manualmente para uma dada key field (ou conjutno de fields)

- Para criar um index usamos o comando
 - db.<collection>.createIndex(keys, options)
- Os MongoDB Indexes usam uma **B-Tree Data Structure** !

V.VII Index Types

Single Field

- Ascending ou Descending Indexes de um único field

Compound Index

- Indexes sobre vários fields
- A ordem dos fields listados num Compound Index tem relevância
- E.g {userid: 1, score: -1} sort por userid de forma ascendente, by score de forma descendente

Multikey Index

- Para indexar um campo que possua um array value

Text Index

Hashed Index

GeoSpatial Index

1, -1 – standard ascending / descending value indexes

```
db.<collection>.createIndex( { field: -1 } )
```

hashed – hash values of a single field are indexed

```
db.<collection>.createIndex( { _id: "hashed" } )
```

text – basic full-text index

```
db.<collection>.createIndex( { comments: "text" } )
```

2d – points in planar geometry

```
db.<collection>.createIndex( { <location field> : "2d" , <additional field> : <value> } , { <index-specification options> } )
```

2dsphere – points in spherical geometry

```
db.<collection>.createIndex( { <location field> : "2dsphere" } )
```

```
// Full collection scan
> db.inventory.find( {qty: { "$gte" : 50 }}).sort( {qty: -1})
{ "item" : "paper", "qty" : 100, "status" : "D" }
{ "item" : "planner", "qty" : 75, "status" : "D" }
{ "item" : "notebook", "qty" : 50, "status" : "A" }

> db.inventory.getIndexes()
[
    {"v": 2, "key": { "_id": 1 }, "name": "_id_", "ns": "test.inventory" }
]

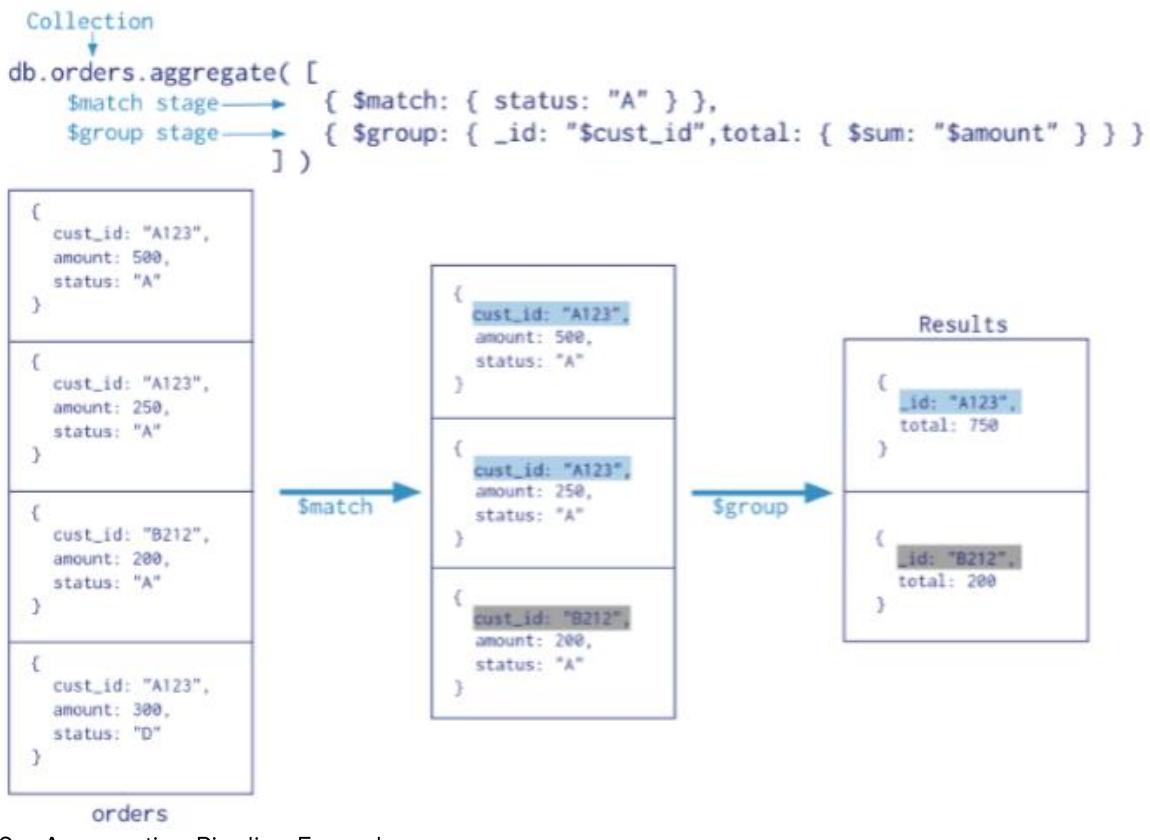
> db.inventory.createIndex( { qty : 1 } )

> db.inventory.getIndexes()
[
    {"v": 2, "key": { "_id": 1 }, "name": "_id_", "ns": "test.inventory" }
    {"v": 2, "key": { "qty": 1 }, "name": "qty_1", "ns": "test.inventory" }
]
```

5.9 – Index Creation Commands e Exemplos

V.VIII Pipeline de Agregação

Os Documents podem entrar numa multi-stage pipeline que transforma os documentos nos resultados da Aggregation



5.10 – Aggregation Pipeline Example

V.IX MapReduce

O paradigma de Data Processing para condensar largos volumes de dados em resultados de agregação úteis

Tanto o Map como o Reduce são funções implementadas ordinariamente com JavaScript

- **Map**

- Função pela qual o document é acessido
- Emit(key, value) usado para emissões

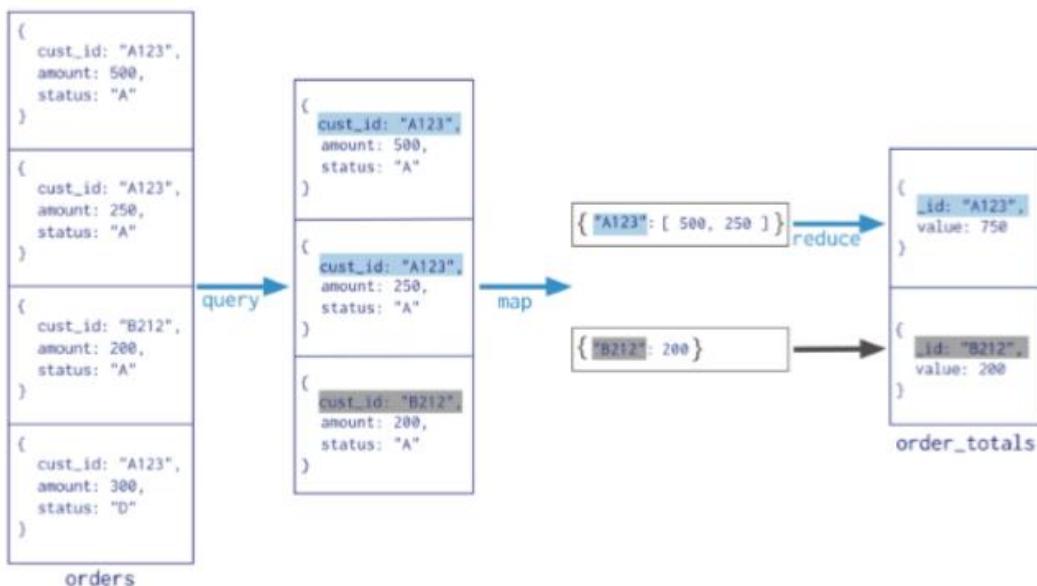
- **Reduce**

- Função na qual uma key e array de valores são fornecidos como argumentos
- O valor reduzido é depois retornado

Para além de outros, é aceite que seja passado uma **query**, **sort** ou **limit** options

A **Out Option** determina o output (i.e o nome da coleção)

```
Collection
↓
db.orders.mapReduce(
  map → function() { emit( this.cust_id, this.amount ); },
  reduce → function(key, values) { return Array.sum( values ) },
  {
    query → { status: "A" },
    output → { "out": "order_totals" }
  }
)
```



5.11 – Exemplo de MapReduce

V.X MongoDB Drivers

Por fim é de referir que o Ecosistema do MongoDB contem documentação para drivers, frameworks, tools e serviços de plataformas que funcionam com MongoDB

```
public class Test {
    public static void main(String[] args) {
        // remove log in the console
        java.util.logging.Logger.getLogger("org.mongodb.driver").setLevel(
            Level.SEVERE);
        MongoClient mongo = new MongoClient("localhost", 27017);
        // os dados foram colocados manualmente no mongo
        MongoDB out = mongo.getDatabase("test");
        System.out.println("-- Coleções na BD " + "" + out.getName() + "");
        MongoIterable<String> x = out.listCollectionNames();
        for (String s : x)
            System.out.println(s);
        MongoCollection<Document> c = out.getCollection("inventory");
        System.out.println("-- Total de documentos em 'inventory': " + c.count());
        FindIterable<Document> docs = c.find();
        for (Document doc : docs)
            System.out.println(doc.toJson());
        mongo.close();
    }
}
```

-- Coleções na BD 'test'
invoice
inventory
collection
orders
--- Total de documentos em 'countries': 5
{ "_id" : { "\$oid" : "59b1b730935c2a0ca72c432c" }, "item" : "journal",
"qty" : 25.0, "status" : "A" }
{ "_id" : { "\$oid" : "59b1b730935c2a0ca72c432d" }, "item" : "notebook",
"qty" : 50.0, "status" : "A" }
{ "_id" : { "\$oid" : "59b1b730935c2a0ca72c432e" }, "item" : "paper", "qty"
: 100.0, "status" : "D" }
{ "_id" : { "\$oid" : "59b1b730935c2a0ca72c432f" }, "item" : "planner",
"qty" : 75.0, "status" : "D" }
{ "_id" : { "\$oid" : "59b1b730935c2a0ca72c4330" }, "item" : "postcard",
"qty" : 45.0, "status" : "A" }

```

public class Test2 {
    public static void main(String[] args) {
        // remove log in the console
        java.util.logging.Logger.getLogger("org.mongodb.driver").setLevel(
            Level.SEVERE);
        MongoClient mongo = new MongoClient("localhost", 27017);
        MongoCollection<Document> coll =
            mongo.getDatabase("test").getCollection("inventory");

        Document doc = new Document("item", "database")
            .append("qty", 1)
            .append("status", "M");
        coll.insertOne(doc);
        FindIterable<Document> docs = coll.find();
        for (Document d : docs)
            System.out.println(d.toJson());
        mongo.close();
    }
}

```

{ "_id" : { "\$oid" : "59b1b730935c2a0ca72c432c" }, "item" : "journal",
"qty" : 25.0, "status" : "A" }
{ "_id" : { "\$oid" : "59b1b730935c2a0ca72c432d" }, "item" : "notebook",
"qty" : 50.0, "status" : "A" }
{ "_id" : { "\$oid" : "59b1b730935c2a0ca72c432e" }, "item" : "paper", "qty"
: 100.0, "status" : "D" }
{ "_id" : { "\$oid" : "59b1b730935c2a0ca72c432f" }, "item" : "planner",
"qty" : 75.0, "status" : "D" }
{ "_id" : { "\$oid" : "59b1b730935c2a0ca72c4330" }, "item" : "postcard",
"qty" : 45.0, "status" : "A" }
{ "_id" : { "\$oid" : "59b2a7e98cbc6f6497c7110" }, "item" : "database",
"qty" : 1, "status" : "M" }

```

public class Test3 {
    public static void main(String[] args) {
        java.util.logging.Logger.getLogger("org.mongodb.driver").setLevel(
            Level.SEVERE);
        MongoClient mongo = new MongoClient("localhost", 27017);
        MongoCollection<Document> coll =
            mongo.getDatabase("test").getCollection("inventory");
        Document doc = new Document("item", "record")
            .append("size",
                new Document("h", 10).append("l", 20).append("w", 30))
            .append("qty", 1)
            .append("status", "R");
        coll.insertOne(doc);
        FindIterable<Document> docs = coll.find(new Document("status", "R"));
        for (Document d : docs)
            System.out.println(d.toJson());
        mongo.close();
    }
}

```

{ "_id" : { "\$oid" : "59b2a9eb8cbc6f6527068b2" }, "item" : "record",
"size" : { "h" : 10, "l" : 20, "w" : 30 }, "qty" : 1, "status" : "R" }

Column Databases

I. O que são – Explicação e Considerações

Column Databases são mais um tipo de NoSQL DBs.

A ideia geral é que vamos guardar e processar dados por coluna em vez de por linha (row)

Tem as suas origens em analytics e business intelligence cujas queries consistem na sua maioria de aggregate queries

Podem ser descritas como um “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

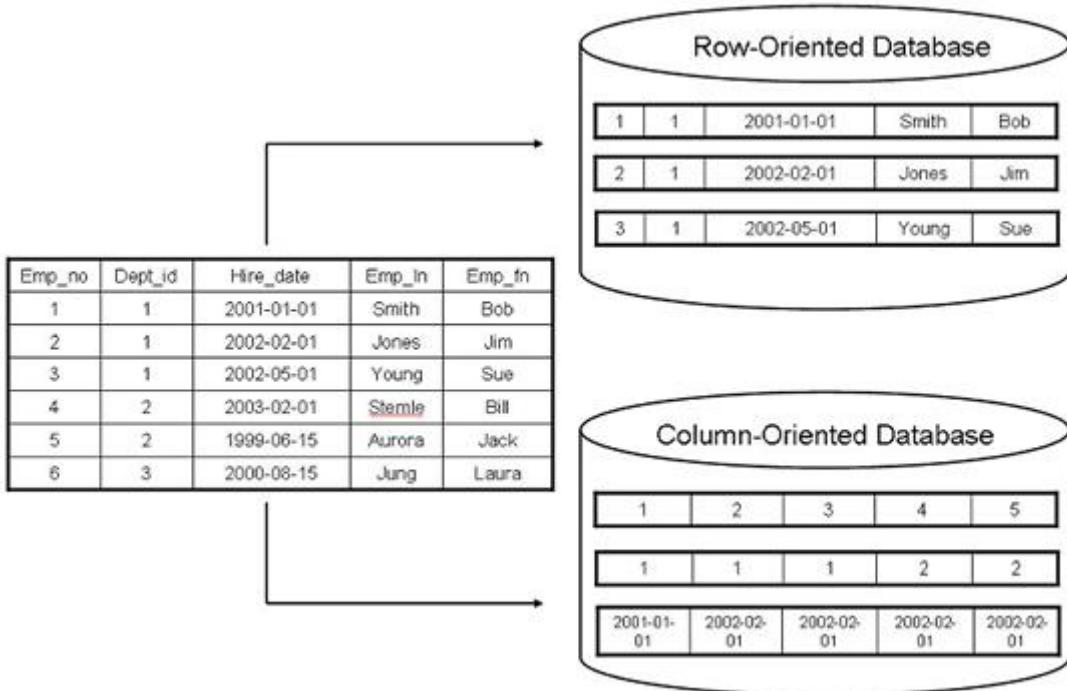
❖ Store by row:

```
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;
```

1.1 – Exemplo de uma Store by row de uma Store by column



1.2 – Outro exemplo de uma Row Oriented vs Column Oriented DB

VANTAGENS:

- Torna algumas queries mesmo muito rápidas
 - Aggregation Queries
 - Funções sobre fields (p.ex average age of users)
- Melhor compressão de dados
 - Ao corer o algoritmo em cada cluna (dados parecidos)
 - Mais notável quando começamos a ter datasets grandes

DESVANTAGENS:

- Aggregation é nice, mas algumas aplicações precisam de ser capazes de mostrar dados para cada individual record

- BDs colunares são geralmente não otimas para esses tipos de queries
- Escrever novos dados pode demorar demasiado tempo
 - Inserir um novo record numa row oriented database é uma simples write operation
 - Fazer update de muitos values numa columnar db pode demorar muito tempo

USE CASES:

- Queries que envolvam poucas colunas
- Aggregation queries contra muitos dados
- Column-Wise Comparaison
- Exemplos incluem Event Logging, Content Management Systems, Blogs, ...
 - Structured flat data with similar schema
- Also groovy for Batch Processing via map reduce

NOT USE CASES:

- Incremental data loading
- **Online Transaction Processing (OLTP) usage**
- Queries contra poucas rows
- Quando precisamos de ACID transactions
- Quando temos queries complexas (JOINS)

- Early Prototypes (quando o design da bd ainda está sujeito a mudança)



1.3 – Exemplos de Column DBs

Em termos de **Data Model** temos:

- **Column Family (Table)**
 - Coleção de Rows parecidas, mas não necessariamente identicas
- **Row**
 - Coleção de Colunas
 - Deve englobar um grupo de dados a ser acedidos juntos
 - Associado a uma **unique row key**
- **Column**
 - Consiste de um **column name** e **column value** (e, possivelmente, de outros metadata records)

- Permit **values** Escalares, Sets, Listas e Mapas

No que toca a **Query Patterns** podemos

- Criar, atualizar ou remover uma row de uma dada column family
- Selecionar rows de acordo com uma row key ou condições simples

WARNING: Por muito que pareça à primeira vista, Wide Column Stores não são só um special kind de RDBMSs com um variable set columns

II. Apache Cassandra

II.I O que é

É uma Cross Platform, open-source Column Database cujas features incluem High Availability, linear scalability, sharding, P2P Configurable Replication, Tunable Consistency e MapReduce Support

2.1 – Logotipo da Cassandra Database



A Cassandra nasceu do problema de performance que o Facebook enfrentou quando correu testes sobre 50TB de User Messages Data numa cluster de 150 nodes em 2 data centers

Search index of all messages in 2 ways:

- term search: search by a key word
- interactions search: search by a user id

Latency	Search Interactions	Search Term
Min	7.69 ms	7.78 ms
Median	15.69 ms	18.27 ms
Max	26.13 ms	44.41 ms

2.2 – Performance dos 2 tipos de searches sobre os dados efetuados

Efetivamente, verifica-se que para grandes quantias de dados, há uma diferença astronomico entre o performance do MySQL e da Cassandra

Average Time	MySQL	Cassandra
Write	~ 300 ms	0.12 ms
Read	~ 350 ms	15 ms

2.3 – Testes de Performance corridos pelo Facebook (FB Use case) para >50Gb de Dados

Em termos de funcionalidades e motivações, Cassandra oferece:

- High Availability
- High Write Throughput
 - Sem sacrificar a eficiencia dos reads
- Fault Tolerance
- High and Incremental Scalability
- Reliability numa escala massiva

II.II Data Center & Cluster

- **Node**
 - Maquina onde a Cassandra está a correr
- **Data Center**
 - Coleção de Related Nodes
 - Sinonimo de Group Replication
 - Replicação é feita por data center
 - Agrupamento de Nodes configurados juntos para objetivos de replicação
 - Utilização de Data Center separados permite-nos
 - Dedicar cada data center a diferentes tasks de processamento
 - Satisfazer pedidos de um data center mais perto do cliente (menos lag)

- Cluster

- Coleção de Data Centers

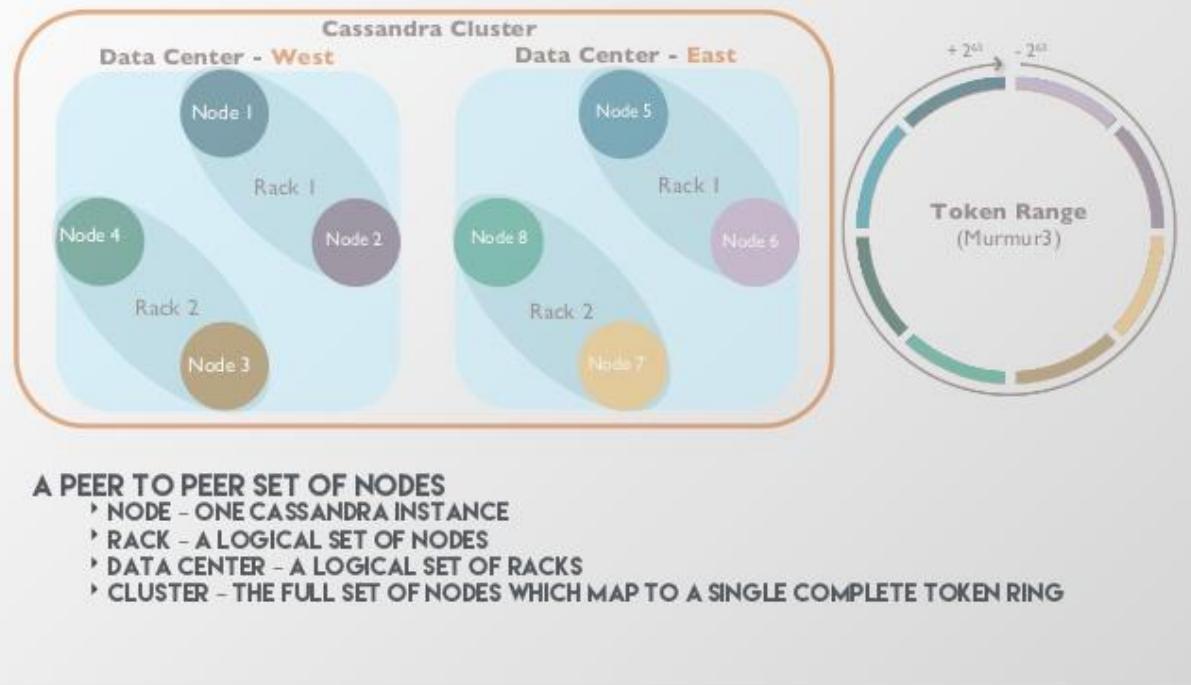
- Os mesmos dados são escritos em todos os Data Centers da Cluster

Cassandra Cluster



2.4 – Dois DataCenters, cada um com 2 dois nodes, pertencentes ao mesmo cluster

WHAT IS A C* CLUSTER?



2.5 – Outro exemplo e explicação da Cassandra Node Cluter

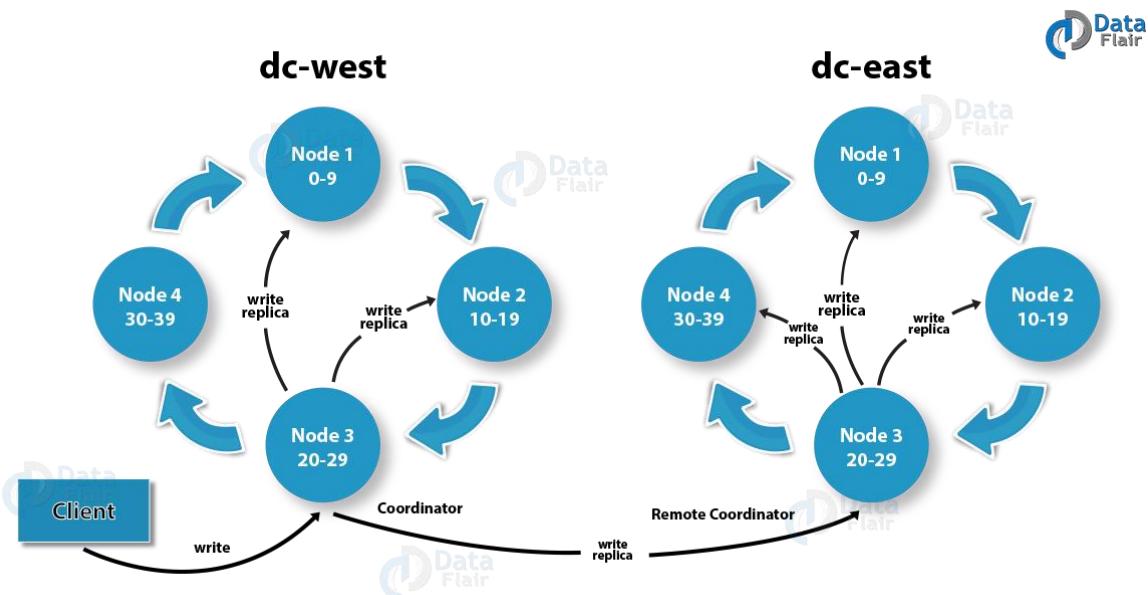
II.III System Architecture

- **Cluster Membership**
 - Como é que os nodes são adicionados ou apagados de uma cluster
- **Partitioning**
 - Como é que dados são particionados pelos nodes
 - Nodes são logicamente estruturados numa Ring Topology
 - Hashed Value de 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 nodes
 - Cada DataItem é replicado por N (replication factor) Nodes

II.IV Network Nodes Topology

- Cluster Data é gerida por um Ring of Nodes
- Cada Node tem parte da BD
- Rows são distribuidas baseadas na primary key
 - Row Lookups são rápidos

- Multiplos nodes tem os mesmos dados, para garantir availability e durability
- Não ha nenhum master node – Todos os nodes podem realizar todas as operações



2.6 – Como é que um write do cliente é propagado para os data centers pela ring topology

II.V Modelo de dados

Keyspaces > Tables > Rows > Column

Keyspace:

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

Table (Column Family):

- Coleção de rows (parecidas)
- Multi-Dimensional Map indexado por chave (row key)
- Table Schema tem de ser especificado mas pode ser mudificado
- 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**

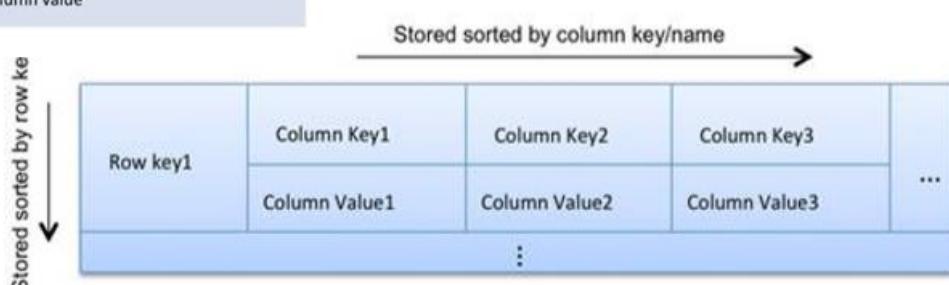
Column:

- Name-Value Pair + Dados Adicionais

Relational Model	Cassandra Model
Database	Keyspace
Table	Column Family (CF)
Primary key	Row key
Column name	Column name/key
Column value	Column value

A nested sorted map is a accurate analogy for each column family:

Map<RowKey, SortedMap<ColumnKey, ColumnValue>>



2.7 – Comparação entre Relational e Cassandra Models ; Analogia da Nested Sorted map para cada column family ;Exemplo de uma Cassandra Row (com várias columns cada uma com key e values)

actors

id					
machacek	trojan	name (Ivan, Trojan)		year 1964	
	sverak	name (Jiří, Macháček)		year 1966	
		movies { medvidek, vratnelahve, samotari }			
	schneiderova	name (Jitka, Schneiderová)		year 1973	
		movies { samotari }			
sverak		name (Zdeněk, Svěrák)		year 1936	
		movies { vratnelahve }			

movies

id					
medvidek	samotari	title Samotáři		year 2000	actors null
	vratnelahve	title Medvídek		director (Jan, Hřebejk)	year 2007
		actors { trojan: Ivan, machacek: Jirka }			
	zelary	title Vratné lahve		year 2006	actors { machacek: Robert Landa }
		title Želary		year 2003	actors { } genres [romance, drama]

2.8 – Exemplo de duas Column Families

blog relational database

users table			blog table		
user_id	username	state	blog_id	user_id	blog_entry
1	jbellis	TX	101	1	Today I ...
2	dhutch	CA	102	2	I am ...
3	egilmore	NULL	103	1	This is ...
subscriber table			category table		
subscriber	blogger	row_id	category	categoryid	
1	2	1	sports	1	
2	1	2	fashion	2	
1	3	3	technology	3	

blog keyspace

users			blog entries		
jbellis	name	state	92dbbeb5	body	user
	jonathan	TX		Today I ...	category
dhutch	name	state	d418a66	body	user
	daria	CA		I am ...	fashion
egilmore	name		6a0b483	body	user
	eric			This is ...	category
subscribes_to			subscribers_of		
jbellis	dhutch	egilmore	jbellis	dhutch	egilmore
dhutch	jbellis		dhutch	egilmore	
egilmore	jbellis	dhutch	egilmore	jbellis	
time_ordered_blogs_by_user			secondary indexes		
jbellis	1289847840615		92dbbeb5		
	92dbbeb5		d418a66		
dhutch	1289847840615				
	d418a66		6a0b483		
egilmore	1289847844275				
	6a0b483				

2.9 – Exemplo de Relational DB vs Cassandra

II.VI Column Values

Os value types aceitos pelas Columns são:

Empty Value:

- Null

Atomic Value:

- Native Data Types
 - Text, Integers, Dates, ...
- Tuples
 - Tuplos 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 em Frozen Mode (elementos são serializados quando são guardados)

II.VII 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 ultima modificação foi feita
- Fornecido automaticamente ou manualmente

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

II.VIII 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, ...

II.IX Cassandra Query Language (CQL)

Declarative query language inspired by SQL

- <https://cassandra.apache.org/doc/latest/cql/>
- <http://docs.datastax.com/en/dse/5.1/cql/>

DDL statements

CREATE KEYSPACE – creates a new keyspace

CREATE TABLE – creates a new table

...

DML statements

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

...

2.11 – Cassandra Query Language Examples

II.X Keyspace

CREATE KEYSPACE

```
CREATE KEYSPACE [ IF NOT EXISTS ] keyspace_name WITH options
```

Replication option is mandatory

- SimpleStrategy
 - one replication factor for the whole cluster
- NetworkTopologyStrategy
 - individual replication factor for each data center

```
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;
```

USE KEYSPACE

```
USE keyspace_name
```

DROP KEYSPACE

```
DROP KEYSPACE [ IF EXISTS ]
```

ALTER KEYSPACE

```
ALTER KEYSPACE keyspace_name WITH options
```

```
ALTER KEYSPACE Excelsior
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 4};
```

2.12 – Comandos para manipulação de Keyspace

II.XI 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 cql_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))*

2.13 – Comandos para Criação de uma Table

```
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)
```

2.14 – Exemplos de Create Tables

II.XII Table Primary Key

A Primary Key tem 2 partes:

Compulsory Partition Key:

- Single Column ou Multiple Columns
- Descreve como é que as table rows são distribuidas pelas partitions

Optional Clustering Key (ou Columns):

- Define a Clustering Order
 - I.e 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.

2.15 – Criação de uma Table com uma Partition Key e Clustering Key

II.XIII Key Roles

Partition Key:

- Responsavel por Data Distribuiton (partitioning) pelos nodes da BD
- Pode ter multiplas colunas

Clustering Key:

- Responsavel por 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)
);
```

2.16 – Criação de uma Table com uma Partition Composite Key e variadas Clustering Keys

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

- ❖ cluster and date fields define the partition (node) where the row is stored
- ❖ Inside the partition, every row 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 ...}}
```

2.17 – Exemplo de como é que a Key são usadas e afetam o store das columns

II.XIV Table – Other Statements

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;
```

2.18 – Comandos para Manipulação de uma Table

II.XV Select Statements

Reads one or more columns for one or more rows in a table

```
SELECT [ JSON | DISTINCT ] ( select_clause | '*' )
FROM table_name
[ WHERE where_clause ]
[ GROUP BY group_by_clause ]
[ ORDER BY ordering_clause ]
[ PER PARTITION LIMIT (integer | bind_marker) ]
[ LIMIT (integer | bind_marker) ]
[ ALLOW FILTERING ]
```

Clauses

- SELECT – columns or values to appear in the result
- FROM – single table to be queried
- WHERE – filtering conditions to be applied on table rows
- GROUP BY – columns used for grouping of rows
- ORDER BY – criteria defining the order of rows in the result
- LIMIT – number of rows to be included in the result

5'

UNIVERSIDADE
AVEIRO

2.19 – Comandos e clauses de Select Statements

II.XVI Select – FROM Clause

Defines a **single table** to be queried

- from the current / specified keyspace
- joining of multiple tables is not possible

Supports:

- **distinct** to remove duplicate rows
- (user-defined) **aggregate functions**
- * to select all columns; and attributes **alias (AS)**
- WRITETIME (**timestamp**) and TTL (**time-to-live**) of a column
 - cannot be used in WHERE clause

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

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

UNIVERSIDADE
AVEIRO

2.20 – From Clause do Cassandra SELECT

52

II.XVII Select – WHERE Clause

Sintaxes parecidas entre CQL e SQL

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

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

Partition Key, Clustering e normal columns suportam diferentes sets de restrições dentro da WHERE clause

Partition Key Columns:

- Suportam = e IN
- Todas as primary key columns tem de ser usadas (restricted), a não ser que tenhamos secondary indexes
- Todas as colunas são precisas para computar a hash que vai permitir localizar os nos que contenham a 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 a query estiver a usar um secondary index
 - Usado em collections* (lists, sets, maps) / maps**

```
CREATE TABLE number0fRequests (
    cluster text,
    date text,
    datacenter text,
    hour int,
    minute int,
    number0fRequests int,
    PRIMARY KEY ((cluster, date), datacenter, hour, minute))
```

```
/* Data will be stored like this:
{datacenter: US_WEST_COAST (hour: 0 {minute: 0 {number0fRequests: 130}} {minute: 1 {number0fRequests: 125}}) ...
 (minute: 59 {number0fRequests: 97})}
 (hour: 1 {minute: 0 ...
 */
```

```
SELECT * FROM number0fRequests
 WHERE cluster = 'cluster1'
 AND datacenter = 'US_WEST_COAST'
 AND hour = 14
 AND minute = 00;
```



```
SELECT * FROM number0fRequests
 WHERE cluster = 'cluster1'
 AND date = '2015-06-05'
 AND datacenter = 'US_WEST_COAST'
 AND hour = 14
 AND minute = 00;
```



```
SELECT * FROM number0fRequests
 WHERE cluster = 'cluster1'
 AND date = '2015-06-05'
 AND hour = 14
 AND minute = 00;
```



```
SELECT * FROM number0fRequests
 WHERE cluster = 'cluster1'
 AND date = '2015-06-05'
 AND datacenter = 'US_WEST_COAST'
 AND hour IN (14, 15)
 AND minute = 0;
```

```
SELECT * FROM number0fRequests
 WHERE cluster = 'cluster1'
 AND date = '2015-06-05'
 AND datacenter = 'US_WEST_COAST'
 AND (hour, minute) IN ((14, 0), (15, 0));
```

```
-- multi-column IN restrictions can be applied to any set of clustering columns.
SELECT * FROM number0fRequests
 WHERE cluster = 'cluster1'
 AND date = '2015-06-05'
 AND (datacenter, hour) 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)

```

2.21 – WHERE Clause examples

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'; X
```



```
/*
   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'; V
```

II.XVIII Select – GROUP BY, ORDER BY & LIMIT

GROUP BY clause:

- Agrupar Rows de uma Table de acordo com certas colunas
- Apenas groupings unduzidos por primary key columns são permitidos

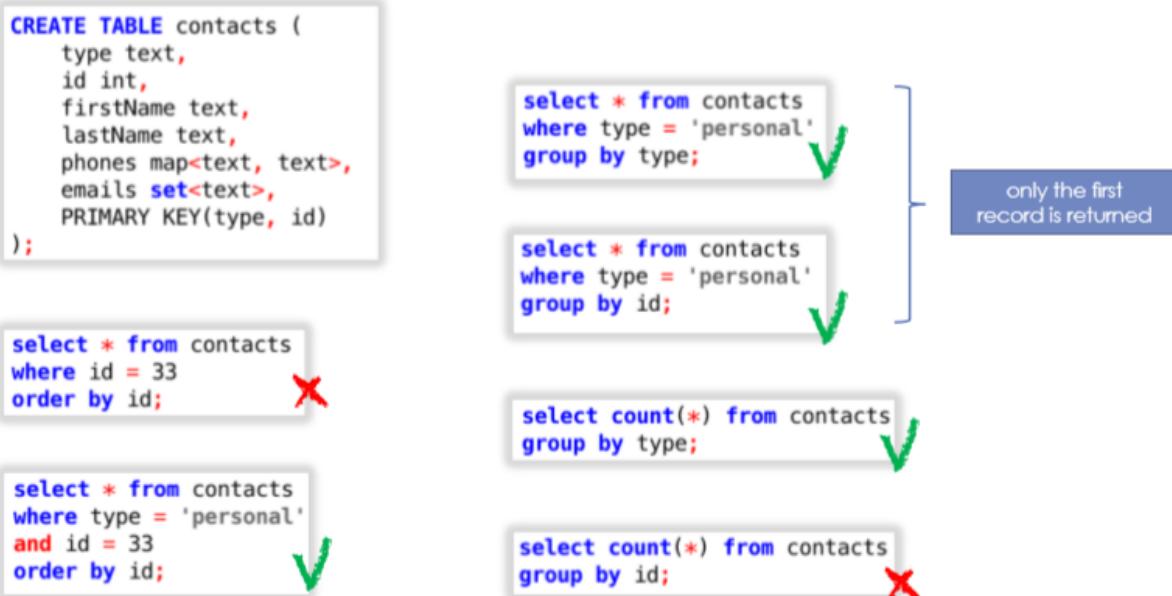
- Quando non-grouping columns são selecionadas sem uma aggregate function, o primeiro value encontrado é sempre retornado

ORDER BY clause:

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

LIMIT clause:

- Limita o numero de rows retornadas num query result



2.23 – GROUP BY, ORDER BY e LIMIT clause examples

II.XIX 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 ...
```

2.24 – UDF Examples

II.XX Select – Aggregates

❖ Native

- COUNT(column), MIN(column), MAX(column), SUM(column) and AVG(column)

❖ User-Defined Aggregate Function (UDA)

- creation of custom aggregate functions

```
CREATE TABLE team_average (
  team_name text,
  cyclist_name text,
  cyclist_time_sec int,
  race_title text,
  PRIMARY KEY (team_name, race_title, cyclist_name)
);

-- UDA: calculate the average
-- value in the column
CREATE AGGREGATE average(int)
SFUNC avgState
STYPE tuple<int,bigint>
FINALFUNC avgFinal
INITCOND (0,0);

-- Test the function using a select statement
SELECT average(cyclist_time_sec) FROM team_average
WHERE team_name='UnitedHealthCare'
AND race_title='Amgen Tour';

1
2
3
4
5
```

— UDF: adds all the race times together and counts the number of entries.
CREATE OR REPLACE FUNCTION avgState (state tuple<int,bigint>, val int)
CALLED ON NULL INPUT
RETURNS tuple<int,bigint>
LANGUAGE java AS
\$\$ if (val !=null) {
 state.setInt(0, state.getInt(0)+1);
 state.setLong(1, state.getLong(1)+val.intValue());
}
return state; \$\$;

— UDF: computes the average of the values passed to it from the state function
CREATE OR REPLACE FUNCTION avgFinal (state tuple<int,bigint>)
CALLED ON NULL INPUT
RETURNS double
LANGUAGE java AS
\$\$ double r = 0;
if (state.getInt(0) == 0) return null;
r = state.getLong(1);
r/= state.getInt(0);
return Double.valueOf(r); \$\$;

2.25 – Aggregate Examples

II.XXI 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

```
CREATE TABLE users (
    username text PRIMARY KEY,
    firstname text,
    lastname text,
    birth_year int,
    country text
)
CREATE INDEX ON users(birth_year);
```

```
SELECT * FROM users;
SELECT * FROM users WHERE birth_year = 1981;
```



```
SELECT * FROM users WHERE birth_year = 1981 AND country = 'FR';
```



```
SELECT * FROM users WHERE birth_year = 1981 AND country = 'FR' ALLOW FILTERING;
```



2.24 – ALLOW FILTERING Examples

II.XXII Insert Statement

Insere uma nova Row numa dada Table:

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

Escreve uma ou mais colunas para uma dada row

Pelo menos as Primary 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 (
    'stesti',
    'Štěsti',
    ('Bohdan', 'Sláma'),
    2005,
    { 'vilhelmova': 'Monika', 'liska': 'Toník' },
    [ 'comedy', 'drama' ]
)
USING TTL 86400

```

2.25 – Insert Examples

II.XXIII Update Statement

Atualiza rows existentes dentro de uma dada table:

- Se a ROW com a dad primary key não existir, é inserida

Todas as Primary Key Columns têm de ser especificadas na WHERE clause

```

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';

```

2-26 – Update Example

```

UPDATE movies
SET
  year = 2006,
  director = ('Jan', 'Svěrák'),
  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'

```

```

UPDATE movies
SET
  actors['vilhelmova'] = 'Helenka',
  genres[1] = 'comedy'
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)
)

```

2-27 – Mais Update Examples

II.XXIV Insert e Update Parameters

❖ 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

```

first_name	last_name	writetime(last_name)
Mary	Rodriguez	1434591198790252
Bill	Nguyen	1434591198798235

2-28 – Insert e Update Parameters Examples

II.XXV Delete Statement

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

WHERE clause é usada para especificar qual row queremos deleted

Multiplas Rows podem ser apagadas com uma unica statement usando o **IN** operator

Um range de rows pode ser apagado usando um inequality operator (p.ex \geq)

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

DELETE phone FROM Users
WHERE userid IN (C73DE1D3-AF08-40F3-B124-3FF3E5109F22, B70DE1D0-9908-4AE3-BE34-5573E5B09F14);
```

2-29 – Delete Example