

Resumo de

CBD – 2

Escrito por Diogo Silva

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

Index

Column Databases.....	3
Graph Databases.....	31
Databases Distribuidas e Paralelas.....	58
Replicação de Dados Distribuidos.....	87
Partitioning de Dados Distribuidos	118
Processamento de Dados Distribuidos – MapReduce	135

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

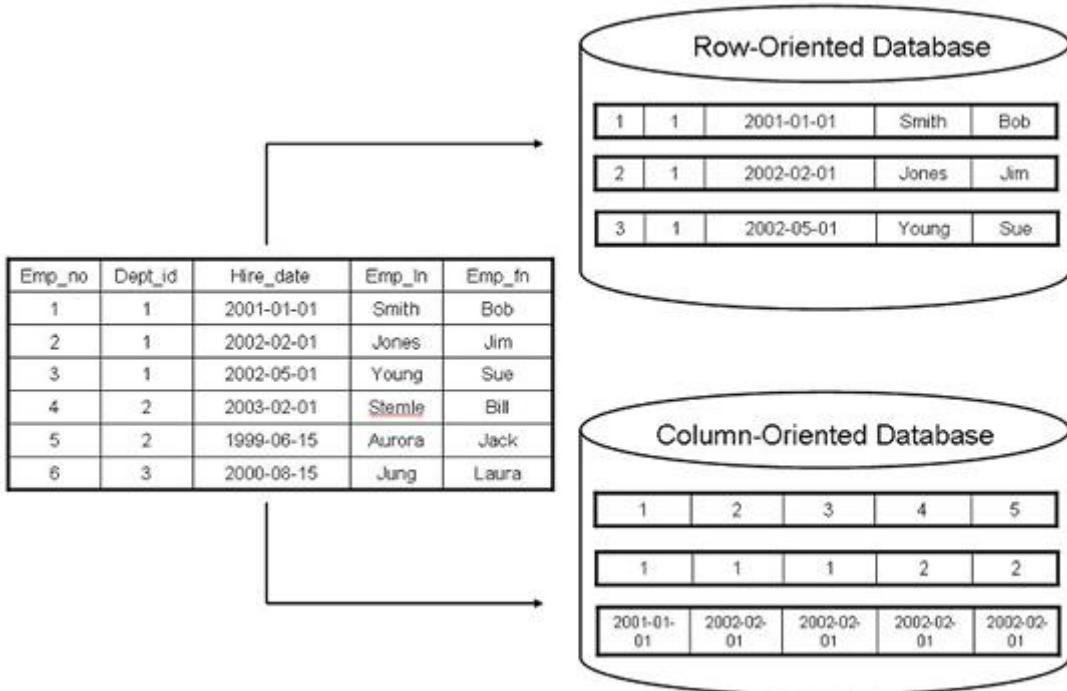
❖ Store by row:

```
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 correr 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 ótimas 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 Comparassion
- 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)



HYPERTABLE^{inc}

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

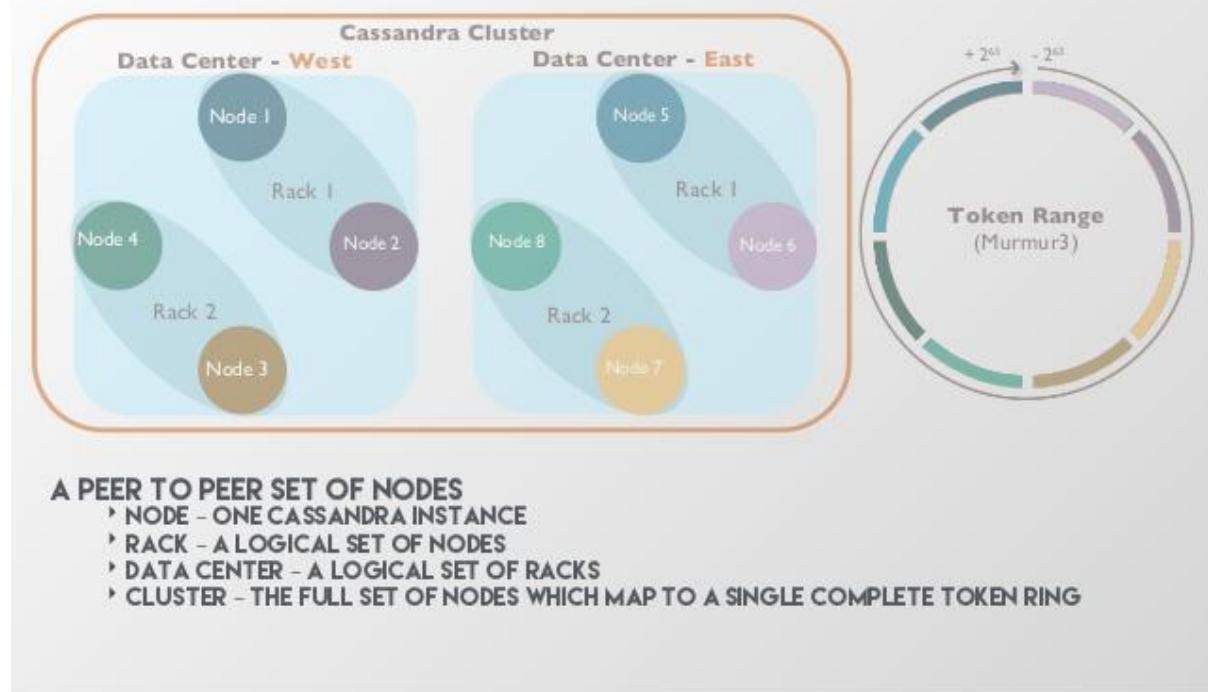
- Os mesmos dados são escritos em todos os Data Centers da Cluster

Cassandra Cluster



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

WHAT IS A C* CLUSTER?



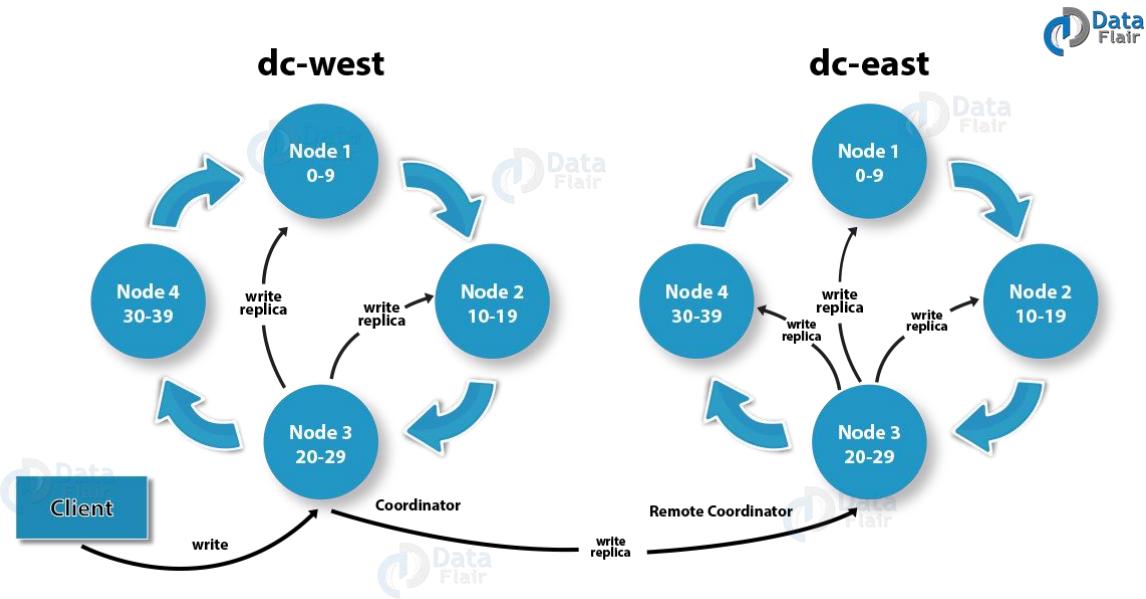
2.5 – Outro exemplo e explicação da Cassandra Node Cluter

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 bsaeadas na primary key
 - Row Lookups são rapidos
- 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

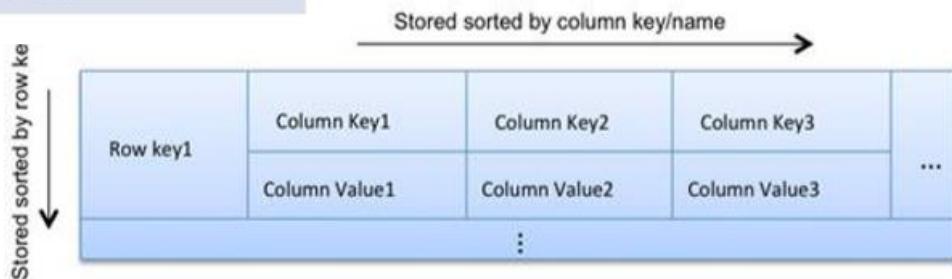
- Cada Row é **uniquely identified** por uma **primary key**

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 an 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

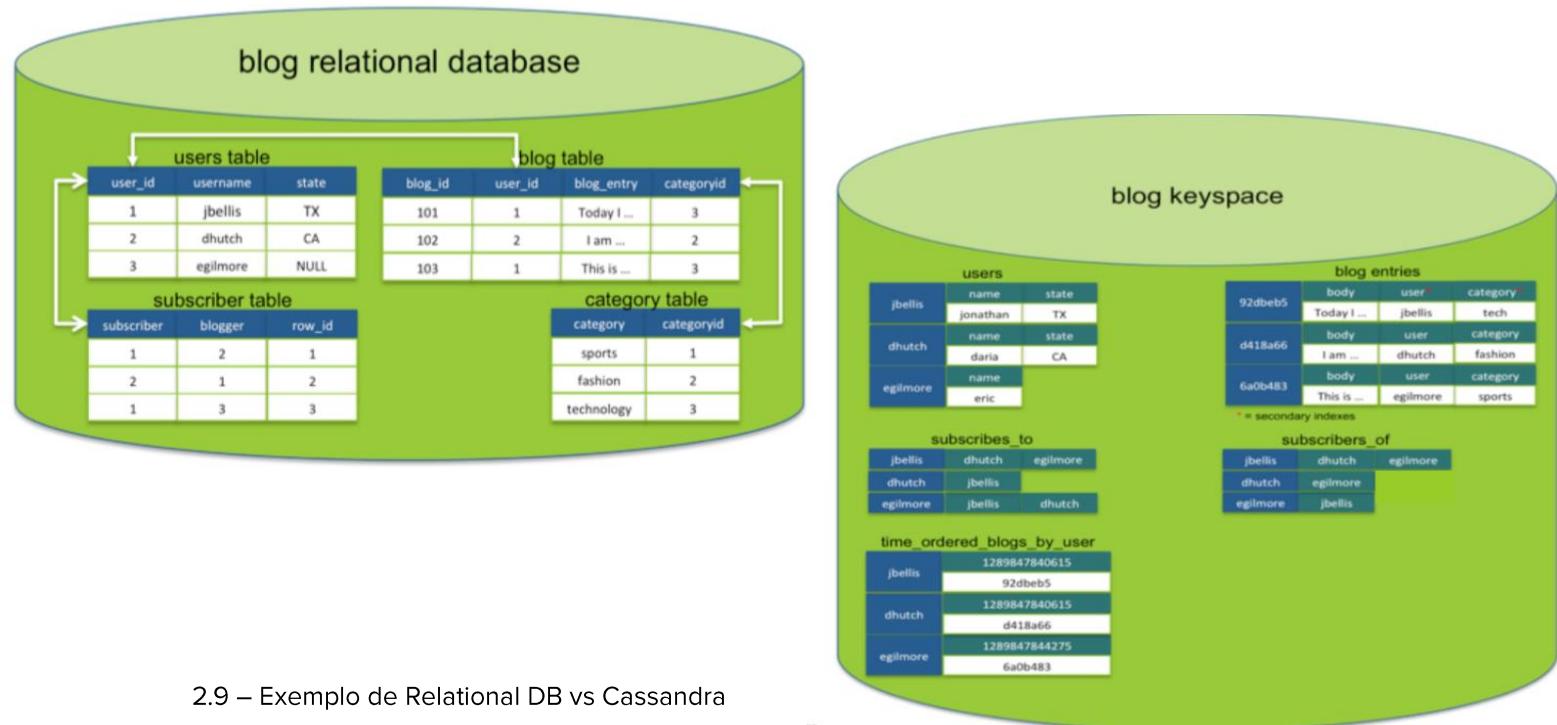
id			
trojan	name (Ivan, Trojan)	year 1964	movies { samotari, medvidek }
machacek	name (Jiří, Macháček)	year 1966	movies { medvidek, vratnelahve, samotari }
schneiderova	name (Jitka, Schneiderová)	year 1973	movies { samotari }
sverak	name (Zdeněk, Svěrák)	year 1936	movies { vratnelahve }

movies

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



2.8 – Exemplo de duas Column Families



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, ...

2.10 – Ferramentas para a utilização de Cassandra

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

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:

- Responsável por Data Distribuição (partitioning) pelos nodes da BD
- Pode ter multiplas colunas

Clustering Key:

- Responsável 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 ]
```

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;
```

videoname	ttl(videoname)	writetime(videoname)
Ondas gigantes na barra!	null	1509294890781000
Aviões de papel!	null	1509294888607000

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 contêm 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* (list, sets, maps) / maps**

```
>, >=, <= 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

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

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

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



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



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



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

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



-- multi-column IN restrictions can be applied to any set of clustering columns.

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

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

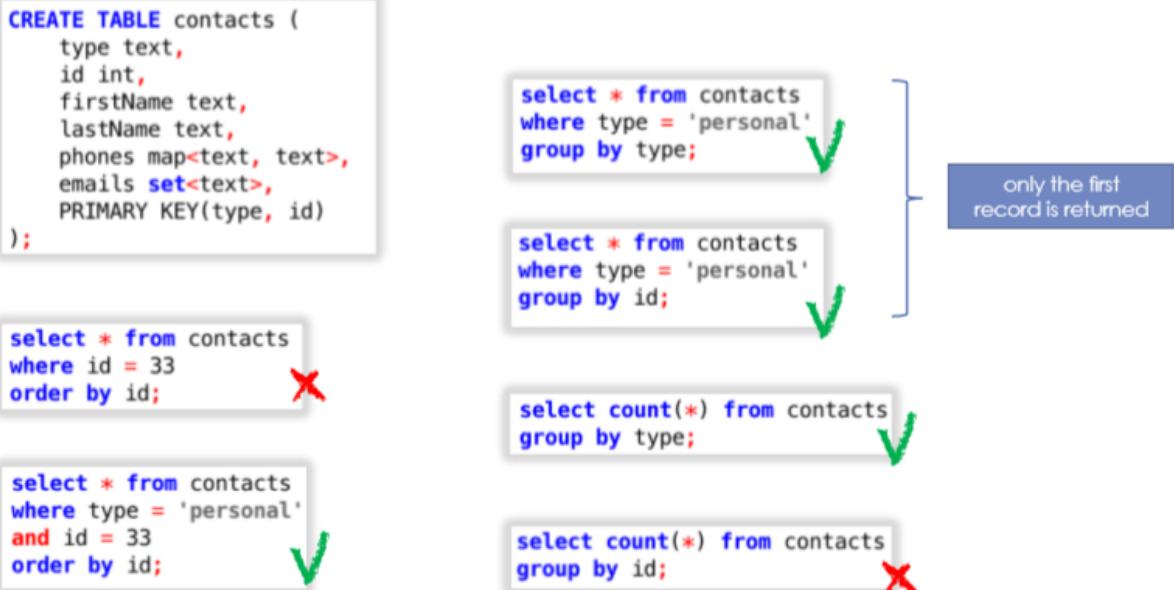
- Quando non-grouping columns são selecionadas sem uma aggregate function, o primeiro value encontrado é sempre retornado

ORDER BY clause:

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

LIMIT clause:

- Limita o numero de rows returnadas 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';
```

```
-- 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,
    { 'vilhelmove': '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

Graph Databases

I. Teoria dos Grafos

Antes de falarmos de Graph-Based Databases convém relembrarmos alguns conceitos relacionados com a teoria de Grafos.

Num grafo, os **dados** são representados como um conjunto de **entidades** e as **relações** que os unem

Para além da representação dos dados (que tem de ser eficientemente representada), são necessárias as seguintes **operações básicas** (que também têm de ser eficientes):

- Encontrar vizinhos de um node
- Verificar se dois nodes estão ligados por uma aresta
- Atualizar a estrutura dos grafos

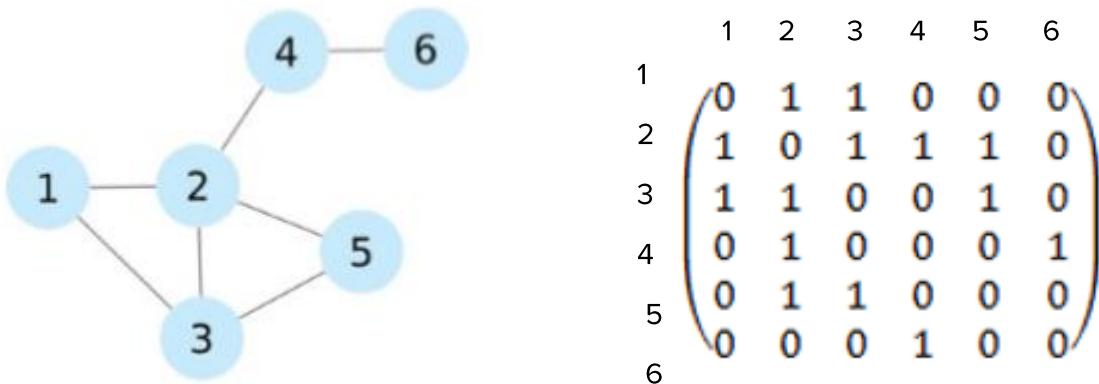
Em termos de numenclatura, é costume definirmos o Grafo, G, como:

- **G = (V,E)**, onde
 - V é o conjunto de Nodes (vertices)
 - E é o conjunto de Edges (arestas)
 - **n = |V|, m = |E|**

Para representarmos grafos é costume utilizarmos Matrizes. Algumas das opções incluem:

- **Matriz de Adjacencia**
 - É um array bidimensional – A – de n x n valores Booleanos (0 ou 1).
 - Cada Index corresponde a um node

- A_{ij} indica se dois nodes, i e j , estão ou não ligados (0 se não estiverem, 1 se estiverem)
- **Vantagens:**
 - Fácil verificar se dois nodes estão ligados
 - Adicionar e remover edges é fácil (basta mudarmos o valor na lista)
- **Desvantagens:**
 - Ocupa um espaço quadrático em função de n ($O(n^2)$)
 - É comum ficarmos com muitos 0s na matriz
 - Ir buscar todos os vizinhos é uma operação $O(n)$
 - Adicionar nodes é fodido (porque temos que adicionar as relações desse novo node com todos os outros)
- Algumas variações deste grafo incluem Directed Graphs e Weighted Graphs



Img 1.1 – Exemplo da Matriz Da Adjacencia de um Grafo

• Lista de Adjacencia

- É um conjunto de listas na qual cada uma vai representar os vizinhos de um node.
- O conjunto das listas de adjacencias vai ser vetor de n ponteiros para listas de adjacencia
- Nos grafos undirected se uma aresta ligar i e j , então a lista de vizinhos de i contém o node j e vice versa
- **Vantagens:**
 - Obter os vizinhos de um node é fácil
 - Adicionar nodes à estrutura é barato

- É uma representação compacta de matrizes esparsas (onde teríamos muitos 0s)
- **Desvantagens:**
 - Verificar a aresta entre dois nodes



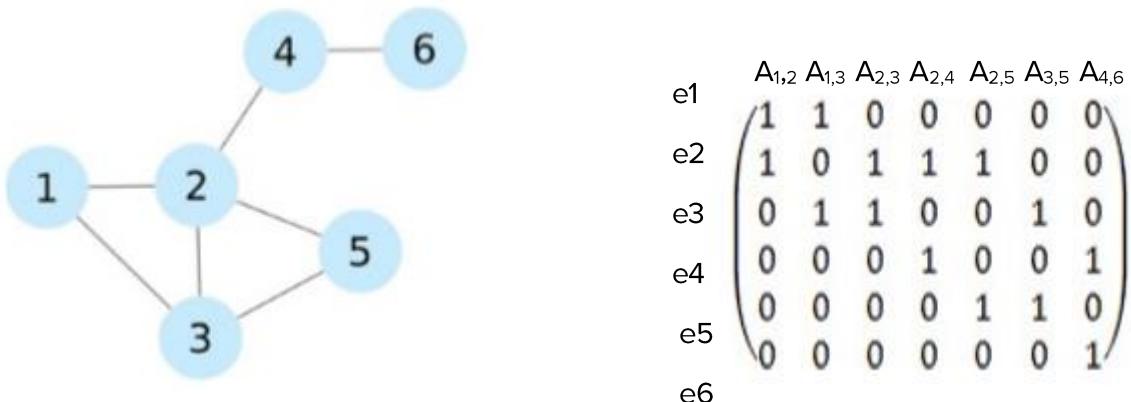
Img 1.2 – Exemplo do conjunto de Listas de Adjacencias

• Matriz de Incidencia

- É uma matriz booleana bi-dimensional com n rows e m colunas
- Cada coluna representa uma aresta
- Cada row representa um node
- **Vantagens:**
 - Representar hipergrafos (onde um edge se liga a um numero arbitrario de nodes)

○ Desvantagens:

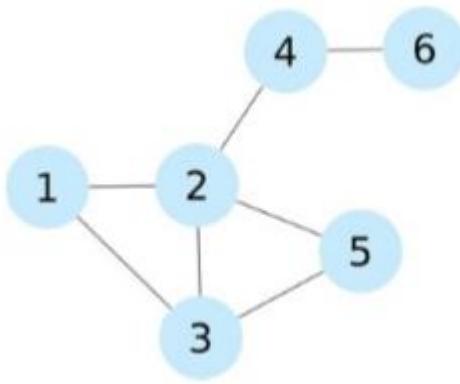
- Requer $n \times m$ bits



Img 1.3 – Exemplo da Matriz Da Incidencia

- **Matriz de Laplace**

- Array bidimensional de $n \times n$ inteiros
- A diagonal da matriz laplaciana indica o **grau** – número de vizinhos - do node
- As outras posições são -1 se dois vertices estiverem ligados ou 0 se não estiverem
- **Vantagens**
 - Permite analizar a estrutura dos grafos através de análise spectral
- **Desvantagens**
 - As mesmas que a da matriz de adjacencia



	1	2	3	4	5	6
1	2	-1	-1	0	0	0
2	-1	4	-1	-1	-1	0
3	-1	-1	3	0	-1	0
4	0	-1	0	2	0	-1
5	0	-1	-1	0	2	0
6	0	0	0	-1	0	1

Img 1.4 – Exemplo da Matriz De Laplace

Para além de sabermos como representar grafos, convém também saber como é que podemos caminhar neles – **Graph Traversal**

Single step **traversal** from element i to element j , where $i, j \in (V \cup E)$

Expose explicit **adjacencies** in the graph

- e_{out} : traverse to the outgoing edges of the vertices
- e_{in} : traverse to the incoming edges of the vertices
- v_{out} : traverse to the outgoing vertices of the edges
- v_{in} : traverse to the incoming vertices of the edges
- e_{lab} : allow (or filter) all edges with the label
- ϵ : get element property values for key r
- e_p : allow (or filter) all elements with the property s for key r
- $\epsilon=$: allow (or filter) all elements that are the provided element

Img 1.5 – Single Step Traversal...no clue o que é que está aqui escrito :/

Conseguimos usar **single step traversals** para compor **complex traversals** de length arbitrária.

Ex. “Encontrar todos os amigos do DS” (plot twist, he has none)

Complex traversals correspondem a viajarmos para todos os outgoing edges de um vertice i (neste caso seria DS), e depois permitir apenas as arestas que tenham o label que queremos (neste caso, friend), depois viajar para todos os incoming vertices dessas arestas.

Para terminar bastava retornar-mos a name property desses nodes

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

Img 1.6 – Ex de uma função de Complex Traversal

Para terminar esta secção vamos só falar de dois tipos de grafos diferentes:

- **Single Relational**
 - Arestas são homogeneas em significado
 - Ex. Todas as arestas representam “amigo de”
- **Multi Relational (property)**
 - Edges são typed ou labeled
 - Ex. Alguns edges são friendship, outros business, ...
 - Vertices e edges num property graph mantêm um conjunto de key/valu pairs
 - Representação de non-graphical data (properties)
 - Ex. Nome do vertice, weight do edge

II. Graph Databases (not Graph-Oriented DBs)

Antes de vermos Graph-Oriented DataBases temos de ver o que é uma base de dados de grafos.

Uma **graph database** é um **conjunto de grafos**

Podemos ter dois tipos de grafos:

- **Directed-labeled Graphs**
 - Ex. XML, RDF, Traffic Networks
- **Undirected-labeled Graphs**
 - Ex. Redes Sociais, Chemical Compounds

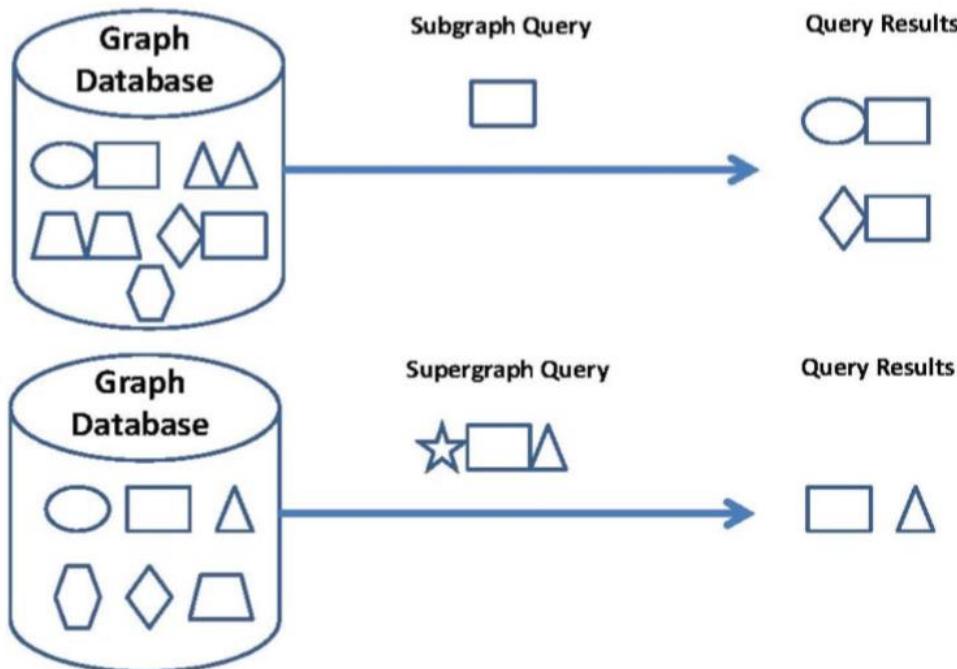
e dois tipos de graph databases:

- **Transactional**
 - Vários grafos pequenos
 - Ex. Chemical compounds, biological pathway
- **Non-Transactional**
 - Poucos grafos grandes
 - Ex. Redes Sociais, Web Graph

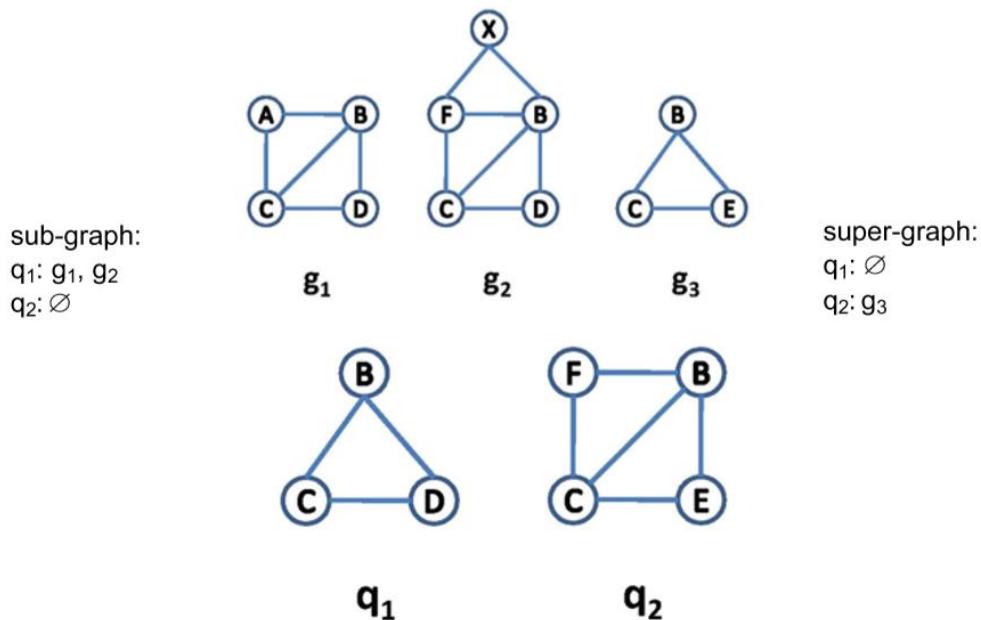
Nas Transactional Graph Databases temos 3 tipos de **queries**:

- **Sub-Graph Queries**
 - Pesquisas para um **padrão específico na graph database** (i.e um subgrafo do grafo)
 - Retorna um grafo mais pequeno do que o grafo original ou um grafo onde algumas partes sejam uncertain
 - O tipo mais geral é SubGraph Isomorphism
- **Super-Graph Queries**
 - Pesquisa pelos **membros da graph database nos quais a estrutura inteira estejam contidas na input query**
- **Similarity (Approximate Matching) Queries**
 - Encontra grafos que sejam similares, mas não necessariamente isomórficos
 - Encontra sub-grafos na bd que sejam similares à query. Permite node mismatches, node gaps, structural differences, etc...

- É usado quando temos graph databases noisy ou incompletas. Nestes casos Approximate Graph Matching Query-Processing techniques podem ser mais uteis e effective que exact matching



Img 1.7 – SubGraph vs SuperGraph queries



Img 1.8 – SubGraph vs SuperGraph query results. O grafo q_1 é o resultado da sub-graph query q_1 (retorna o grafo que tenha um padrão de g_1 e g_2), que é equivalente à super graph query q_1 (grafo que não seja igual ao g_1). O grafo q_2 é o resultado da sub-graph query q_1 (retorna o grafo que não tenha o padrão igual ao g_2), que é equivalente à super graph query q_2 (retorna grafo que contenha o padrão do g_3).

Para terminar esta secão vamos so falar da **Sub-Graph Query Processing**. Temos duas possiveis técnicas de processamento:

- **Mining-Based Graph Indexing Techniques**

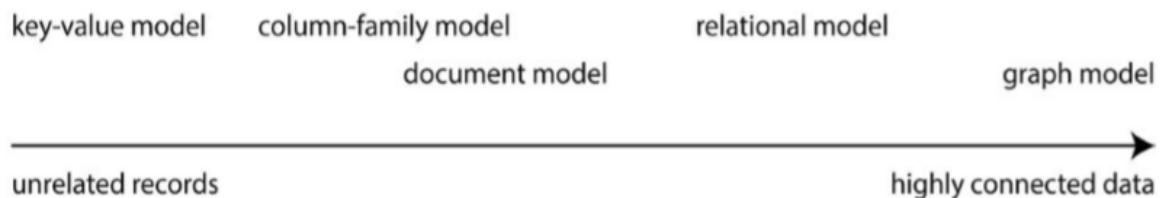
- A ideia destas técnicas é que se features de um query graph q não existir num data graph G, então G não pode conter q como sub-grafo
- Graph mining methods extraem selected features (sub-structures) dos membros da graph database
- É criado um inverted index para cada feature
- Responder a um sub-graph query q corresponde a:
 - Identificar set das features de q
 - Usar um inverted index parar ir buscar todos os grafos que contenham as mesmas features que q

- **Non-Mining-Based Graph Indexing Techniques**

- Focarmo-nos em indexer os constructs inteiros de uma graph database, em vez de indexarmos apenas algumas selected features
- Vantagens:
 - Consegue gerar graph updates com menos custos
 - Não está dependente da effectiveness das selected features
 - Não precisa de reconstruir indexes inteiros
- Desvantagens:
 - Consegue ser menos effective no que toca a filtragem
 - Pode precisar de realizar expensive structure comparasions no processo de filtragem

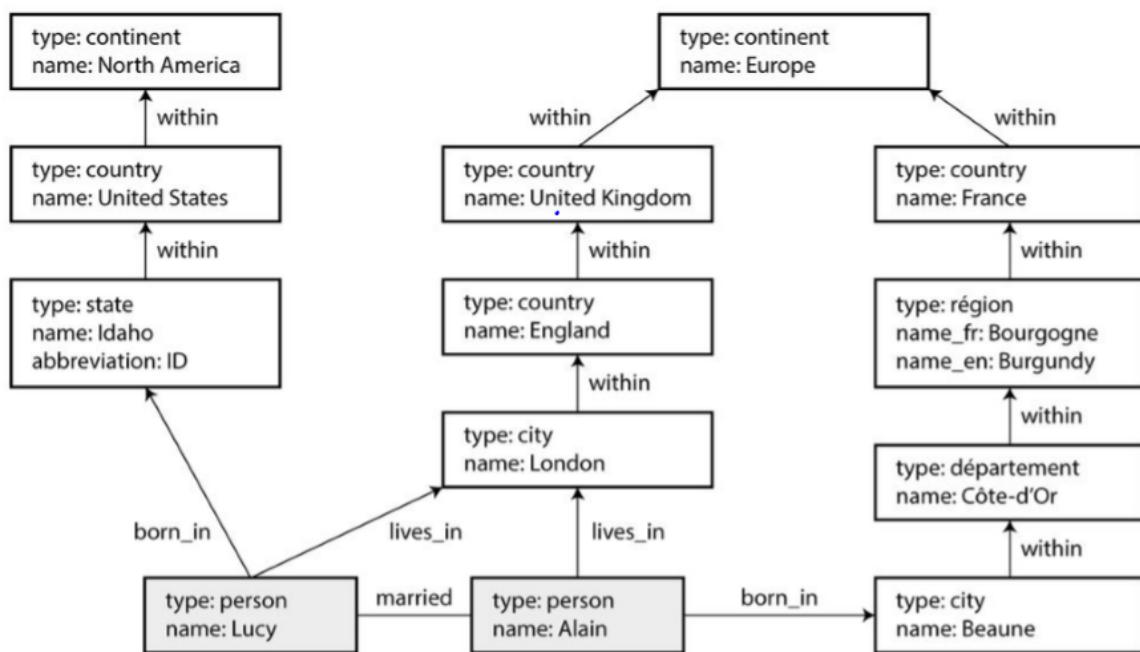
III. Graph-Like Data Models

Relações Many-To-Many são importantes para distinguir different data models



Img 2.1 – Types of Databases

O Relational Model consegue tartar casos simples de Many-To-Many relationships, mas quando as coneçções se tornam mais complexas, a utilização de grafos é muito mais natural



Img 2.2 – Graph Like Data Models

Os grafos são constituídos por dois tipos de objetos:

- Vertices (nodes or entities)
- Edges (relationships)

Muitos tipos de dados podem ser modelados como um grafo.
Alguns exemplos incluem:

- Social Graphs
 - Vertices são pessoas
 - Edges indicam as pessoas que se conhecem umas as outras
- Web Graphs
 - Vertices representam web pages
 - Edges indicam links HTML para outras páginas
- Road ou Rail Networks
 - Vertices são junctions e edges estradas ou caminhos ferreiros entre elas

Para além disso, existem muitos algoritmos famosos que operam sobre grafos:

- PageRank
- Shortest Path

Existem vários tipos diferentes de **struturar** e fazer **queries** de dados em grafos. Dois exemplos incluem:

- **Triple Store Model**
 - Implementado por Datomic, AllegroGraph, etc..
- **Property Graph Model**
 - Implementado por Neo4J, Titan, ...

Algumas linguagens de query para grafos são:

- **Cypher** (usada pelo Neo4j)
- SPARQL
- Datalog

IV. Property Graphs

Num property graphs...

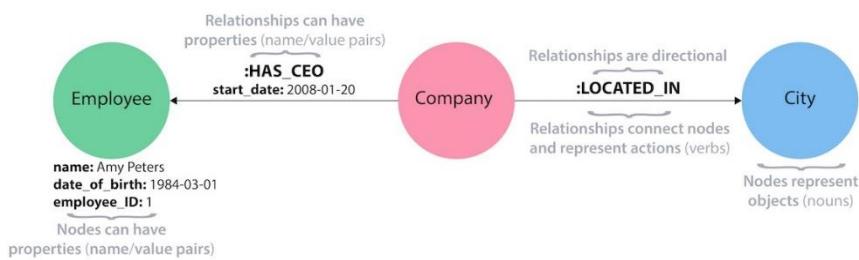
- Cada **Vertex** consiste num:
 - Unique Identifier
 - Set de outgoing edges
 - Set de incoming edges
 - Coleção de propriedades (key-valued pairs)
- Cada **Edge** consiste em:
 - Unique Identifier
 - O vertice no qual o edge começa (tail vertex)
 - O vertice no qual o edge termina (head vertex)
 - Um label que descreve o tipo da relação entre os dois vertices
 - Coleção de propriedades (key-value pairs)

Qualquer vertice pode ter um edge que o ligue a outro vertice.

Não ha nenhum schema que restrinja o tipo de coisas que podem ou não estar associadas

Dado um vertice podemos eficientemente encontrar todos os incoming e outgoing edges, portanto podemos eficientemente caminhar o grafo

São usados labels diferentes para diferentes relações, pelo que é permitido guardarmos diferentes tipos de informação num unico grafo e manter na mesma um modelo de dados limpo

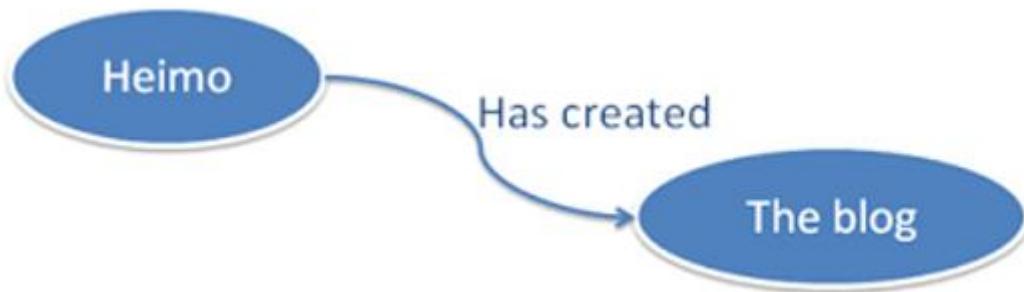


Img 4.1 – Exemplo de um simples Property Graph (com 3 nodes e 2 relações)

V. Triple-Stores

O **Triple-Store Model** é mais ou menos equivalente ao Property Graph Model (basically usa palavras diferentes para descrever as mesmas ideias)

Neste modelo, a informação é guardada como parte de uma **three-part statement** – Sujeito, Predicado, Objeto



Img 5.1 – Exemplo de um three part statement – O Heimo Criou O Blog

O **sujeito** do triplo é equivalente a um **vertice** no grafo

O **objeto** pode ser:

- Um Valor num primitive datatype (p.ex uma string ou um numero)
 - Nesse caso o predicado e o objeto são equivalentes a uma key,value property do subject vertex
 - P.ex (lucy,age,33) simboliza o vertice lucy com as propriedades age - 33
- Outro Vertice no grafo
 - Nesse caso o predicado é um edge no grafo, o sujeito é o tail vertex e o objeto é o head vertex
 - P.ex (lucy, marriedTo, alan)
 - ❖ Using *Turtle*, a format that is a subset of Notation3 (N3).

Img 5.2 –
Exemplos de
statements
usando Turtle

```
@prefix : <urn:x-example:>.  
:_lucy a :Person; :name "Lucy"; :bornIn _:idaho.  
:_idaho a :Location; :name "Idaho"; :type "state"; :within _:usa.  
:_usa a :Location; :name "United States"; :type "country"; :within _:namerica.  
:_namerica a :Location; :name "North America"; :type "continent".
```

VI. Graph Oriented Databases

Graph Oriented Databases são um tipo de NoSQL Databases que se focam no tratamento de dados altamente conectados.

Estas BDs focam-se em **modelar a estrutura do grafo** e as **propriedades**.

Os **grafos** podem ser **direcionados** ou **não** e constituem numa **coleção** de **nodes** (real world entities) e **vertices** (relationships entre eles).

Tanto os **nodes** como as **relações** podem **possuir propriedades**

Os **query patterns** incluem:

- Criar, atualizar ou remover nodes/relationships num grafo
- Algoritmos de grafos (shortest paths, spanning trees, ...)
- General Graph Traversals
- Queries Sub-Graph e Super-Graph
- Similarity Queries (matching aproximado)

Use Cases:

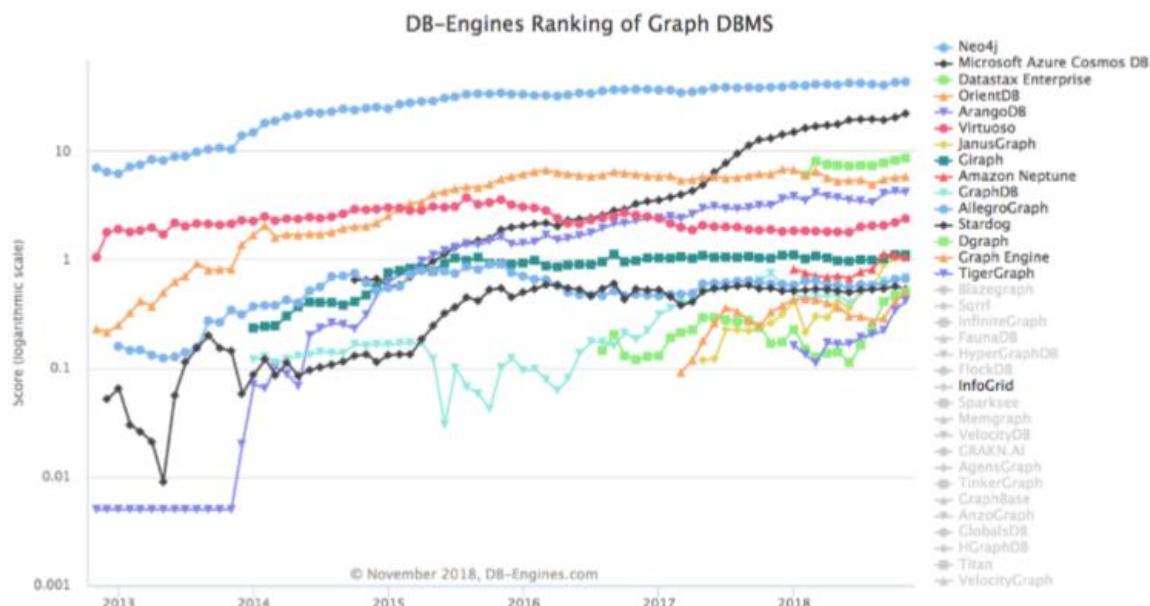
- Social networks, routing, dispatch e location-based systems, recommendation engines, chemical compounds, biological pathways, linguistic trees, ...
- I.e quando é natural representar os dados com um grafos

Not Use Cases:

- Quando precisamos de extensas batch operations
 - Operações que afetem muitos nodes/relationships
- Quando grafos muito grandes têm de ser guardados
 - Graph Distribution é difícil ou impossível



6.1 – Exemplos de Graph Databases. O Neo4j, Titan, Apache Giraph, Infinite Graph e FlockDB são representativos, enquanto que o OrientDB, OpenLink Virtuoso e Arango



6.2 – Panorama de popularidade das Graph DBs

VII. Neo4j

O **Neo4j** é uma *Graph Database Open Source*, com alta **escalabilidade, availability, fault tolerant**, com **master slave replication** e que permite **ACID Transactions** e é **embeddable**.

Foi desenvolvida pela Neo Technology e implementada em Java, tendo sido lançada em 2007 como cross-platform.



7.1 – Neo4j Logo

Como Query Language utiliza o **Cypher** (altamente expressiva).

Fornece ainda uma **traversal framework**

As features do Neo4j são:

- **Data Model – Flexible Schema**
 - O Neo4j segue um data model chamado **Native Property Graph Model**
 - O Grafo vai conter **nodes** (entidades) e esses nodes vão estar ligados entre si (representado através de **relationships**).
 - Nodes e Relationships guardam dados no formato **key-value** chamados **propriedades**
 - Em Neo4j não há necessidade de seguir um fixed schema
- **Propriedades ACID**
 - Neo4j suporta regras ACID (Atomicity, Consistency, Isolation e Durability)

- **Escalabilidade e Reliability**

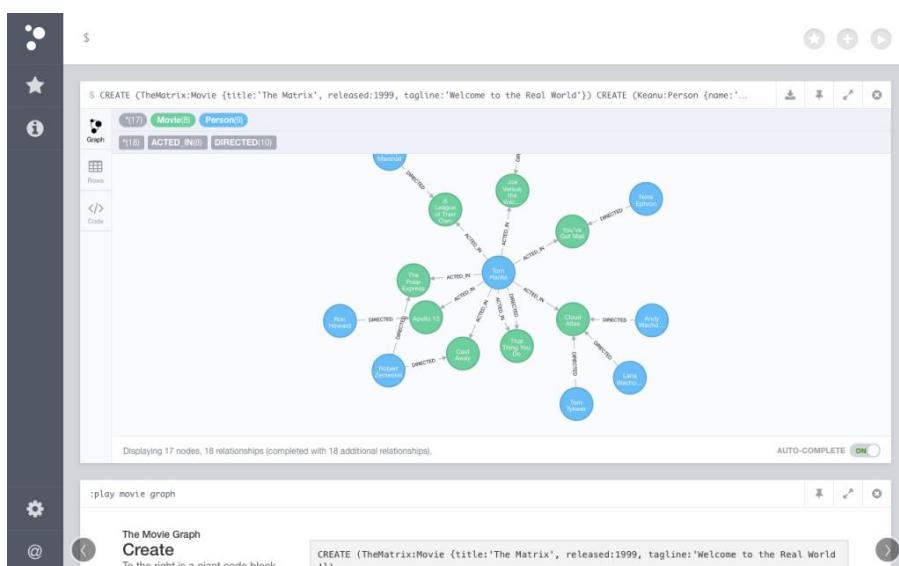
- Podemos escalar databases aumentando o numero de reads/writes e o volume de dados sem que isso afete a velocidade de processamento de queries ou a integridade dos dados
- Neo4j fornece também suporte para **replication de dados** para segurança e reliability

- **Cypher Query Language**

- O Neo4j utiliza uma query language declarativa muito poderosa conhecida como Cypher.
- Utiliza ASCII-art para representar grafos
- Cypher é facil de aprender e pode ser usado para criar e ir buscar relações usando dados sem ser preciso utilizar queries complexas como JOINS

- **Built-In Web Application**

- O Neo4j fornece uma built-in **Neo4j Browser Web Application** com a qual Podemos criar e realizar queries aos dados do grafo



7.2 – Neo4j Web App

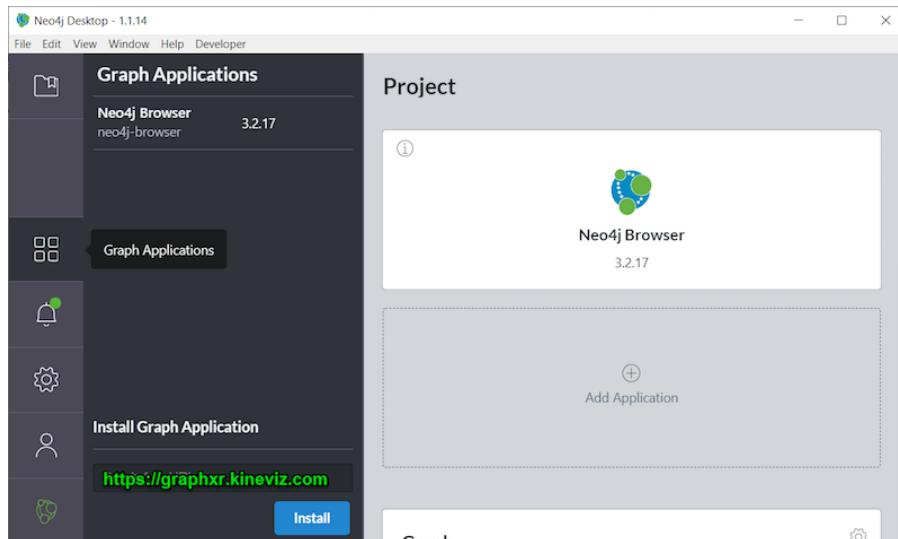
- **Drivers**

- O Neo4j consegue trabalhar com várias Drivers

- Suporta REST API para trabalhar com programming languages e technologies como Java, Spring, Scala, ...
- Suporta Java Script para trabalhar com UI MVC Frameworks como Node JS
- Para desenvolver Java Applications suporta dois tipos de Java API: Cypher API e Native Java API
- Etc...

- **Indexação**

- Suporta indexes utilizando Apache Lucene



7.3 – Neo4j Interface

VII.I Data Model

A estrutura do Database System do Neo4j é:

- **Instance -> Single Graph**

O grafo vai ser um Property Graph – **Directed Labeled Multigraph** (coleção de vertices e arestas)

Cada **Node**:

- Tem um **Unique Internal Identifier**
- Pode estar associado a um conjunto de **labels** (que nos permitem categorizar os nodes)
- Pode estar associado a um set de **propriedades** (que nos permite guardar dados adicionais nos nodes)

Cada **Relationship**:

- Tem um **Unique Internal Identifier**
- Tem uma **direção**
 - Relações podem ser facilmente viajadas em qualquer direção
 - Direções podem ignoradas nas queries
- Tem um **start** e um **end node** (podem ser recursivas – i.e loops/ciclos são permitidos)
- Está associado a um exatamente um **type**
- Pode estar associado a um conjunto de **propriedades**

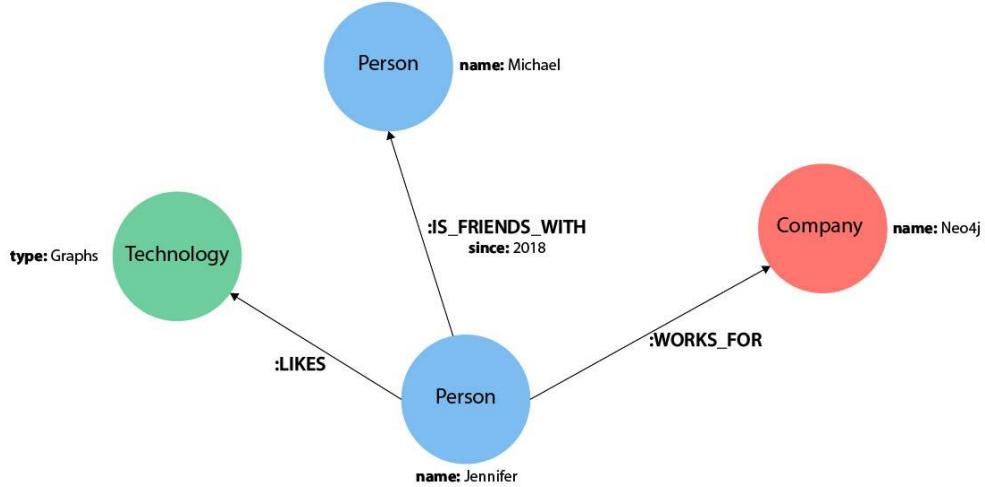
As **Propriedades** dos nodes e/ou relações:

- São um **Key-Value Pair**
 - **Key** é uma string
 - **Value** é um valor atomic de um qualquer primitive data type ou um array de values de um primitive data type

Os **Primitive data types** são:

- **Boolean**
 - True ou False
- **Byte, Short, Int, Long**
 - Inteiros com 1 Byte, 2 Bytes, 4 Bytes ou 8 Bytes
- **Float, Double**
 - Floating-Point Numbers com 4 Bytes ou 8 Bytes

- **Char**
 - Um caracter Unicode
- **String**
 - Uma sequencia de Unicode Chars



7.1.1 – Data Model do Neo4j – notem-se os Labels dos nodes (Person, Company e Technology), as propriedades dos nodes (type, name), as relações (dos tipos – likes, isfriendswith e worksfor) e propriedades (since)

VIII.II Cypher

O **Cypher** é a query language usada pelo Neo4j

É uma Delcarative Graph Query Language que permite querying e updates expressivos e eficientes

Clauses, como p.ex MATCH, RETURN, CREATE, ..., (quase de forma arbitaria) chained together.

O resultado intermediario de uma clause passa para a clause subsequente

VII.III Cypher Nodes

O Cypher usa um **par** de **parenteses** para representar Nodes (quase que como um circulo ou retangulo com rounded corners, dai a ideia que o Cypher usa ASCII-art para representar grafos)

○

- Represents an anonymous, uncharacterized **node**.

(matrix)

- If we want to refer to the node elsewhere, we can add an variable

(:Movie)

- The Movie **label** declares the node's type or role

(matrix:Movie)

(matrix:Movie {title: "The Matrix"})

(matrix:Movie {title: "The Matrix", released: 1999})



- The node's **properties** (title, released, et cetera) are represented as a list of key/value pairs, enclosed within a pair of braces

(matrix:Movie:Promoted)

7.3.1 – Representação dos Nodes em Cypher

VII.IV Cypher Relações

O Cypher usa um **setas** para representar relações

- , ->, <-

- Relationships are arrows pointing from one node to another

(node1)-[:REL_TYPE]->(node2)

- General relation, from node1 to node2

(actor:Person)-[:ACTED_IN]->(movie:Movie)

- Retrieve all nodes that had a relationship type ACTED_IN with other nodes.

‣ **Query examples**

```
MATCH (node1)-[rel:TYPE]->(node2)
RETURN rel.property
```

- Generic format, from node to node2.

```
MATCH (actor:Person)-[rel:ACTED_IN]->(movie:Movie)
```

```
RETURN rel.roles
```

- Roles of actors that acted in any movie.

```
MATCH (n)-->(m) RETURN n, m;
```

- every pair of nodes with a relationship going from n to m.

7.4.1 – Representação das Relationships em Cypher. Inclui também alguns exemplos de queries

VII.V Cypher Padrões

Combinando a sintaxe das relações e dos nodes, podemos criar **padrões**

```
MATCH (matrix:Movie {title:"The Matrix"} )  
  <-[role:ACTED_IN {roles:["Neo"]}]-->  
    (keanu:Person {name:"Keanu Reeves"})  
RETURN matrix, role, keanu
```



7.5.1 –

```
MATCH cast = (:Person)-[:ACTED_IN]->(:Movie)  
RETURN cast
```

Representação de Padrões em Cypher

VII.VI Selection

Realizado por MATCH...RETURN Clauses

MATCH

```
MATCH (n) RETURN n
```

- all nodes

```
MATCH (me:Person) WHERE me.name="My Name" RETURN me.name  
MATCH (me:Person {name:"My Name"}) RETURN me.name
```

```
MATCH (movie:Movie)  
WHERE movie.title = "Mystic River"  
SET movie.released = 2003  
RETURN movie.title AS title, movie.released AS released
```

title	released
Mystic River	2003

7.6.1 – Match Clause Example

VII.VII Filtering

Realizado com WHERE queries

WHERE

```
MATCH (tom:Person)-[:ACTED_IN]->()-[:ACTED_IN]-(actor:Person)
WHERE tom.name="Tom Hanks" AND actor.born < tom.born
RETURN DISTINCT actor.name AS Name
• ??
MATCH (gene:Person)-[:ACTED_IN]->()-[:ACTED_IN]-(other:Person)
WHERE gene.name="Gene Hackman" AND exists( (other)-[:DIRECTED]->() )
RETURN DISTINCT other
• ??
MATCH (gene:Person {name:"Gene Hackman"})-[:ACTED_IN]->(movie:Movie),
(other:Person)-[:ACTED_IN]->(movie),
(robin:Person {name:"Robin Williams"})
WHERE NOT exists( (robin)-[:ACTED_IN]->(movie) )
RETURN DISTINCT other
• ??
```

7.7.1 – Filtering Query

VII.VIII Ordering

Com Cypher podemos Ordenar, Limitar, ir buscar só os Distinct

ORDER BY, LIMIT, SKYP, DISTINCT

Return the five oldest people in the database

```
MATCH (person:Person)
RETURN person
ORDER BY person.born
LIMIT 5;
```

List of the oldest actors

```
MATCH (actor:Person)-[:ACTED_IN]->()
RETURN DISTINCT actor
ORDER BY actor.born
```

7.8.1 – Limitação e Ordenação por propriedades

VII.IX Variable Length Paths

```
MATCH (node1)-[*]-(node2)
```

- ❖ Relationships that traverse any depth are:
`(a)-[*]->(b)`
- ❖ Specific depth of relationships
`(a)-[*depth]->(b)`
- ❖ Relationships from one to four levels deep
`(a)-[*1..4]->(b)`
- ❖ Relationships of type KNOWS at 3 levels distance:
`(a)-[:KNOWS*3]->(b)`
- ❖ Relationships of type KNOWS or LIKES from 2 levels distance:
`(a)-[:KNOWS|:LIKES*2..]->(b)`

7.9.1 – Variable Length Path Queries

VII.X Indexes

O **Neo4j NÃO** usa indexes para acelerar os JOINs.

Em vez disso, os Indexes são úteis para encontrar **starting points** por **value**, textual **prefix** ou **range**.

Por exemplo:

To search efficiently people by name:

```
CREATE INDEX ON :Person(name);
```

Now, the lookup of "Gene Hackman" will be faster

```
MATCH (gene:Person)-[:ACTED_IN]->(movie),
      (other:Person)-[:ACTED_IN]->(movie)
WHERE gene.name="Gene Hackman"
RETURN DISTINCT other;
```

To remove the index:

```
DROP INDEX ON :Person(name);
```

7.10.1 – Exemplo do uso de Indexes

VII.XI Agregação

O **Cypher** fornece apoio para vários tipos de aggregate functions, como count, min, max, avg, sum e collect

- **count(x)** Count the number of occurrences
- **min(x)** Get the lowest value
- **max(x)** Get the highest value
- **avg(x)** Get the average of a numeric value
- **sum(x)** Sum up values
- **collect(x)** Collect all the values into an collection

```
MATCH (person:Person)-[:ACTED_IN]->(movie:Movie)
RETURN person.name, count(movie)
ORDER BY count(movie) DESC
LIMIT 10;
```

- Top ten actors who acted in the most movies

7.11.1 – Exemplo de queries de aggregation

VII.XII Creating/Deleting

Criação de Nodes:

```
CREATE (n: <Tag> {<Property 1>: <Value 1>, <Property 2>: <Value 2>, ...})  
  
CREATE (node:label { key1: value, key2: value, ... })  
  
CREATE (Aveiro)  
  
CREATE (Porto),(Coimbra),(Espinho)  
  
CREATE (ric:person:player)  
  
CREATE (leo:person:player)  
  
CREATE (aveiro:cidade{name:"Aveiro"})  
  
CREATE (ricg:player{name: "Ricardo Gomes",  
YOB: 1985, POB: "Porto"})
```



7.12.1 – Queries de Criação de Nodes

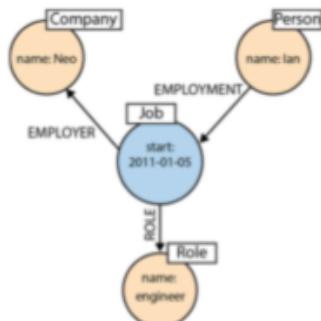
Criação de Relações:

```
CREATE <starting node>-[r:<relationship name> {<relationship properties>}]-><ending node>
```

```
CREATE (RuiPatricio:player{name: "Rui Patrício", YOB: 1988, POB: "Leiria"})
CREATE (PT:Country {name: "Portugal"})
CREATE (RuiPatricio)-[r:Guarda_Redes]->(PT)
CREATE (RuiPatricio)-[:JogadorDeFutebol]->(PT)
RETURN RuiPatricio, PT
```



```
CREATE (:Person {name:'Ian'})-[:EMPLOYMENT]->
    (employment:Job
        {start_date:'2011-01-05'}) -[:EMPLOYER]->
    (:Company {name:'Neo'}), (employment)-[:ROLE]->
    (:Role {name:'engineer'})
```



7.12.2 – Queries de Criação de Relações

Remoção de Nodes e Relações:

O **DELETE** vai remover nodes, relationships ou paths de um data graph.

As Relações têm de ser removidas antes dos Nodes, a não ser que se especifique o modifier DETACH

```
MATCH (<node> {<parameter 1>: <value 1>, ...})
<DETACH> DELETE n
```

```

MATCH (p:Person {name:"My Name"})
DELETE p
    • Remove node "My name". Error if it has relations
MATCH (me:Person {name:"My Name"})
OPTIONAL MATCH (me)-[r]-()
DELETE me,r
    • Remove me (node "My name") and any relationships with "me"..
MATCH (n) DETACH DELETE n
    • delete all nodes and relationships

```

7.12.3 – Queries de Delete

VII.XIII Importação de Dados

O **Neo4j** permite-nos importar dados de CSVs

Content of "movies.csv"

```

id,title,country,year
1,Wall Street,USA,1987
2,The American President,USA,1995
3,The Shawshank Redemption,USA,1994

```

In Cypher:

```

LOAD CSV WITH HEADERS
FROM "http://neo4j.com/docs/stable/csv/intro/movies.csv"
AS line
CREATE (movie:Movie
{ id:line.id, title:line.title, released:Int(line.year) });

```

7.13.1 – Como carregar os conteudos de um CSV em Cypher

VII.XIV Outras Write Clauses

SET clause

- allows to...
 - set a value for a property, or remove a property when NULL is assigned
 - replace all the current properties with new ones
 - add new properties to the existing ones
 - add labels to nodes
- Cannot be used to set relationship types

REMOVE clause

- Allows to...
 - remove a particular property
 - remove labels from nodes
- Cannot be used to remove relationship types

7.14.1 – Clauses SET e REMOVE. Mais informação de Neo4j em

<https://github.com/HerouFenix/CBD/tree/master/Pratica4#initial-neo4j-interactions>

Databases Distribuidas e Paralelas

I. Bases de Dados Distribuidas

Até agora temos sempre pensado nas bases de dados como sendo unidades centralizadas, mas efetivamente podemos ter bases de dados **centralizadas ou distribuídas**

- **Centralized Databases**

- Dados localizados num único sítio
- Todas as funcionalidades DBMS são realizadas por um servidor
- Este servidor vai estar encarregue de:
 - Enforçar propriedades ACID das transações
 - Controlo de concorrência
 - Mecanismos de recuperação
 - Responder a Queries
- This might be a bit too much load on a single server...

- **Distributed Databases**

- Dados guardados em múltiplos sítios (cada um a correr a DBMS)
- Nova noção de transações distribuídas
- As funcionalidades DBMS vão estar distribuídas sobre múltiplas máquinas

Devemos considerar bases de dados distribuídas porque:

- **Scalability**

- Se os **volumes de dados, read load ou write load** se tornar **muito grande** para uma única máquina tratar

podemos potencialmente espalhar essa carga sobre **multiplas máquinas**

- **Fault Tolerance / High Availability**
 - Se a aplicação **precisar de trabalhar continuamente** mesmo que uma (ou várias) máquinas se fodam, podemos usar multiplas máquinas para fornecer **redundancia**. Quando uma falha, outra toma o seu lugar
- **Latency**
 - Aplicações são, por natureza, distribuidas.
 - Se temos **users espalhados** por todo o **mundo**, ter **servidores em várias localizações** pode ser fortuito visto que os users vão poder ser **servidos** por um datacenter mais **geograficamente perto** deles

II. Parallel Processing

Processamento Paralelo é uma tecnica de computação que corresponde a **dividir um problema grande em varios problemas mais pequenos** que podem ser **resolvidos em paralelo**

Ex:

Processar 1 tB

A 10 mB/s -> **1.2 dias** para iterar sobre todos os dados

1000 x paralelamente -> **1.5 minutes** para iterar sobre todos os dados

Large Scale Parallel Database Systems têm vindo a ganhar popularidade e serem usados para:

- Guardar grandes volumes de dados

- Processar time-consuming decision-support queries
- Fornecer alto throughput para transaction processing

III. O maior problema com as BDs atuais

Atualmente, as **bases de dados** requerem **grande volume de dados** o que implica uso de **discos** e **grandes quantidades de memoria**.

Temos portanto bottlenecks visto que:

Speed(disk) << speed(RAM) << speed(microprocessor)

O maior problema é portanto o I/O Bottleneck

Mas existem soluções para aumentar a **I/O Bandwidth**:

- **Particionamento de dados**
- **Acesso de dados com paralelização**

Moore's law: processor speed growth (with multicore): 50 % per year

DRAM capacity growth: 4 × every three years

Disk throughput: 2 × in the last ten years

3.1 – Previsões de crescimento de capacidade de processamento

IV. Bases de Dados Paralelas

Bases de dados paralelas conseguem **melhorar a velocidade de processamento** e de **I/O** através do uso de **multiplos CPUs** e **discos** em **paralelo**.

Os dados podem ser **particionados** sobre vários discos e cada **processador** pode **trabalhar independentemente** na sua **partição**

Basicamente é a ideia de utilizar paralelismo em data management para conseguir fornecer **high performance, availability e extensability**.

Vamos então conseguir suportar bases de dados muito grandes com muitas cargas.

As **queries** vão poder ser corridas em **paralelo**

Vai ser, porém, necessário **controlo de concorrência** para tratar de conflitos

Ao desenhar uma base de dados paralela devemos considerar:

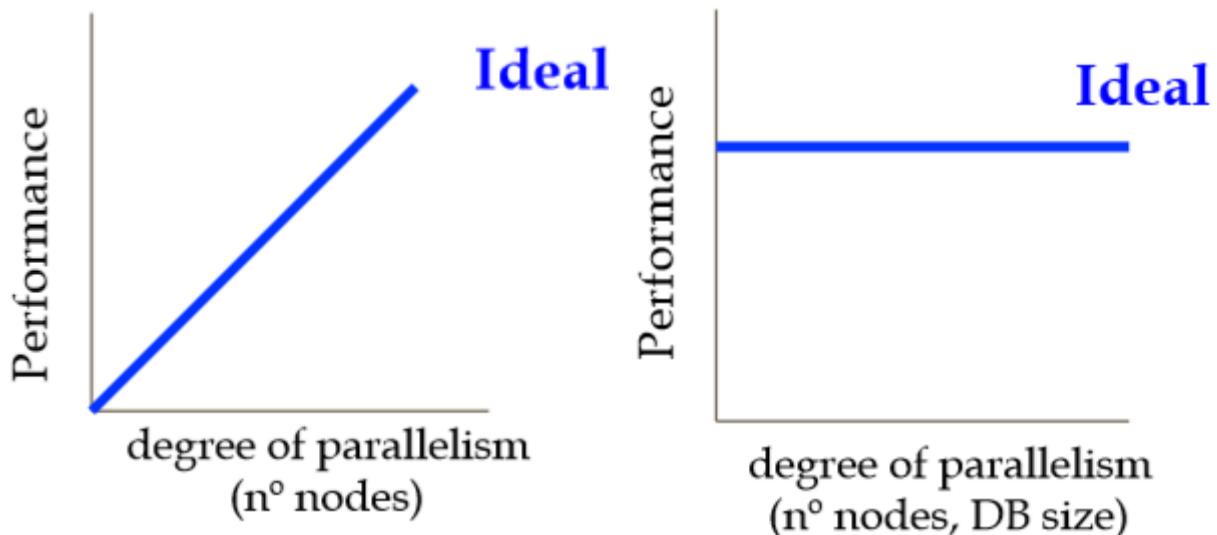
- Data Placement
- Parallel Query Processing
- Load Balancing

A maioria da research que tem sido feita tem apontado para o context do modelo relacional que fornece uma boa base para data-based parallelism – Operações individuais relacionais (sort, join, aggregation, ...) podem ser executadas em paralelo!

Os objetivos principais da Parallel DBMS são:

- **High Performance** através de **parallelization** de várias operations
 - **High Throughput** com Inter-Query Parallelism
 - **Low Response Time** com Intra-Operation Parallelism
 - **Load Balancing** com a habilidade de sistemas para dividir uma dada workload de forma igual sobre todos os processadores

- **High Availability** exploiting **Data Replication**
 - **High Throughput** com Inter-Query Parallelism
 - **Low Response Time** com Intra-Operation Parallelism
 - **Load Balancing** com a habilidade de sistemas para dividir uma dada workload de forma igual sobre todos os processadores
- **Extensibility** com os goals ideais
 - **Linear speed-up**
 - Refere-se ao incremento linear em performance para um constant database size enquanto que o número de nodes (i.e processing e storage power) são aumentados linearmente
 - **Linear scale-up**
 - Refere-se ao sustained performance para um aumento linear em termos de database size e numero de nodes
 - Se os resources aumentarem em porporção com o aumento do data size, o tempo é constante



4.1 – Linear Speed Up e Scale Up

V. Bases de Dados Paralelas VS Distribuidas

Os **principios basicos** das **Parallel DBMS** são os mesmos que as **Distributed DBMS**, porém as **tecnicas** para parallel database systems são **diferentes**

Tipicamente temos:

Parallel DB

- ❖ Fast interconnect
- ❖ Homogeneous software
- ❖ High performance is goal
- ❖ Transparency is goal

Distributed DB

- ❖ Geographically distributed
- ❖ Data sharing is goal (may run into heterogeneity, autonomy)
- ❖ Disconnected operation possible

5.1 – Diferenças entre Parallel e Distributed DBs

É de notar que **Processamento distribuido** normalmente utiliza **parallel processing** (mas não vice versa)

Podemos ter parallel processing numa única máquina

Por norma são feitas as seguintes assumptions:

Parallel Databases

- Machines are physically close to each other (e.g. same server room)
- Machines connects with dedicated high-speed LANs and switches
- Communication cost is assumed to be small
- Architecture: can be **shared-memory, shared-disk or shared-nothing**

Distributed Databases

- Machines can be in distinct geographic locations
- And connected using public-purpose network, e.g., Internet
- Communication cost and problems cannot be ignored
- Architecture: usually **shared-nothing**

5.2 – Assumptions sobre Parallel e Distributed DBs

VI. Barreiras do Paralelismo

- **Startup**
 - O tempo necessário para começar uma operação paralela pode dominar o atual tempo de computação
- **Interference**
 - Quando acedemos a recursos partilhados, cada novo processo atrasa os outros – Hot Spot Problem
- **Skew**
 - O tempo de resposta de um set de parallel processes é o tempo do processo mais lento

Técnicas de parallel data management foram desenhadas para ultrapassar estas barreiras

VII. Parallel & Distributed DBMS Techniques

- **Data Placement**
 - Physical placement da BD em vários nodes
 - Static vs Dynamic
- **Parallel Data Processing Algorithms**
 - Select é fácil
 - Join (e outras non-select operations) são mais fóridas
- **Parallel Query Optimization**
 - Escolha dos melhores parallel execution plans
 - Paralelização automática de queries e load balancing
- **Distributed Transaction Management**

VIII. Database Architectures

Precisamos de arquiteturas que sejam escaláveis para higher loads.

Temos dois tipos de arquiteturas

- **MultiProcessor Architecture**
 - Shared Memory – SM
 - Shared Disk – SD
 - Shared Nothing – SN
- **Hybrid Architectures**
 - Non-Uniform Memory Architecture – NUMA
 - Cluster

Vamos agora ve-las em maior detalhe

VIII.I Shared Memory

É a abordagem mais simples – **Comprar uma máquina mais poderosa**

Também conhecido por **Vertical Scaling** ou **Scaling Up**

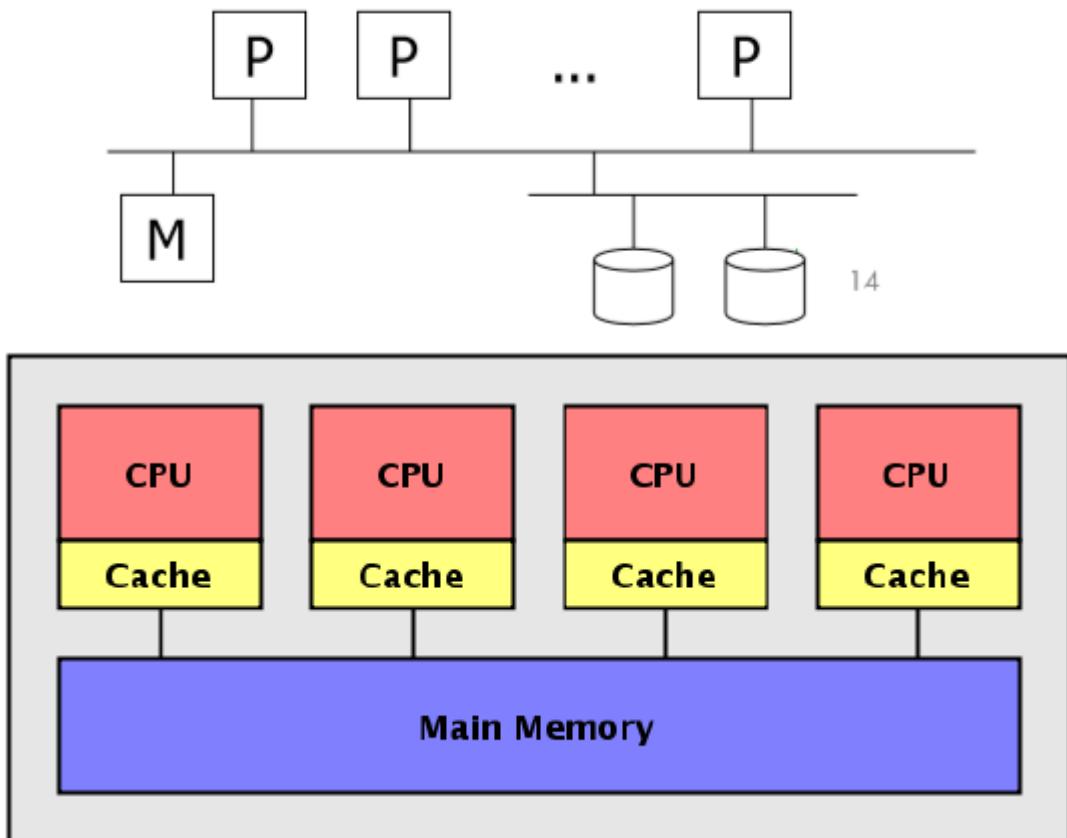
Multiplos processadores partilham o mesmo main memory (RAM) space, mas cada processador tem o seu próprio disco (HDD).

São fornecidas comunicações sobre todos os processadores e evita copias redundantes

Bottlenecks:

- O custo é super-linear
 - Uma maquina com o dobro dos recursos (CPU,RAM, disk) tipicamente custa o dobro do preço

- Uma máquina com o dobro do tamanho não implica que seja capaz de lidar com o dobro da carga
- Oferece fault tolerance muito limitada



8.1.1 – Diagramas de Shared Memory

VIII.II Shared Disk

Usa várias máquinas com CPUs e RAM independentes mas guarda os dados num array de discos que é partilhado entre as máquinas via uma rede rápida

Usado para Data Warehousing Workloads

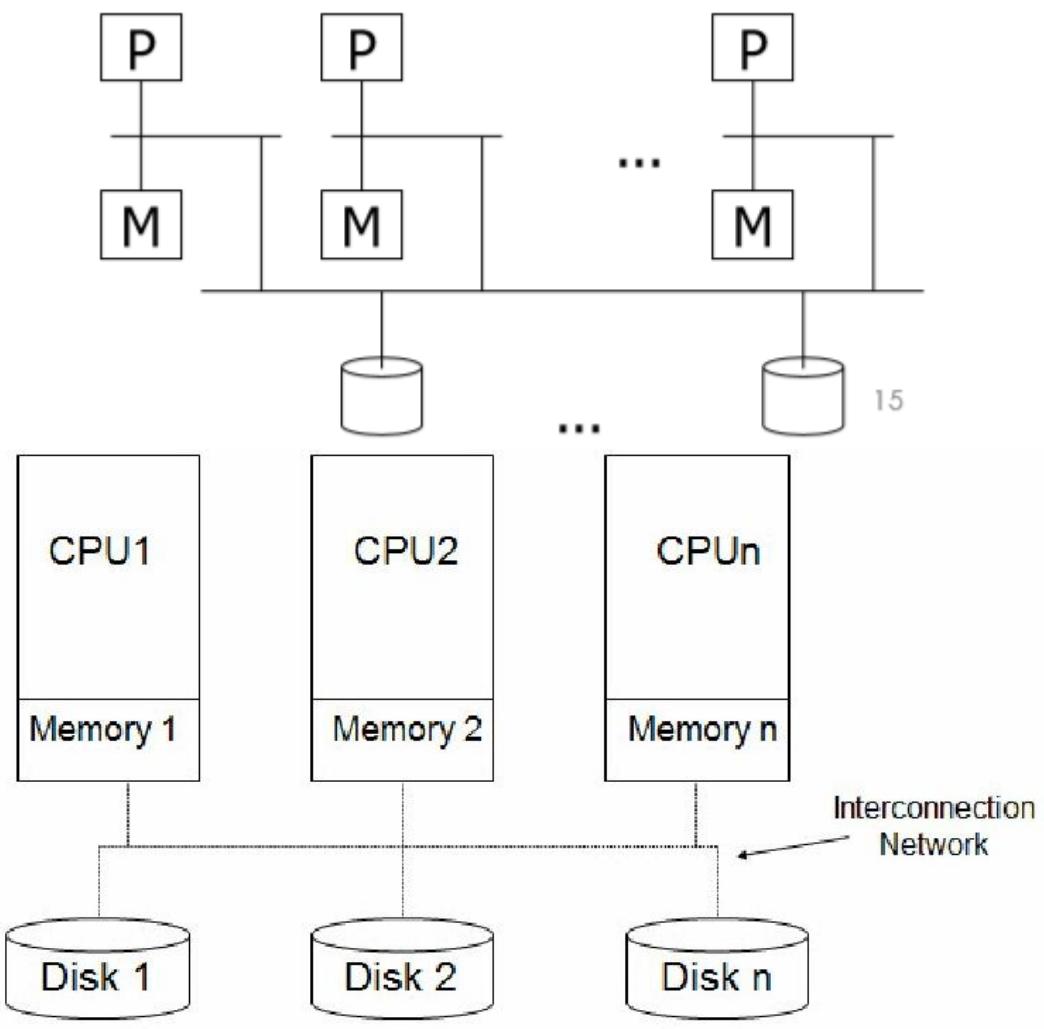
Vantagens sobre Shared Memory:

- Cada processador tem a sua propria memória (não é um bottleneck)

- É uma forma simples de fornecer um certo grau de fault tolerance

Bottlenecks:

- Escalabilidade limitada



Img 8.2.1 – Diagramas de Shared Disk

VIII.III Shared Nothing

Cada máquina ou máquina virtual que esteja a correr o Database Software é chamada de Node e usa CPUs, RAM e Discos que trabalham de forma independente

Também se chama da **Horizontal Scaling** ou **Scaling Out**

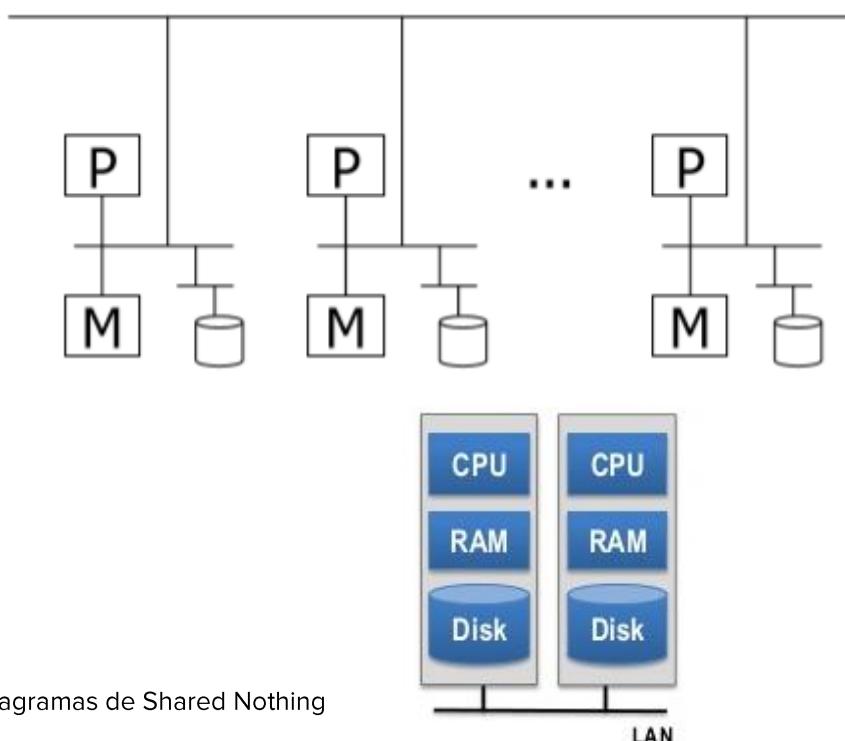
Qualquer coordenação entre nodes é feita ao nível do software usando uma network convencional

Vantagens

- Melhor Preço/Performance ratio
- Extensibility
- Availability
- Reduz Latencia

Desvantagens

- Complexidade
- Dificuldade de load balancing



Img 8.3. 1 – Diagramas de Shared Nothing

VIII.IV Arquiteturas Híbridas

Antes de vermos as próximas duas arquiteturas convém discutirmos o que são arquiteturas híbridas.

Existem várias possíveis **combinations** das três arquiteturas básicas que acabamos de apresentar que apresentam possíveis **trade-offs** diferentes entre **custo, performance, extensibility availability, etc...**

Arquiteturas Híbridas tentam obter as **vantagens de diferentes arquiteturas:**

- **Eficiencia e Simplicidade das Shared-Memory**
- **Extensibility e Custo do Shared Disk ou Shared Nothing**

Temos então **Numa** e **Clusters**

VIII.V NUMA e CC-NUMA

Shared Memory vs Distributed Memory

Mistura dois diferentes aspectos:

- **Addressing**
 - Single Address space e multiplos address spaces
- **Physical Memory**
 - Central e Distributed

A **NUMA** usa um único address space distribuido em memória física

- Facilita application portability
- Extensibility

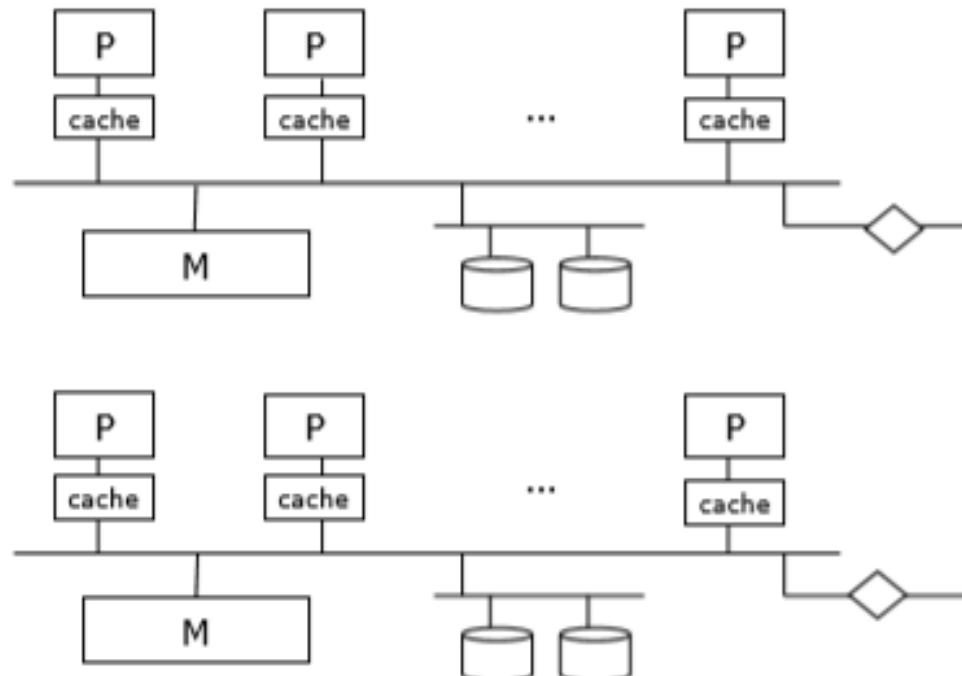
A aplicação desta arquitetura mais bem sucedida é a **Cache Coherent NUMA** (CC-NUMA)

O princípio da CC-NUMA é:

- Main Memory distribuída como Shared Nothing
- MAS, qualquer processador tem acesso às memórias de todos os outros processadores
 - Remote memory access muito eficiente com apenas alguns tempos (tipicamente 2-3) de custo a acesso local

Diferentes processos conseguem aceder aos mesmos dados num conflicting update node

Global Cache Consistency Protocols são necessários



Img 8.5. 1 – Diagrama CC-NUMA

IX. Distributed Data Storage

Temos duas formas de distribuir dados pelos nodes:

- **Replication**

- Mantem um acopia dos mesmos dados em varios diferentes nodes, potencialmente em sitios diferentes
- Fornece redundancia – se alguns nodes não estiverem disponíveis, dados podem ser servidos na mesma através dos nodes restantes
- Também pode aumentar performance

- **Partitioning**

- Dividir uma BD grande em vários subsets mais pequenos chamados **partitions**
- Diferentes partitions podem ser atribuidas a diferentes nodes

Estas duas tecnicas não são mutalmente exclusivas – podemos ter replication e partitioning combinados

X. Data Transparency

Define-se como sendo o grau de abstração apresentado ao user, relativamente aos detalhes de como e onde é que os data items estão guardados num distributed system

Devemos considerar transparency issues em relação a:

- **Mecanismo de Replicação**
- **Mecanismo de Particionamento**
- **Location**

XI. I/O Parallelism

Reduz o tempo necessário para ir buscar relações de um disco através de partitioning

Horizontal Partitioning – tuplos de uma relação estão divididos sobre múltiplos discos

Partitioning Techniques (número de discos = n):

- **Round Robin**
 - Manda o i^{th} tuple inserido numa relação para o disco
 - $i \bmod n$
- **Hash Partitioning**
 - Aplica uma hash function a um ou mais attributes que vão de 0 até $n-1$
- **Range Partitioning**
 - Associa um range de key attribute(s) a cada partition

XII. Query Parallelism

XII.I InterQuery Parallelism

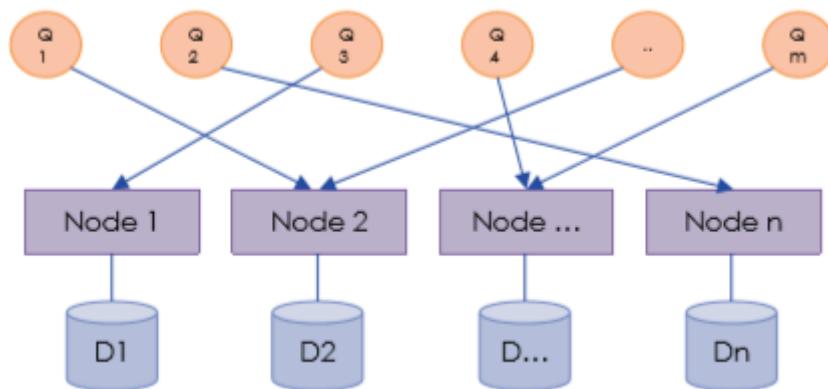
Execução paralela de múltiplas queries geradas por transações concurrentes

Usado para aumentar o **transactional throughput** – Usado principalmente para fazer **scale up** de um transaction processing system para suportar um grande número de transactions por segundo

Forma mais fácil de paralelismo para suportar numa **shared-memory parallel database**

Mais complicado em **shared-disk** ou **shared-nothing**:

- Locking e Logging tem de ser coordenados passando mensagens entre os processadores
- Data num local buffer pode ser atualizada por outro processador
- Cache-Coherence tem de ser mantida – Read e Writes de dados num buffer têm de encontrar a latest version dos dados

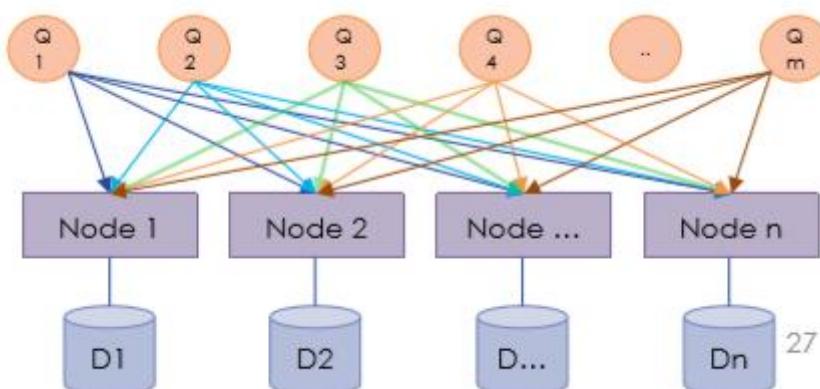


Img 12.1. 1 – Diagrama que mostra InterQuery Parallelism

XII.I IntraQuery Parallelism

Execução de uma unica query em paralelo em multiplos processadores/disks

O mesmo operador é executado por muitos processadores, cada um a trabalhar num subset de dados



Img 12.2. 1 – Diagrama que mostra IntraQuery Parallelism

Temos duas formas complementares de IntraQuery Parallelism:

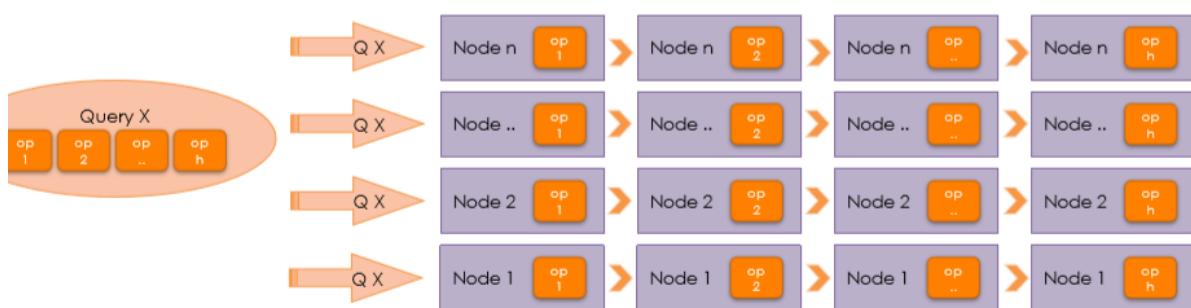
- **Intra-Operation**

- Paralelizar a execução de cada operação individual na query
- Escala melhor com increasing parallelism porque o numero de tuplos processados por cada operação é tipicamente maior do que o numero de operações numa query

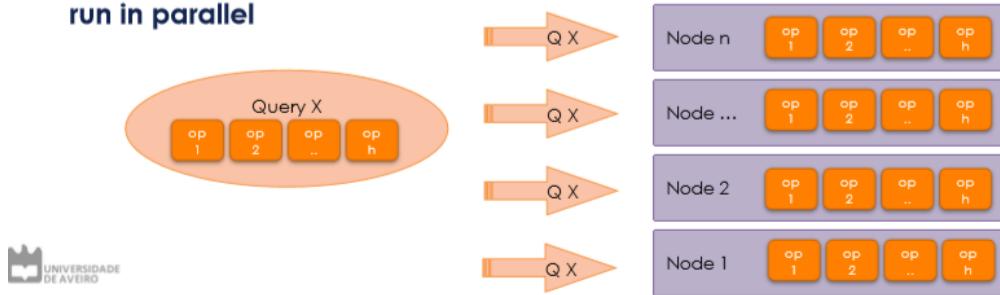
- **Inter-Operation**

- Executar diferentes operações de uma query expression em paralelo

❖ **Inter-operator (Pipeline):** ordered (or partially ordered) tasks and different machines are performing different tasks



❖ **Intra-operator (Partitioned):** a task divided over all machines to run in parallel



Img 12.2.2 – Inter-operator vs Intra-operator

XIII. Parallel Data Processing

Nesta secção vamos assumir que temos:

- Read only queries
- Uma shared-nothing architecture (consegue eficientemente ser simulada numa shared-memory ou shared-disk system)
- n processadores (P_0, \dots, P_{n-1}) e n discos (D_0, \dots, D_{n-1}) onde cada disco D_i está associado ao processador P_i , se um processador tiver múltiplos discos (pode ser facilmente simulado mesmo que um processador tenha muitos discos)

Vamos ver **Parallel Algorithms** que são usados no Relational Model

Algoritmos paralelos para relational algebra operators são os building blocks necessários para qualquer parallel query processing

Parallel data processing deve dar exploit de **intra-operator parallelism** (a query é paralelizada e os nodes realizam todas operações pedidas pela query)

Vamos-nos focar em **sort, select e join operators** (se bem que outros operadores binários tal como **union** podem ser tratados de formas parecidas)

XIII.I Parallel Selection - $\sigma_c(R)$

A relação R é particionada sobre m máquinas

Cada partição de R é cerca de $|R|/m$ tuplos

Cada máquina faz scan na sua própria partição e aplica a **Selection condition** de c

Data Partitioning – Impact

- **Round Robin** ou **Hash Function** (sobre o mesmo tuple)
 - Relação é esperada que seja bem distribuída sobre vários todos os nodes

- Todas as partições vão ser scanned
- **Range ou Hash-Based** (on the selection column)
 - Relações podem ser clustered em poucos nodes
 - Poucas partições têm de ser tocadas

Parallel **Projection** também é straightforward

- Todas as partições vão ser tocadas
- Não é sensível no que toca a como os dados estão particionados

XIII.II Parallel Sorting

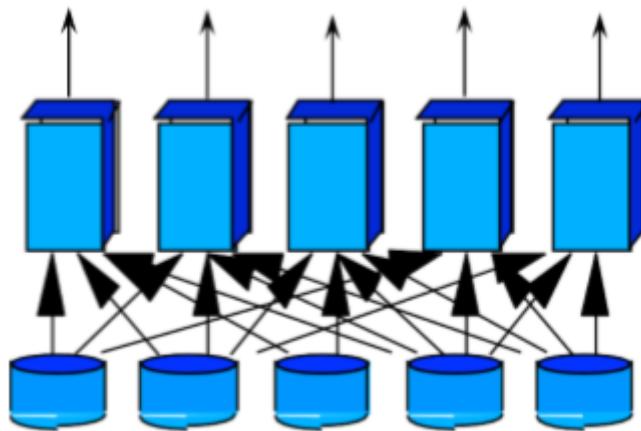
Temos dois tipos:

Range-Partitioning Sort

1. Escolhe processadores P0, ..., Pm onde m <= n-1 para realizarem o sort
2. Re-particionar R baseado nos ranges (nos sorting attributes) em m partições
 - a. Este passo requer I/O e communication overhead
3. A máquina i recebe todas a ith partitions de todas as máquinas que dão sort da partition sem interagir com as outras
 - a. Pi guarda os tuplos que recebe temporariamente no disco Di
4. **Final Merge** operations são triviais – Range Partitioning assegura que para i, j, m os key values num processador Pi são todos menores que os da key values em Pj

Tem o problema da **Skewed Data**:

- Ranges podem ter width diferentes
- Aplicar sampling phase primeiro



Img 13.2.1 – Range-Partitioning Sort

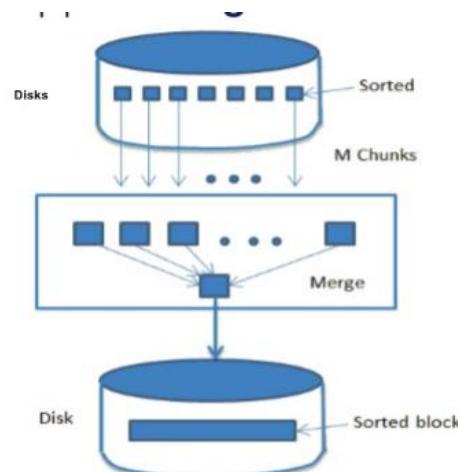
Parallel External Sort-Merge

Assume que a relação já foi particionada pelos discos D0, ..., Dn-1 (de qualquer forma)

Cada node organiza os seus próprios dados

Todos os nodes começam a mandar os seus sorted data (um bloco de cada vez) para uma unica maquina

Esta maquina aplica **merge-sort techniques** a medida que os dados chegam



Img 13.2.2 – Parallel Sorting

XIII.III Parallel Join

A JOIN operation requer que **pares de tuplos** sejam **testados** para vermos se satisfazem a **join condition**

Se tuplos satisfazem a **Join Condition** o par é adicionado aos outputs do Join

Em geral temos os passos:

1. **Parallel Join Algorithms** tentam dar **split** dos **pares** que estão a ser **testados** sobre varios processadores
2. **Cada processador computa** parte do join **localmente**
3. **Resultados** de cada processador são **colecionados** juntos para **produzir** um **ultimo resultado**

Teos 3 Basic Parallel Join Algorithms para bases de dados particionadas:

- **Parallel Nested Loop** (PNL)
- **Parallel Associative Join** (PAJ)
- **Parallel Hash Join** (PHJ)

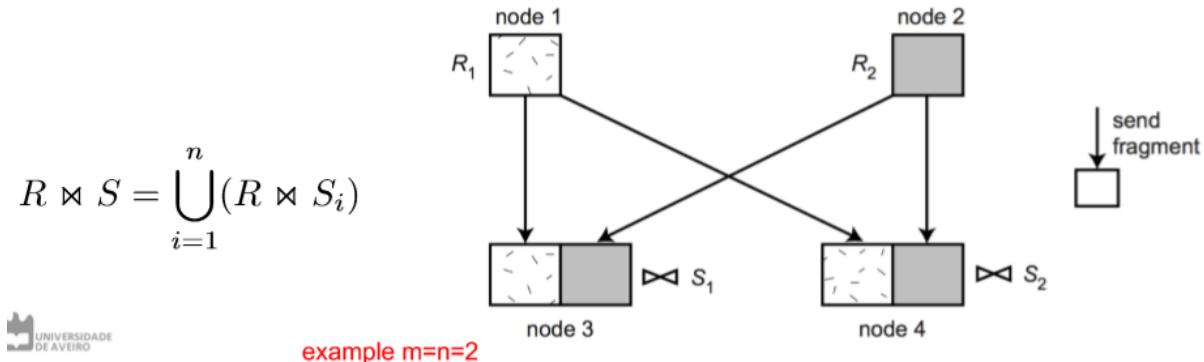
Todos estes algoritmos são **intra-operator parallelism**

Tambem são aplicaveis a outros operators complexos tais como duplicate elimination, union, intersection, ..., com poucas adaptações

As seguintes imagens ilustram estes 3 algoritmos utilizando o seguinte exemplo.

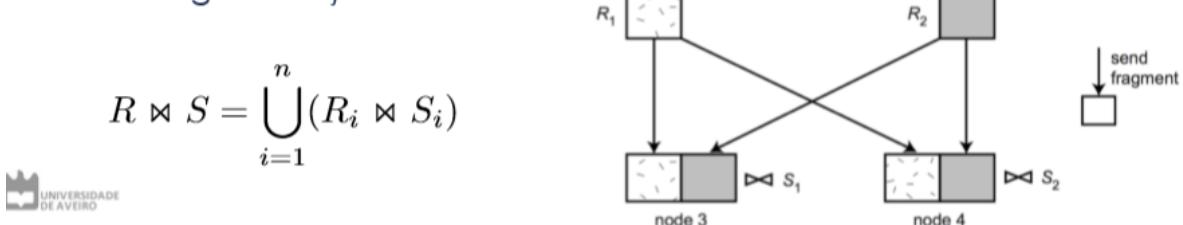
Join de duas relações R e S que estão particionadas em m e n nodes respetivamente

- ❖ **Cartesian product** of relations **R** and **S**, in parallel.
- ❖ **Simplest** and most **general** method
- ❖ Algorithm phases:
 - each fragment (replica) of R is sent to each node containing a fragment of S** (there are n such nodes)
 - this phase is done in parallel by m nodes
 - each S-node j receives relation R entirely, and locally joins R with fragment S_j .**
 - join processing may start as soon as data are received



Img 13.3.1 – Parallel Nested Loop Join

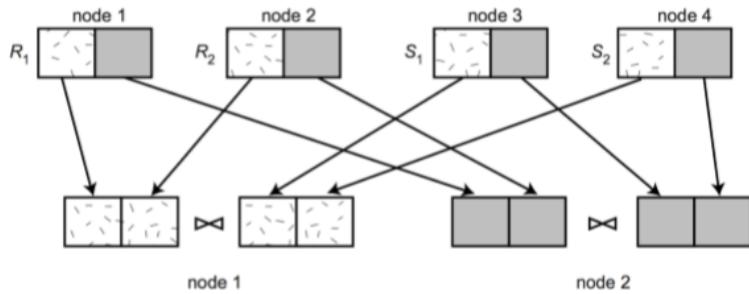
- ❖ Applies **only to equijoin** with **one** of the **operand relations partitioned** according to the **join attribute**
- ❖ Assume
 - **equijoin** predicate is on **attribute A** from **R**, and **B** from **S**
 - **S** is **partitioned** according to **hash function applied** to attribute **B**
 - tuples of **S** that have the same $h(B)$ value are placed at the same node
 - **no knowledge** of how **R** is **partitioned**
- ❖ Algorithm phases:
 1. relation **R** is **sent** associatively to the **S-nodes based** on the **hash function** h applied to **attribute A**
 2. **each S-node j receives** in parallel from the different **R-nodes** the relevant subset of **R** (i.e., R_j) and **joins it locally** with the fragments S_j



Img 13.3.2 – Parallel Associative Join

- ❖ Generalization of parallel associative join algorithm
- ❖ Also applies to equijoin but does not require any particular partitioning of the operand relations
- ❖ **Basic idea:** partition of R and S into the **same number p** of mutually exclusive sets (**fragments**) R_1, R_2, \dots, R_p , and S_1, S_2, \dots, S_p ,
- ❖ The **p nodes** may actually be **selected at run time** based on the load of the system
- ❖ Algorithm phases:
 1. **build:** **hashes R** on the **join attribute**, **sends it to the target p nodes** that build a hash table for the incoming tuples
 2. **probe:** **sends S associatively** to the **target p nodes** that probe the hash table for each incoming tuple

$$R \bowtie S = \bigcup_{i=1}^p (R_i \bowtie S_i)$$



Img 13.3.3 – Parallel Hash Join

XIV. Parallel Processing Costs

Join Processing é atingido com um grau de paralelismo que é ou n ou p .

Cada algoritmo requer movermos pelo menos uma das relações no operando

O cenário ideal é aquele em que não ha skew no particionamento nem overhead devido à parallel evaluation

Expected speed-up: $1/n$

Porem devemos ter em consideração Skew e Overheads:

- time taken by a parallel operation can be estimated as:

$$T_{cost} = T_{part} + T_{asm} + \max(T_0, T_1, \dots, T_{n-1})$$

T_{part} - time for partitioning the relations (including communications costs)

T_{asm} - time for assembling the results

T_i - time taken for the operation at processor P_i . This needs to be estimated taking into account the skew, and the time wasted in contentions

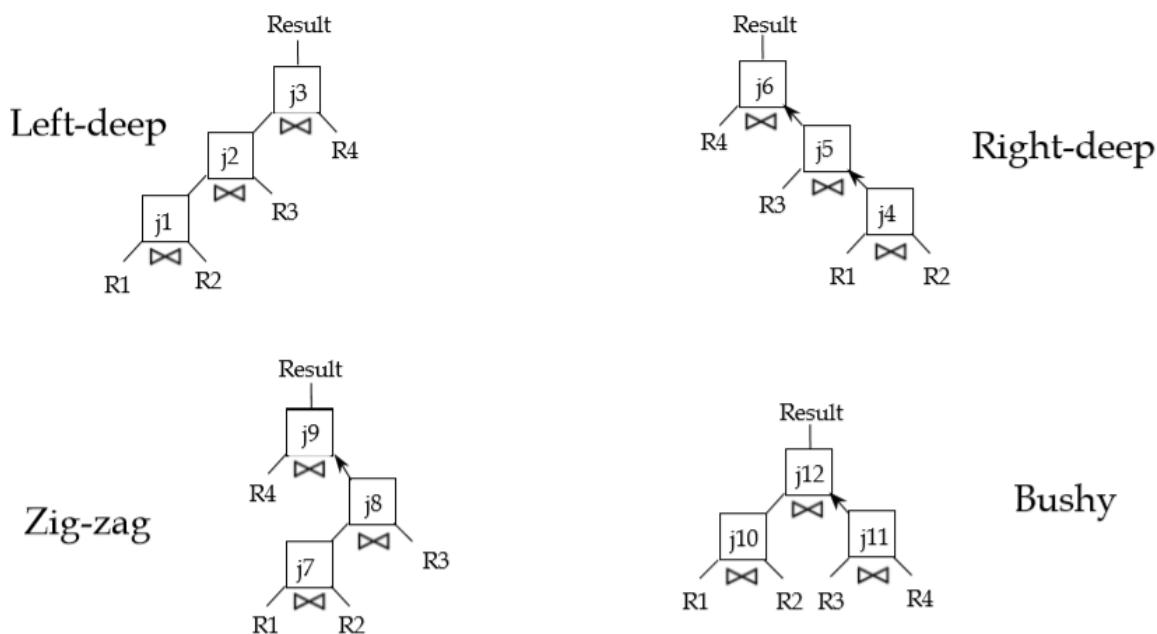
Img 14.1 – Estimativa de tempo que uma parallel operation demora

XV. Parallel Query Optimization

O objetivo de Query Optimization é selecionar o “melhor” possível parallel execution plan para uma query considerando os seguintes componentes:

- **Search Space**

- Modela alternative execution plans como operator trees
- Left deep vs Right deep vs Bushy trees



Img 15.1 – Search Space Operator Trees

- **Search Strategy**

- Programação dinâmica para um pequeno search space
- Randomized para large search spaces

- **Cost Model** (Abstraction do Sistema de execução)
 - Physical schema info (partitioning, indexes, ...)
 - Estatísticas e cost functions

O objetivo é **minimizar a movimentação de dados entre máquinas**

❖ Even better execution plan:

```

1. On M2 compute
   INSERT INTO TEMP1 SELECT DISTINCT D FROM S;

2. Copy TEMP1 to M1

3. On M1 compute
   INSERT INTO TEMP2
   SELECT A, B FROM R join TEMP1 on B = D;

4. Copy TEMP2 to M2    R × S

5. On M2 compute
   INSERT INTO ANSWER
   SELECT A, C FROM TEMP2 join S on B = D;
  
```

– TEMP2 is **left semijoin** of R and S

- ❖ Very Good if TEMP1 and TEMP2 are relatively small
- ❖ Two Machines
 - M1 has the relation R(A,B)
 - M2 has the relation S(C,D)
- ❖ Query


```
SELECT A, C FROM R join S on B = D;
```
- ❖ Result
 - must be at M2

➤ Options:

1. Copy S to M1
2. Compute the result
3. Send the result to M2

OR

1. Copy R to M2
2. Compute the result

➤ Scenarios:

size R ≈ S

?

size R > S

Img 15.2 – Best Execution Plan Example

XVI. Load Balancing

Consiste em balançar a carga de diferentes transactions e queries por diferentes nodes.

É essencial para maximizar throughput

Problemas aparecem para intra-operator parallelism com skewed data distributions:

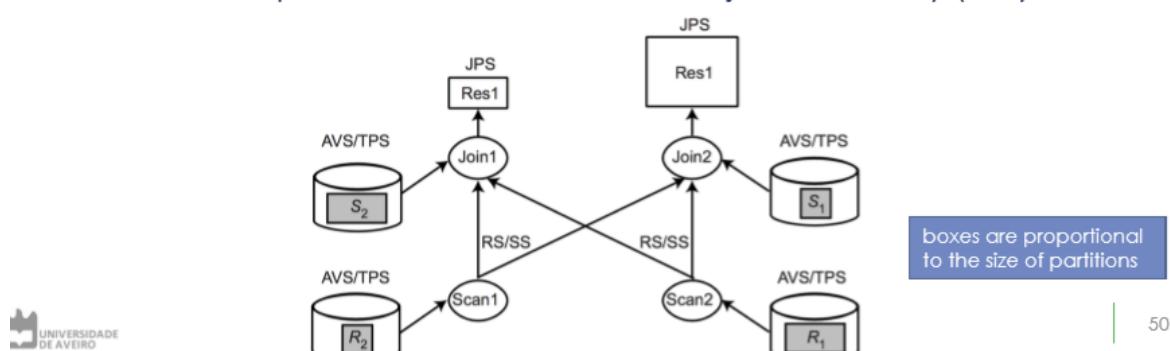
- **attribute data skew (AVS)**
 - inherent to dataset (e.g., there are more citizens in Paris than in Aveiro).
- **tuple placement skew (TPS)**
 - introduced when the data are initially partitioned (e.g., with range partitioning)
- **selectivity skew (SS)**
 - introduced when there is variation in the selectivity of select predicates on each node
- **redistribution skew (RS)**
 - occurs in the redistribution step between two operators (similar to TPS)
- **join product skew (JPS)**
 - occurs because the join selectivity may vary between nodes

Img 16.1 – Skewed Data Distributions

Soluções para estes problemas são:

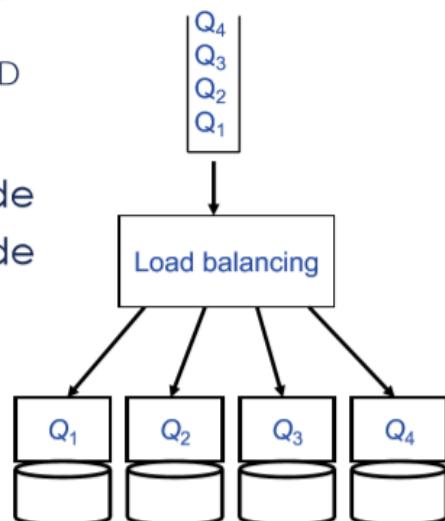
- **Algoritmos de paralelismo sofisticados capazes de lidar com skew**
- **Alocação dinâmica de processadores (durante execution time)**

- ❖ A **query** over two relations **R** and **S** that are **poorly partitioned**
 - due to either data (AVS) or the partitioning function (TPS)
 - processing times of instances (scan1 and scan2) are not equal
- ❖ **Join** operator case is **worse**
 - the number of tuples received is different due to the poor redistribution of the partitions (RS) or variable selectivity according to the partition of R processed (SS)
 - uneven size of S partitions (AVS/TPS) yields different processing times for tuples sent by scan operator. The result size is different from one partition to the other due to join selectivity (JPS)



Img 16.2 – Exemplo de Data Skew

- ❖ Choose the node to execute Q
 - round robin
 - the least loaded
 - Need to get load information
- ❖ Failover
 - In case a node N fails, N 's queries are taken over by another node
 - requires a copy of N 's data or SD
- ❖ In case of interference
 - data of an overloaded node are replicated to another node



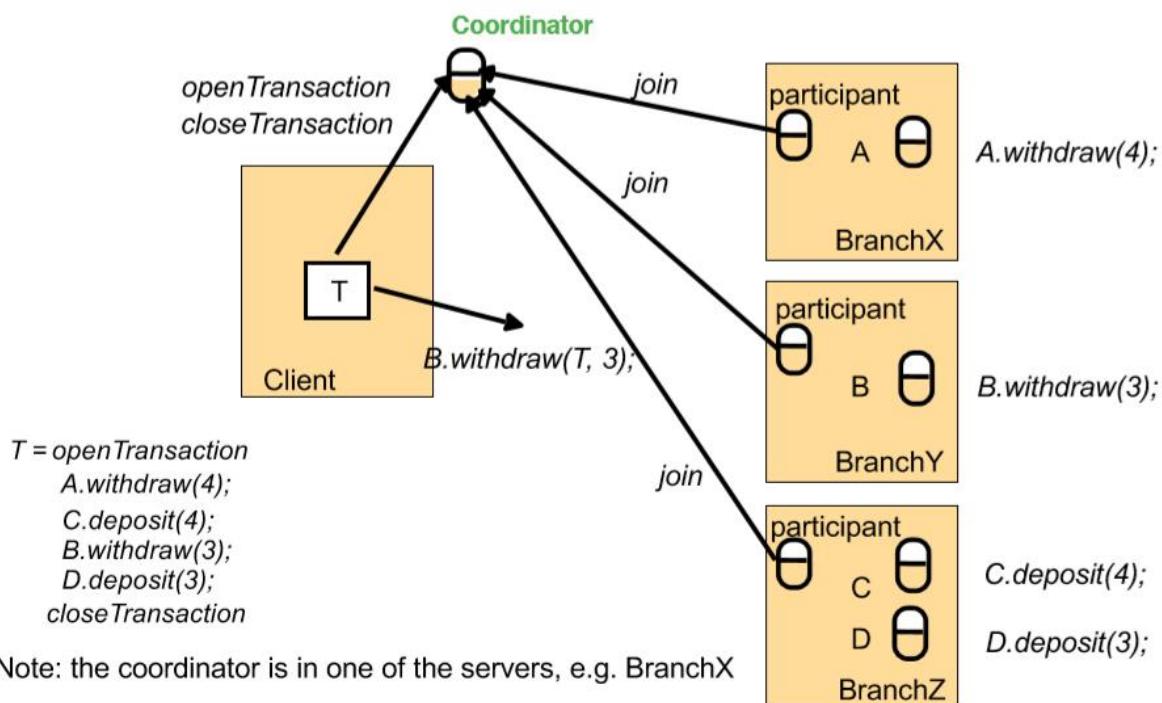
Img 16.3 – Load Balancing numa DB Cluster

XVII. Distributed Transactions

Transactions podem aceder a dados de diversos sitios

Cada sitio tem um **local transaction manager** responsavel por:

- Manter um log para motivos de recuperação
- Participar na coordenação da atual execução de transactions que estão a ser executadas no site de que está responsavel



Img 16.4 – Distributed Banking Transaction

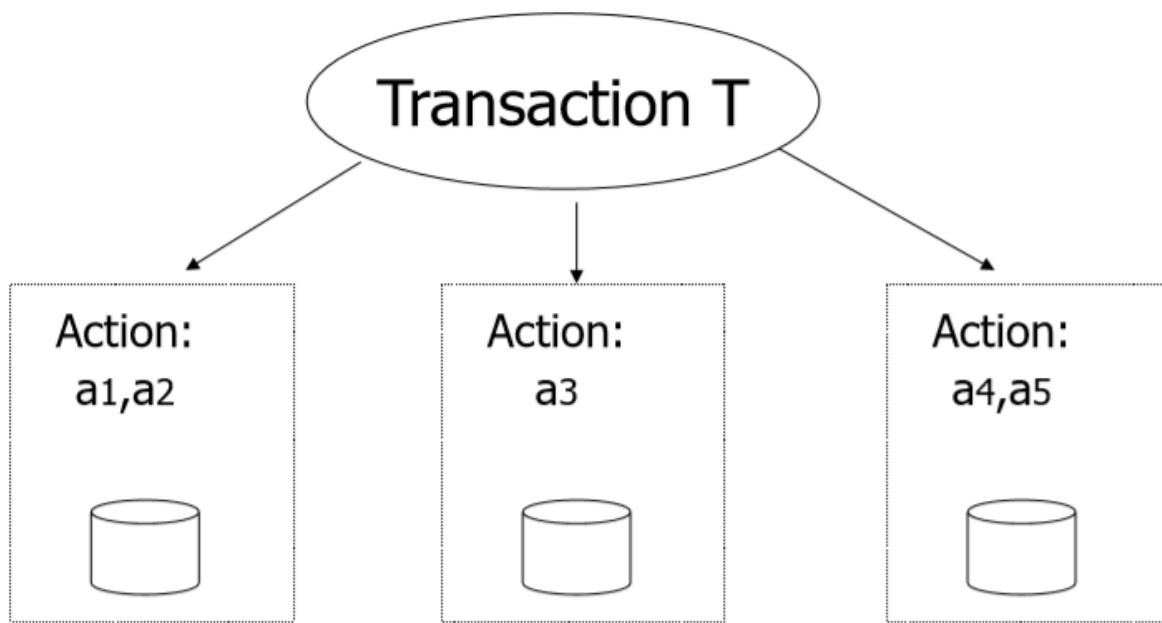
XVIII. Distributed Commit Problem

Os Commits devem ser atómicos.

Como é que transações distribuidas tem componentes em vários sitios e conseguem executar de forma atómica na mesma?

Através de Two-Phase Commits

Exemplos incluem Centralized 2PC, Distributed 2PC, Linear 2PC, ...



XIX. Two-Phase Commit Protocol

Temos duas fases:

- **Primeira fase – coordenador coleciona um voto (commit ou abort) de cada participante**
 - Participantes guardam os resultados parciais num storage permanente antes de votarem
- **Segunda fase – coordenador toma uma decisão**
 - Se todos os participantes quiserem fazer commit e nenhum crashou, o coordenador faz multicast de uma mensagem de “commit”
 - Toda a gente faz os commits
 - Se um participante falhar, então durante a recovery ele consegue receber a commit message do coordinator
 - Else, se todos os participantes crasharem ou abortarem o coordenador faz multicast de uma abort message para todos os participantes
 - Toda a gente aborta

Replicação de Dados Distribuídos

I. Replicação

Replicação significa manter uma **cópia de dados exatamente iguais** em **multiplas máquinas** que estejam ligadas por uma **network**

A Replicação é feita para:

- **Reducir Latencia**
 - Manter dados geograficamente perto dos users
- **Aumentar Availability**
 - Permitir que um sistema continue a trabalhar mesmo que algumas partes falhem
- **Scalability**
 - Para aumentar o numero de máquinas que conseguem servir read queries (aumentando portanto o read throughput)

Vamos assumir que o dataset é pequeno o suficiente tal que cada máquina consiga manter uma cópia do dataset inteiro

Problemas com Replicação começam a ocorrer **quando existe alteração dos dados replicados :(**

Tradeoffs que temos que considerar com replicação incluem:

- **Replicação Síncrona vs Assíncrona**
- **Como é que são tratadas falhas das replica**

Os algoritmos mais comuns para replicar mudanças entre nodes são:

- **Single Leader**
- **Multi Leader**
- **Leaderless**

II. Single Leader Replication

II.I Leaders e Followers

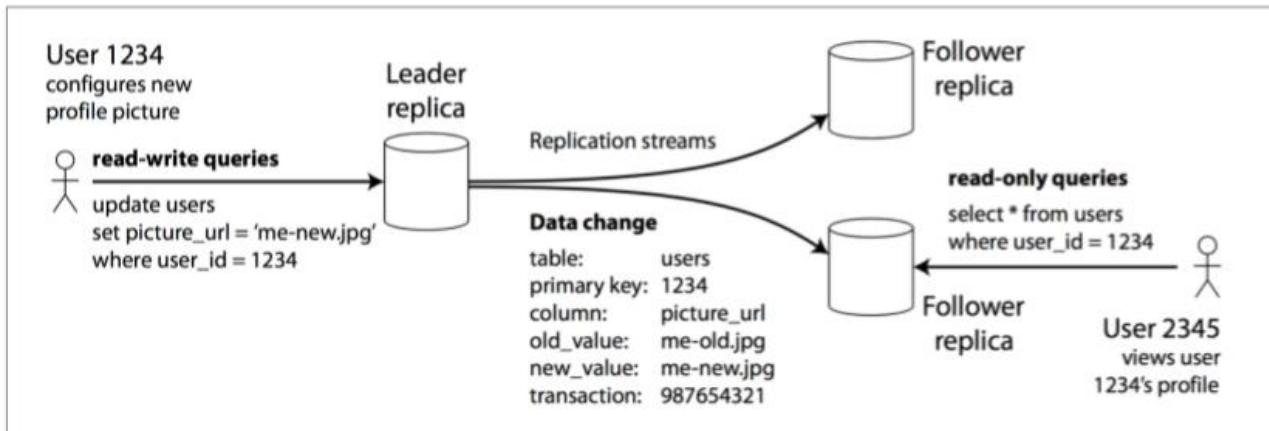
Uma **Replica** é qualquer node que guarde uma cópia da BD

Todas as escritas para a base de dados têm de ser processadas por todas as réplicas

Em **Leader Based Replication** (ou active/passive, master-slave replication) uma das réplicas é designada de **Líder** (ou Master/Primary), enquanto que as outras chamam-se **Followers** (ou read replicas, slaves, hot standbys, filhos da puta, etc)

É a solução mais comum para data replication

II.II Funcionamento



Img 2.2.1 – Exemplo de Single Leader replication

O algoritmo Single Leader funciona da seguinte forma:

1. Cliente realiza uma write query na BD. Esta é enviada para o Líder
2. Líder escreve os novos dados no seu local storage
3. Leader manda os dados que foram alterados para os followers
 - a. Utilizando replication log ou change stream
4. Cada follower pega no log recebido do líder e atualiza a sua própria cópia local da BD
 - a. Todos os writes são processados na mesma ordem pela qual o líder os processou

No que toca a read queries, o cliente pode ler da BD fazendo a query ao líder ou a qualquer um dos followers

II.III Replicação Síncrona vs Assíncrona

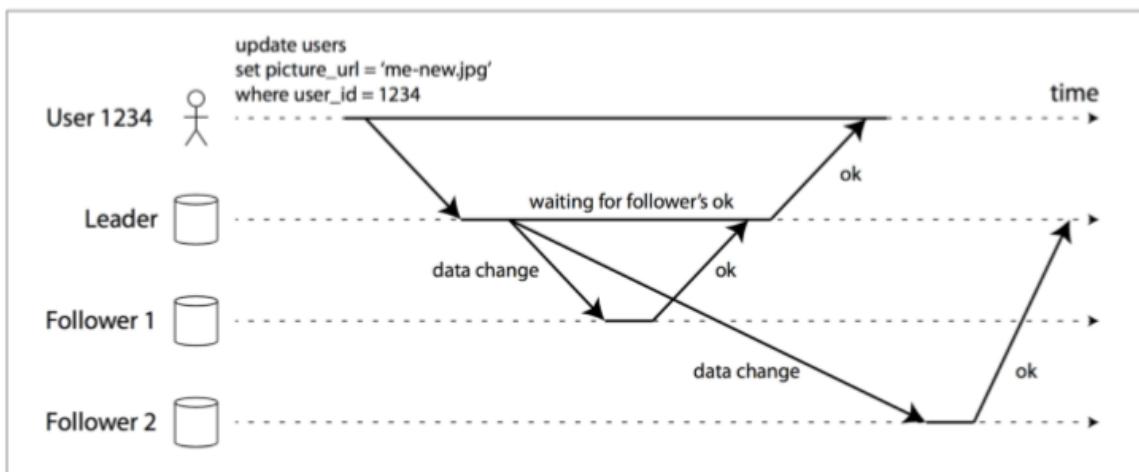
Replicação tende a ser rápida e a maioria dos database systems são capazes de aplicar as mudanças aos followers em menos de um segundo.

Porém não temos garantias de quanto tempo é que a replicação vai demorar.

Existem circunstâncias nas quais os followers podem atrasar-se em relação ao líder por vários minutos ou mais

P.ex se o follower estiver a recuperar de um failure, se o sistema estiver a operar perto de maximum capacity ou se ocorrerem network problems entre os nodes

- ❖ Follower 1 replication is **synchronous**: the leader waits for follower 1 (write) confirmation before reporting success to the user, and before making the write visible to other clients
- ❖ Follower 2 replication is **asynchronous**: the leader sends the message, but doesn't wait for a response from the follower



In relational databases, this is often a configurable option; other systems are often hard-coded to be either one or the other
Img 2.3.1 – Exemplo de Synchronous vs Asynchronous replication

Vantagem de replicação Síncrona

- Os followers tem uma up-to-date copy consistente dos dados
- Se o líder falhar, os dados continuam disponíveis nos dados

Desvantagem de replicação Síncrona

- O Write não pode ser processado se um follower síncrono não responder (p.ex se tiver crashado ou ocorrer algum tipo de network fault)
- O líder é obrigado a **bloquear todos os writes** até que a replica síncrona esteja disponível outra vez
- É impratico ter todos os seguidores síncronos

- Em prática, se ativarmos synchronous replication numa BD, isso significa que apenas um dos followers é sincrono e os outros são assíncronos

Para além de Synchronous Replication, temos também:

- **Semi-Synchronous Configuration**
 - Se um synchronous follower ficar unavailable, ou se tornar demasiado lento, um dos asynchronous followers torna-se sincrono
 - Garante uma up-to-date copy dos dados em pelo menos dois nodes – O Líder e um follower sincrono
- **Fully Asynchronous Configuration**
 - Frequentemente utilizada
 - Fornece a **vantagem** que o líder **consegue continuar a processar writes**, mesmo que todos os seus followers se tenham atrasado
 - Porém, o **write não é garantido que seja durable**, mesmo que tenha sido confirmado pelo cliente
 - Se o líder falhar e não for recuperável, qualquer write que ainda não tenha sido replicado para os followers vai ser completamente perdido

O enfraquecimento da durabilidade de asynchronous replication é quase inevitável quando temos muitos followers ou se estes estiverem geograficamente distribuídos !

II.IV Criar e dar setup de novos Followers

Como é que se configura um novo Follower? Simplesmente copiar todos os data files entre nodes é (tipicamente) insuficiente visto que os clientes estão constantemente a escrever para a BD.

Podíamos prender a BD e não permitir que writes fossem efetuados até que o novo follower estivesse up-to-date, mas isso iria contra do objetivo da Replication de conseguir fornecer High Availability

Temos então o seguinte algoritmo que não requer nenhum downtime:

1. Pegamos num snapshot consistente da base de dados do líder a algum ponto no tempo (a maioria das BDs tem esta feature de fazer snapshots para motivos de backups)
2. Copiamos o snapshot para o novo follower node
3. Conectamos o follower ao líder
4. O novo follower vai pedir ao líder por todas as data changes que foram efetuadas desde que o snapshot que usou para inicializar os dados foi criado
5. Quando o follower acabar de processar os data changes desde o snapshot podemos dizer que o novo follower caught up com os restantes

II.V Node Outages

Qualquer Node no Sistema pode ir abaixo de forma não expectável devido a uma fault, ou de forma planeada para a realização de maintenance

O objetivo é fazermos com que o sistema como um todo continue a correr, mesmo com individual failed ou stopped nodes – Diminuir ao máximo o impacto de um node outage

Em leader-based replication temos que considerar dois tipos de fails para tentarmos alcançar high availability:

Follower Failure

Leader Failure

II.VI Follower Failure – Catchup Recovery

Podem ocorrer duas situações:

- **Follower Crash** e é restarted
- **Ocorrem problemas de Network** (leader <-> follower)

Existe um fácil Follower Recovering process:

- Followers mantem um log de data changes (recebido do líder)
- Log é usado para saber qual foi a ultima transaction antes que a fault tenha ocorrido
- Com isto podemos:
 1. Ligar ao líder e pedir todas as data changes que ocorreram durante o tempo em que o follower esteve desconectado
 2. Aplicar essas mudanças
 3. Continuar a receber um stream de data changes como normalmente (regular operation state)

II.VII Leader Failure - Failover

Tratar o failure de um líder é um bocado mais tricky:

1. Um dos followers deve ser promovido a líder
2. Clientes precisam de ser reconfigurados para passarem a mandar os writes ao novo líder
3. Outros followers precisam de começar a consumir data changes a partir do novo líder

Failover - In [computing](#) and related technologies such as [networking](#), **failover** is switching to a [redundant](#) or [standby](#) [computer server](#), [system](#), hardware component or network upon the failure or [abnormal termination](#) of the previously active [application](#),^[1] server, system, hardware component, or network.

Failovers do líder podem acontecer:

- **Manualmente**
 - Um administrador é notificado que o líder falhou e pode tomar os passos necessarios para criar um novo líder
- **Automáticamente**

Para os Automatic Failovers temos o seguinte processo:

1. Determinar que o líder falhou

- a. Muitas coisas podem potencialmente correr mal
- b. Não existe nenhuma foolproof way de detetar
- c. A maioria dos sistemas usa um simples **timeout** – assume-se que um node morreu se não responder por algum período de tempo
 - i. Não existe nenhuma solução fácil para definir o timeout a partir do qual determinamos se o líder morreu ou não...
 - ii. **Duração mais longa** requer mais tempo para recovery caso o líder tenha mesmo falhado
 - iii. **Duração mais curta** pode detetar unnecessary failovers (p.ex uma temporary load spike pode causar com que o response time de um node seja mais alto do que o normal, ou um network glitch pode atrasar a receção e envio de packets).
 - iv. Se o sistema já tiver sofrido de problemas devido a high load ou network problems, um unnecessary failover vai provavelmente piorar a situação...
 - v. Por esta razão, as operation teams tendem a preferir realizar Failovers Manuais

2. Escolher um novo líder

- a. **Processo de eleição** (o líder é escolhido pela maioria de online replicas)
- b. O melhor candidato para liderança tende a ser a replica mais up-to-date
- c. Um novo líder pode ser appointed pelo previously-elected controller node
- d. Fazer com que todos os nodes estejam de acordo em que líder votar pode causar **consensus problems**

3. Reconfigurar o sistema para usar o novo líder

- a. Clientes precisam de mandar os seus write requests ao novo líder

- b. Vamos ter problemas caso o **antigo líder volte a ficar online**, visto que este pode continuar a achar que é o líder
- c. O sistema tem de garantir que o líder antigo se torna num follower e consegue reconhecer o novo líder

O que pode correr de mal com este processo de recuperação de Failovers?

- **Asynchronous Replication**

- O novo líder pode não ter recebido todos os writes do velho líder
- Se o velho líder der rejoin à cluster apos o novo líder tenha sido escolhido o qe deve acontecer a esses writes?
- O novo líder pode ter recebido **conflicting writes** entretanto
- **Solução** (masi comum):
 - Descartar todas os unreplicated writes do velho líder
 - Isto pode violar as expectativas de durabilidade do cliente porém

- **Split Brain**

- Dois nodes ambos acham que são o líder
- Se ambos os leaders aceitarem writes, e não houver nenhum processo para resolver conflictos, os dados podem ser perdidos ou corrupted
- **Solução – Safely Catch:**
 - Alguns sistemas tem mecanismos para desligar um node se dois líderes forem detetados
 - O problema com isto é que pode acontecer a situação em que accidentalmente desligamos ambos os nodes...

II.VI Replication Methods

Vamos agora ver os replication methos usados em leader-based replication:

- **Statement-Based Replication**

- O líder cria um **log** de **write statements executados** e manda esse **statement log** para os followers
- Cada follower parse e executa o statement como se este tivesse sido recebido diretamente do cliente
- **Problemas:**
 - Statements que chamam funções **não-deterministicas** (p.ex NOW() ou RAND()) vão gerar valores diferentes em cada replica
 - Statements que usem **auto-incrementing column** ou dependam de existing data na BD vão dar merda
 - Statements que tenham **side-effects** (e.g triggers, SP, UDFs) podem resultar na ocorrencia de diferentes side-effects em cada replica, a não ser que os side-effects sejam completamente deterministicos
 - Precisamos de garantir a ordem de execução dos statements em cada node
- **Solução:**
 - O líder pode replicar todas as non-deterministic function calls com um fixed return value quando o statement for logged
 - Desta forma todos os followers vão ter os mesmos valores
- Esta solução porém é um coto merda
- Concluindo, outros replication methos são geralmente preferidos
- O MySQL antigamente usava isto. VoltDB ainda usa e garante a segurança ao obrigar a que as transações sejam deterministas

- **Write-Ahead Log Shipping (WAL)**

- Storage Engines tendem a guardar todos os Writes num Log
- Um Log é um append-only sequence of bytes que contem todos os writes da BD
- A ideia é então usar o **log para construir a replica noutro node**. O lider, para além de escrever o log no disco, manda-o também pela network para os seus followers
- **Desvantagem:**
 - O Log descreve os dados num nível muito baixo
 - Isto torna a replicação closely coupled com o storage engine
 - Dificulta a capacidade de corrermos diferentes versões do database software no lider e nos followers
- Este metodo é usado pelo PostgreSQL e Oracle, entre outros

- **Logical Log Replication**

- Formatos de **Log diferentes** para **replication** e **physical storage**
- Mais **facil** de se **manterem backwards-compatible**
 - O lider e o follower podem correr versões diferentes do software da BD ou usar different storage engines
- Um Logical Log Format é **mais facil de ser parsado** por **aplicações externas** (p.ex data warehouse para offline analysis ou para construir custom indexes e caches)
- Um logical log para relational BDs pode conter, p.ex, informação sobre rows Inseridas, Apagadas e Atualizadas

- **Trigger-Based Replication**
 - **Replicação à camada da aplicação**
 - Fornece mais flexibilidade
 - Exemplos de uso:
 - Replicação de um subset de dados
 - Replicação de um tipo de BD para outro
 - Approaches (ao nível da camada do sistema da BD)
 - Ao ler um database log
 - Usando database features
 - Triggers conseguem dar log de database writes numa tabela separada onde um processo externo os consiga ler
 - Trigger-based replication tipicamente tem overheads maiores do que os outros replication methods

II.VII Replication Lag

Replication Lag é o que acontece quando seguidores assíncronos vêm outdated information

Causa possíveis inconsistências na BD visto que se corremos a mesma query no líder e num follower ao mesmo tempo, podemos ter resultados diferentes

É crítico em **read-scaling architecture**:

- Nesta arquitetura, procuramos adicionar followers para aumentar a capacidade de servir read-only requests
- Remove carga do líder
- Opção atraente para workloads que consistem, maioritariamente de reads, com um pequeno número de writes

Eventual consistency é a ideia de que estamos num estado inconsistente apenas temporariamente – numa fração de segundo que, em prática, não deve ser noticeable – e que eventualmente vamos obter consistencia

Muito Lag é um problema real para muitas aplicações, podendo este acilmente aumentar para vários segundos ou minutos quando o sistema estiver a operar perto da capacidade maxima, ou quando ocorrerem problemas de network

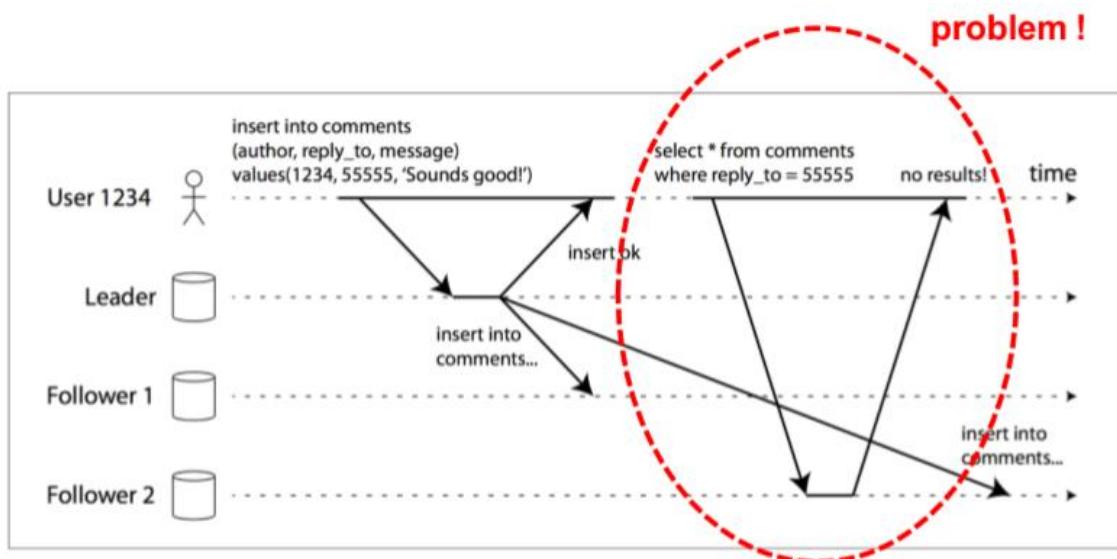
Temos então as seguintes soluções para resolver Replication Lag Problems:

- **Reading your own Writes**

- Quando novos **dados são submetidos**, são enviados para o **líder**, mas quando o **user vir os dados**, estes **podem ser lidos de um follower**
- Esta afirmação é especialmente apropriada se os dados forem frequentemente vistos mas só ocasionalmente escritos
- Um problema com replicação assincrona é que novos dados podem ainda não ter chegado à replica
- A solução aqui proposta é a de manter **Read-After-Write consistency** (ou Read-your-writes consistency)
- Para implementar **Read-After-Write consistency** temos que:
 - **Ler do líder algo que o user possa ter modificado, caso contrario ler de um follower**
 - Para isto, é **necessária uma maneira de saber que dados foram modificados**
 - Potenciais problemas incluem o facto que, se a maioria das coisas forem potencialmente editáveis pelo user, a approach não vai ser effective, negando o

beneficio de read scaling (porque vamos estar sempre a ler do líder)

- Temos, em alternativa outros critérios para saber quando devemos ler do líder:
 - **Dar track do tempo que ocorreu desde o ultimo update**
 - Durante X minutos desde o ultimo update, todos os reads são feitos do líder
 - **Cliente guarda um timestamp dos seus writes mais recentes**
 - O sistema consegue garantir que a réplica que está a servir qualquer read para o user reflete updates, pelo menos até aquele timestamp



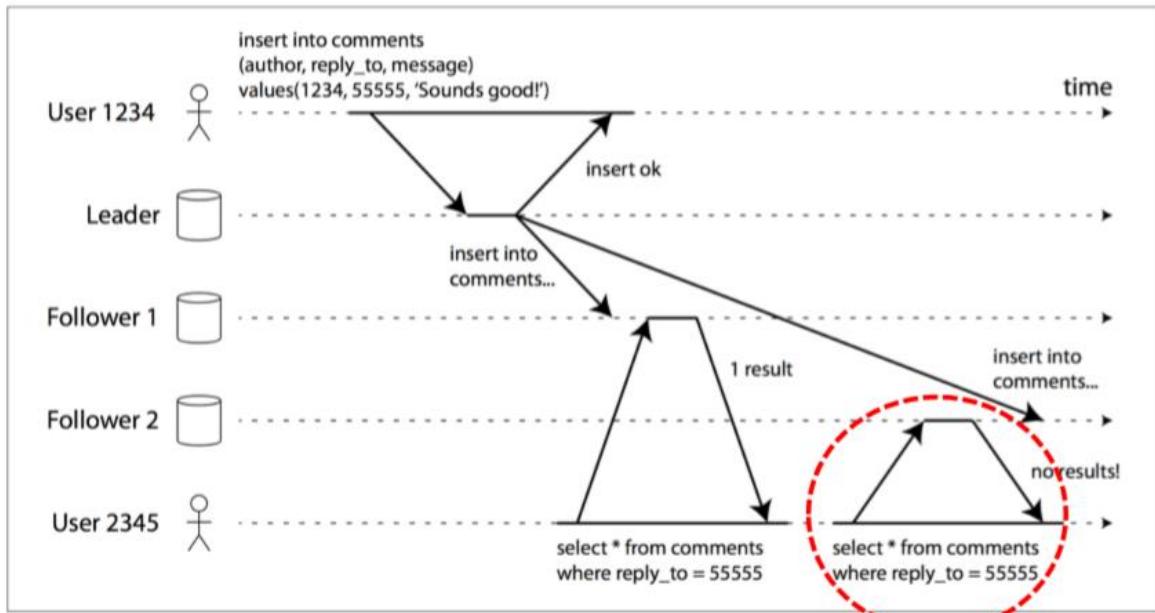
Img 2.7.1 – Problema que acontece caso o cliente tente ler dados de um follower que ainda não tenha recebido o update do líder

• Monotonic Reads

- Quando lemos dados podemos estar a ver valores antigos.
- Isto pode acontecer se reads forem feitos a partir de diferentes replicas.
- Este cenário é bastante provável, p.ex, se um user der refresh de uma web page, visto que cada request é routed para um random server

- **Monotonic Reads** garante que este tipo de anomalia não sucede, garantindo que cada user faz os reads **sempre a partir da mesma replica**
- Obviamente, diferentes users podem usar diferentes replicas

❖ A user is seeing things *moving backwards in time*

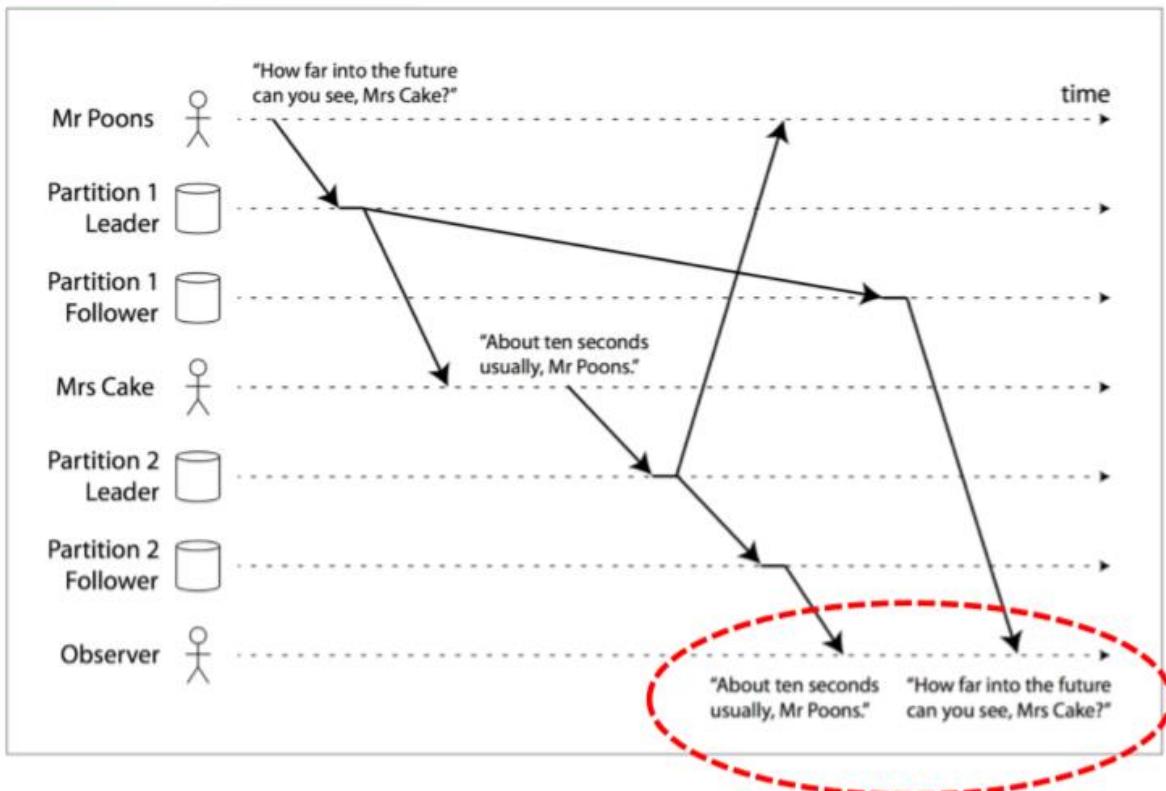


Img 2.7.2 – Problema que acontece caso o cliente leia dados de dois seguidores diferentes e estes tiverem estados inconsistentes

• **Consistent Prefix Reads**

- Replication lag anomalies tem a ver com a **violação de causalidade**
 - Se algumas partições forem replicadas mais lentamente que outras, um observador pode ver uma resposta antes que veja a pergunta
- **Consistent prefix reads** previne este tipo de anomalia garantindo que se uma sequencia de writes acontece numa dada ordem, então qualquer pessoa que esteja a ler esses writes vai vê-los aparecer nessa mesma ordem

- Este problema acontece em bases de dados particionado (sharded) (which we'll see later uwu)



Img 2.7.3 – Problema que acontece caso um observador leia resultados de writes por uma ordem diferente da qual eles foram escritos

III. Multi-Leader Replication

III.I Introdução

Leader Based replication tem alguns problemas, nomeadamente:

- Todos os writes tem de passar pelo mesmo líder
- Se o líder estiver inacessível não vamos poder escrever para a BD

Porém, na **configuração Multi-Leader Configuration, mais do que um node vai ser capaz de aceitar writes!**

Também chamada de master-master replication ou active/active

É uma extensão de leader-based replication na qual a replicação acontece da mesma forma que temos visto até agora – Cada node que processa um write deve dar forward das mudanças dos dados para todos os outros nodes

Cada lider simultaneamente age como um follower para o outro lider

III.II Use Cases

Multi-Leader setup dentro de um único datacenter não é recomendável:

- Os benefícios, neste caos, raramente vão ser úteis devido à complexidade acrescida

É então recomendado para:

- **Multi-datacenter Operations**
 - Uma base de dados com replicas em diferentes datacenters deve ser usada para haver tolerância a failure de um datacenter inteiro ou para que existam datacenters mais perto dos users
 - No Multi-leader setup, vai ser possível que exista um líder em cada data-center
 - Dentro de cada datacenter temos uma Leader-Follower Replication normal
 - O líder de cada datacenter replica as mudanças dos outros datacenter leaders (como se fossem followers uns dos outros)

- **Vantagens**

- **Performance**

- Single-Leader configuration adiciona latencia significante à escrita entre datacenters caso não se use Multi-Leader setups

- **Tolerancia a datacenter problems**

- A multi-leader configuration vai permitir que cada datacenter continue a operar independentemente dos outros

- **Tolerancia a network problems**

- A multi-leader configuration com replicação assincrona normalmente tolera network problems melhor do que single leader
 - Uma temporary network interruption não previne que os writes sejam processados

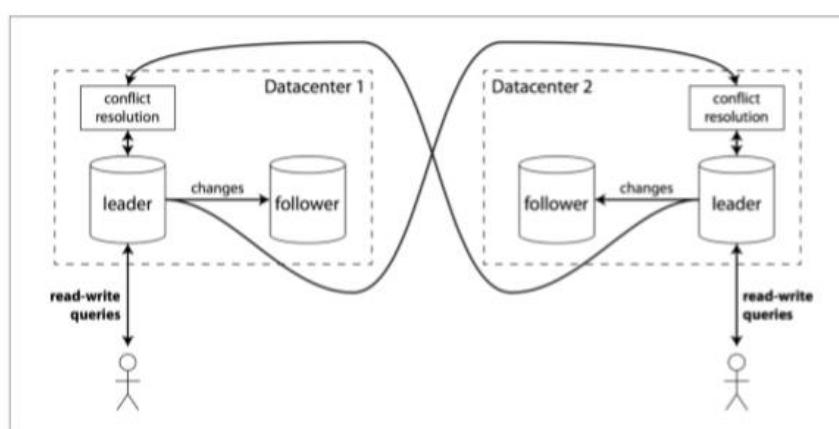
- **Desvantagens**

- **Write Conflicts**

- Os mesmos dados podem ser concorrentemente modificados

- **Auto-Incrementing keys, triggers e integrity constraints podem ser problemáticas**

- **Multi-leader replication é, normalmente, considerada perigosa e deve ser evitada se possível** (wait what)



Img 3.2.1 – Exemplo de Multi-Datacenter Operation com Multi-Leader

- **Clients com offline operation**

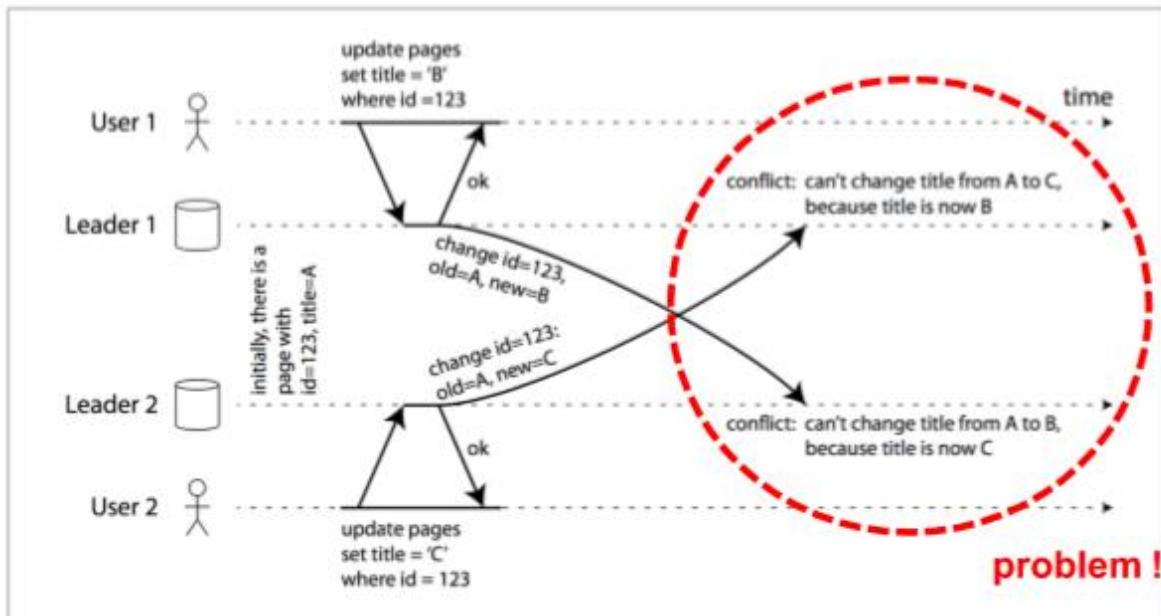
- Todos os devices vão ter uma local database que age como um lider
- O processo de multi-leader replication vai ser **assincrono**
 - Todas as mudanças feitas de forma offline precisam de ser synced com um servidor e com outros devices quando este voltar a ficar online
 - **Replication lag de horas ou até dias**
- De um ponto de vista arquitetónico, é essencialmente o mesmo que multi-leader replication entre data-centres mas taken to the EXTREME
 - Funciona quase como que se cada device fosse um datacenter e a network connection entre eles fosse unreliable

- **Collaborative Editing**

- Aplicações de Real-Time Collaborative Editing permitem que várias pessoas editem um documento de forma simultânea
- Quando um user edita o document as mudanças são **aplicadas instantaneamente à local replica e replicadas para o servidor** e quaisquer outros users que também estejam a editar o documento de **forma assincrona**
- Para evitar conflitos de edição, as aplicações têm de obter um **lock** no document **antes que o user o possa editar**
- **Faster Collaboration** faz requests de uma unidade de change muito pequena para evitar locks
 - Isto permite que multiplos users editem de forma simultanea, mas também tras a dificuldade de multi-leader replication, incluindo resolução de conflitos

III.III Multi-Leader Write Conflicts

O maior problema com multi-leader replication é o facto que **Write Conflicts** podem acontecer (o que não sucede em Single Leader configurations)



Img 3.3.1 – Exemplo de um Write conflict que sucede devido a dois líderes atualizarem o mesmo record de forma concorrente. O conflito é detetado quando as mudanças são replicadas de forma assíncrona

Um grande número de implementações multi-leader tratam conflicts de forma muito má.

A melhor estratégia para lidar com conflicts é **evita-los** (lmao genius big brain time)

Se a aplicação conseguir assegurar que todos os writes para um particular record são feitos pelo mesmo líder, então conflitos não podem acontecer (olha no shit colega)

Podemos tornar a **conflict detection sincrona**, ou seja, esperar que o write seja replicado para todas as replicas antes de dizer ao user que o write foi realizado com sucesso. Porém ao fazermos isto estamos a foder

a maior vantagem que multi-leader replication tem – O facto de permitir que cada replica aceite writes de forma independente

Bases de dados têm de resolver conflitos de forma convergente – Todas as replicas têm de ter os mesmos valores quando as mudanças tiverem sido replicadas

Temos ainda duas formas de **conflict resolution**

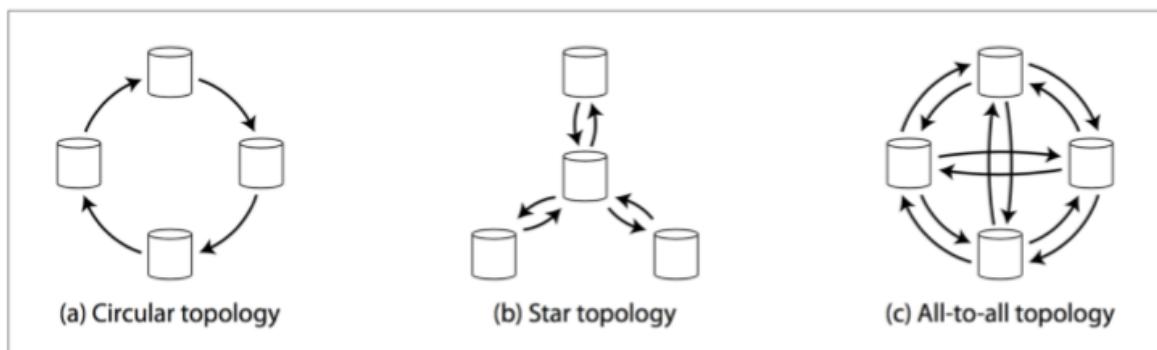
- **Convergent Conflict Resolution**, dos quais são exemplos:
 - **Writes com Unique ID**
 - O maior destes será o vencedor
 - Podemos usar a timestamp – tecnica conhecida como **last write wins (LWW)**
 - **Replicas com unique ID**
 - Writes de replicas com maior numero são o vencedor
 - **Merge dos valores** (i.e data concatenation)
 - **Armazenar o conflito numa estrutura de dados explicita** com toda a informação e escrever **application code** que seja capaz de resolver esses conflitos posteriormente (p.ex perguntando ao user que dados manter)
- **Custom Conflict Resolution Logic**, que consiste em:
 - Escrever lógica de resolução de conflitos usando application code
 - Usualmente aplicada a individual row ou document level e não a transações inteiras
 - Código pode ser executado no processo de write ou de read

III.IV Multi-Leader Replication Topologies

Replication Topology descreve os paths de comunicação que propagam os writes de um node para outro

Existem diversos tipos de topologias:

- **All-to-All**
 - Topologia mais geral
- **Circular**
 - Topologia mais restrita
 - MySQL, por default, só suporta esta topologia
- **Star**
 - Pode ser generalizada a uma árvore



Img 3.4.1 – Exemplos de Multi-Leader Topologies

IV. Leaderless Replication

IV.I Introdução

Em Leaderless Replication, não existe nenhum leader (schocker omg :O)

Qualquer replica aceita writes dos clientes

Existem implementações nas quais o cliente manda diretamente os seus writes a várias replicas

Noutras um **coordinator node** faz isto por parte do cliente (ao contrário de um líder, o coordenador não enforca nenhuma ordenação particular dos writes, simplesmente age como uma proxy enviando os writes do cliente para os nodes)

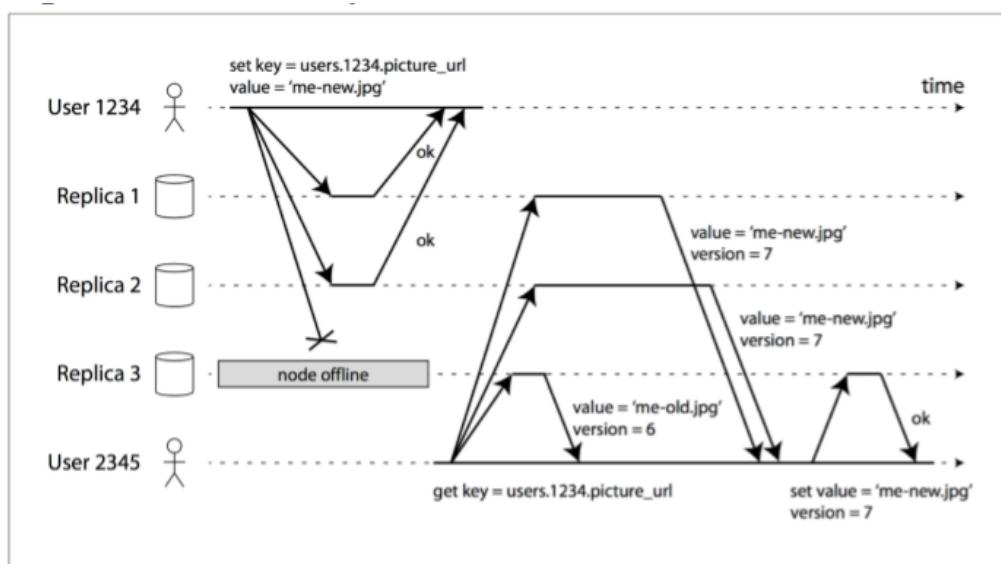
Leaderless Replication foi utilizada em alguns dos mais antigos replicated data systems:

- **Tornou-se Hype** após a Amazon a ter usada para o seu in-house Dynamo System
- **Riak, Cassandra e Voldemort** são todas datastores inspiradas pela Dynamo

IV.II Escrever quando um node está down

Numa leaderless configuration, não existe o conceito de **failover**

Quando um node fica offline fodeu-se e ta fodido, nobody cares



Img 4.2.1– Exemplos do que acontece quando, numa BD com 3 replicas, uma das replicas fica offline (Replica 3). Cagamos e seguimos em frente irmão

IV.III Quando um node volta a ficar Online

Se um unavailable node voltar a ficar online o cliente pode começar a ler a partir dele.

Qualquer write que tenha acontecido enquanto o node estava down não vão estar instantaneamente no node... Pode fazer com que reads retornem Stale (outdated) values nas respostas

Read Requests são mandados para vários nodes em paralelo

Cientes podem receber respostas diferentes dos vários nodes, para determinar qual é o mais up-to-date utilizamos um **Version number**

IV.IV Recuperar de Missing Writes

Como é que um node que esteve offline e voltou agora a estar online consegue reaver os writes em falta?

Podemos utilizar dois mecanismos:

- **Read Repair**
 - Cliente lê a partir de varios nodes em paralelo
 - Se detetar Stale Responses, escreve os valores mais recentes na replica da qual recebeu a stale data
 - Funciona fixe para valores que são frequentemente lidos
- **Anti-Antropy process**
 - Para além disso, alguns datastores tem **background processes** que constantemente procuram por diferenças dos dados nas replicas e resolvem esses problemas

Nem todos os sistemas implementam ambos os mecanismos

Sem anti-entropy processes, os valores que são raramente lidos podem estar missing de várias replicas, pelo que vão ter **reduced durability** (Caso do Voldemort)

IV.V Quorum para Reading e Writing

quó·rum

(latim *quorum*, dos quais, genitivo plural do pronome relativo *qui*, *quæ*, *quod*, o qual, quem, que)

substantivo masculino

1. Número necessário de membros para que uma assembleia possa funcionar.
2. Número de pessoas imprescindível para a realização de algo.

Plural: quórums.

Palavras relacionadas: [quorum](#).

Img 4.5.1 – Definição de Quorum

Quorum Consistency é uma técnica que procura fazer com que os clientes adquiram permissão de várias réplicas antes de realizarem operações de escrita ou leitura

Se n replicas existem, precisamos de criar um **Read Quorum** – Coleção de r réplicas arbitrárias - para ler e um **Write Quorum** – coleção de w réplicas arbitrárias - para escrever

Para prevenirmos read-write conflicts precisamos de cumprir a **Quorum Condition** – $w + r > n$

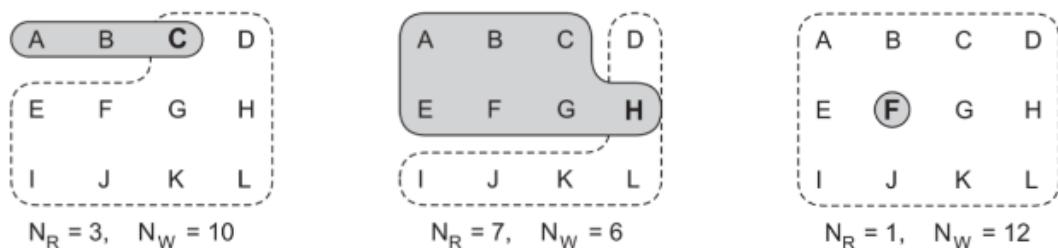
Ou seja, para queries realizadas a n replicas, cada write tem de ser confirmado por w nodes para ser bem sucedido, e o cliente deve receber respostas iguais de pelo menos r nodes.

Sem Quorum os writes ou reads retornam um erro ou stale data

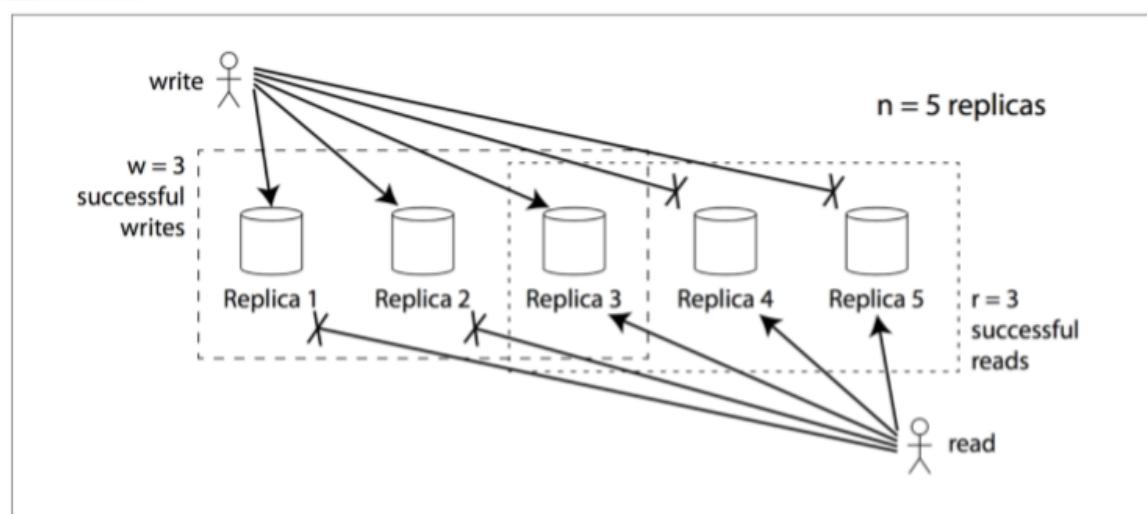
Normalmente, reads e writes são sempre mandados para todas as n replicas em paralelo, mas com BDs que façam uso de Quorum Consistency, vai ser possível tolerarmos o failure de individual nodes sem precisarmos de mecanismos de failover!

Os valores de w e r são tipicamente configuráveis, sendo preciso apenas cumprir a Quorum Condition

(a) Escolha correcta de read e write (b) Posível write-write (c) Escolha correcta chamada ROWA (Read One Write All)



Img 4.5.2 – Exemplo de escolha dos valores de R e W que cumprem a quorum condition ($W + R > N$)



- ❖ A common choice is to make n an odd number (typically 3 or 5), and to set $w = r = (n + 1) / 2$ (rounded up)
- ❖ However, you can vary the numbers as you see fit
 - for example, a workload with few writes and many reads may benefit from setting $w = n$ and $r = 1$
 - this makes reads faster, but has the disadvantage that just one failed node causes all database writes to fail

Img 4.5.3 – Exemplo do uso de Quorum consistency

Se não houver Quorum Consistency ($w+r \leq n$), reads e writes vão ser enviados para n nodes na mesma, mas vai ser preciso um número mais pequeno de successful responses para a operação ser considerada bem sucedida.

Isto vai fazer com que stale data seja mais provável de ser lida

Esta configuração permite latência mais pequena e availability maior – se houver network interruption e muitas replicas se tornarem unreachable, há maior chance que consigamos continuar a processar reads e writes se não impusermos quorum consistency

IV.V Sloppy Quorum (uwu)

Em grandes clusters, devido a network interruption, um cliente pode não ser capaz de se ligar a todos os nodes necessários para criar um quorum

Nesses casos, os designers da BD enfrentam um tradeoff:

- a) Será preferível retornar erros a todos os requests para os quais não conseguimos chegar a um quorum de w ou r nodes?
- b) Ou será melhor aceitar writes de qualquer maneira e escrever os em alguns nodes que sejam reachable mas que não estejam entre os n nodes nos quais o valor normalmente vive?

A opção b) é conhecida por **Sloppy Quorum**

Writes e Reads continuam a precisar de w e r respostas bem sucedidas, mas essas devem poder incluir nodes que não estejam entre os designados n “home” nodes para um valor

Assim que a network interruption for fixed, quaisquer writes feitos a um node temporariamente aceite on behalf de outro node são mandados para os “home nodes” apropriados.

A este processo chamamos **Hinted Handoff**

IV.VI Multi-Datacenter Operation

Leaderless replication é apropriada para ser usada em **multi-datacenters**

Tolera conflicting concurrent writes, network interruptions e latency spikes.

Cassandra e Voldemort, p.ex, implementam os seus multi-datacenter support dentro de um leaderless model normal (permitindo a especificação de quantas n replicas é que temos em cada datacenter)

Cada Client Write é enviado a todas as replicas, independentemente do datacenter.

O cliente, por norma, espera pelo acknowledgement de um quorum den odes dentro do seu local datacenter, de forma a evitar delays e interrupções que ocorram nas ligações cross-datacenter

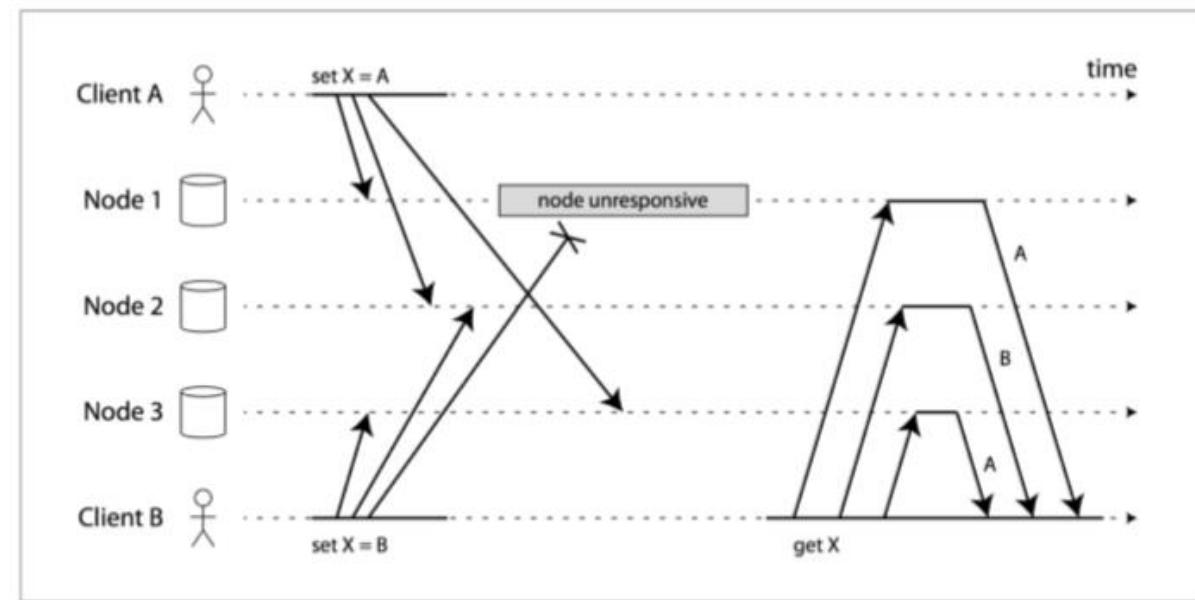
Os Higher-Latency writes para outros datacenters estão normalmente configurados para acontecer de forma assincrona

IV.VII Detetar e Tratar Concurrent Writes

Vários clientes podem realizar concurrent writes para a mesma key

Conflitos vão ocorrer mesmo que quorums muito estritos sejam usados.

O problema é o facto que os eventos podem chegar por uma ordem diferente a diferentes nodes (devido a network delays ou partial failures)



Img 4.7.1 – Problema de Concurrent Writes

Temos então as seguintes formas para tratar Concurrent Writes:

- **Last Write Wins – LWW**

- Cada **replica guarda apenas o valor mais recente**, permitindo que **valores mais antigos** sejam **overwritten e descartados**
- Requer uma maneira de detetar qual dos writes é mais recente (p.ex na ultima figura não é muito claro qual write foi iniciado primeiro)
- Uma possivel forma é através do uso de timestamps – Dando attach de um timestamp a cada write, e escolhendo o maior, vamos conseguir o write mais recente
- Consegue atingir **eventual convergence** mas ao **custo de durability** (dados vão ser overwritten)
- LWW é o unico conflit resolution method suportado em Cassandra, sendo uma feature opcional em RIAK

- **Keys With Version Number**

- Funciona da seguinte forma:
 1. O servidor mantem um **version number para cada key** que é incrementado sempre que um valor for escrito nessa key
 2. **Clientes têm de ler a key antes de lhe escrever**
 3. Quando a key é lida, o server retorna os **valores** (que não foram overwritten) e o **version number da key**
 4. O cliente pode atualizar um item, mas só se o version number no server side não tiver mudado
 5. Se houver um version mismatch, isso significa que alguém (um processo concorrente) modificou o item antes de nós – O nosso update attempt vai falhar porque estavamos a tentar atualizar uma versão stale do item
 6. Se isto acontecer, vamos simplesmente tentar ir buscar os valores mais recentes antes de os tentarmos atualizar
- Amazon DynamoDB chama a isto “Optimistic Locking with Version Number”

- **Merging Concurrently Written Values**

- Garante que **não ha dados dropados silenciosamente**
- Os clientes têm de dar merge dos concurrently written values (O Riak chama a esses valores concorrentes – irmãos, é lidar irmaum)
- Possíveis approaches são:
 - Usar uma simples LWW ou Version Number Approach
 - Fazer algo mais inteligente, como p.ex, uma union para dar merge dos values
 - Não apagar dados, em vez disso deixamos um marcador com um version number apropriado para indicar que o item foi removido ao dar merge

de siblings. Esse deletion marker chama-se de **tombstone**

- **Version Vectors**

- Algoritmo usado para multiplas replicas sem lider
- Usa um **Version Number** por **Replica** e um **Version Number** por **Key**
- A esta coleção de version numbers chamamos de **Version Vector**
- Cada replica incrementa o seu proprio verison number quando processa um write
- Para além disso, cada replica tem de dar keep track de todos os version numbers que já viu de todas as outras replicas.
- Pode depois utilizar essa informação para perceber que values devem ser overwritten e que values devem ser mantidos como irmãos
- Existem algumas variants desta ideia como o dotted version vector usado em Riak 2.0

Partitioning de Dados Distribuidos

I. Partitioning

Partitionar significa partir uma base de dados grande em vários subsets mais pequenos chamados **partições**, de forma a que diferentes partições possam ser atribuídas a diferentes nodes (processo conhecido como **sharding**)

Isto é deve ser feito quando temos datasets muito grandes ou quando possuimos um very high query throughput

Normalmente, partições estão definidas de forma a que **cada pedaço de dados** (cada record, row ou document) **pertence a exatamente uma e uma só partição** (i.e duas partições não tem os mesmos dados)

Cada partição pode ser considerada uma pequena base de dados

BDs podem suportar operações sobre múltiplas partições ao mesmo tempo.

Em MongoDB, Elasticsearch e SolrCloud, partições chamam-se **Shards**.

Em BigTable chamam-se **Tablets**.

Em Hbase chamam-se **Regions**

Em Cassandra e Riak chamam-se **Vnodes**

Em Couchbase chamam-se **vBuckets**

Particionamento de dados é realizado, maioritariamente por razões de **escalabilidade**:

Um large dataset pode ser distribuido sobre multiplos discos, e a query load distribuida sobre multiplos processos

Partições distintas podem ser colocadas em diferentes nodes numa **Shared-Nothing Cluster**.

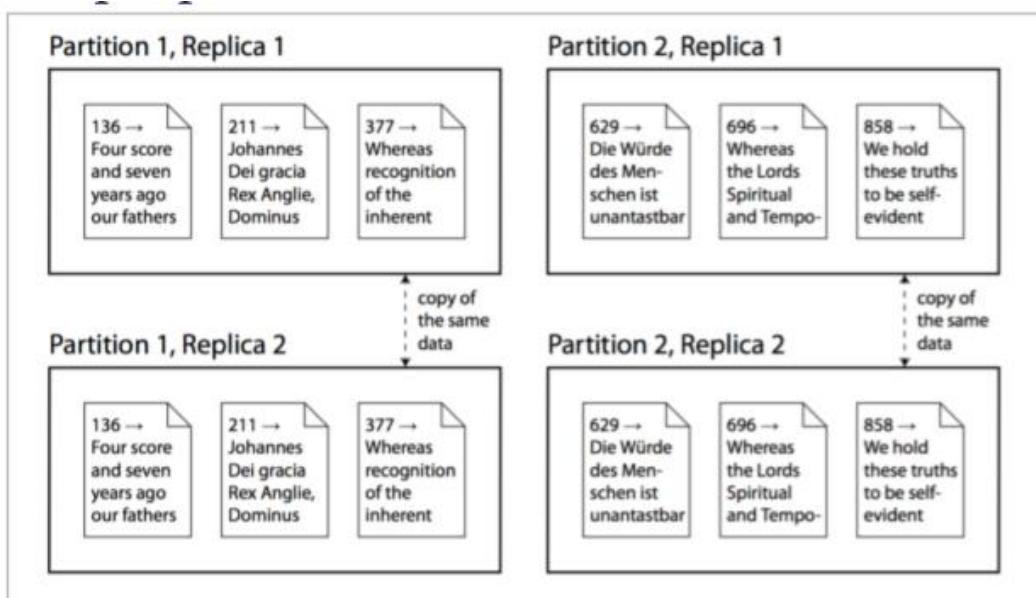
Neste caso, vamos ter pequenas queries que vão operar numa unica partição – Cada node pode executar queries independentemente para a sua própria partição. Com isto podemos escalar o query throughput simplesmente adicionando mais nodes

Queries grandes e complexas podem ser, com isto, potencialmente paralelizadas sobre multiplos nodes!

II. Partitioning e Replication

Partitioning pode, e é, normalmente combinado com **replication**

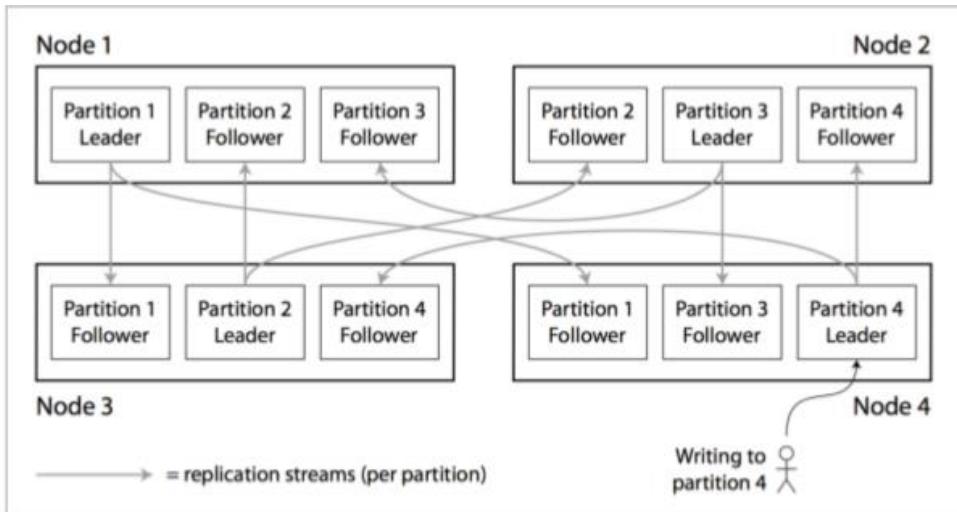
Um node **pode** guardar mais do que uma partição, e uma partição **pode** estar guardada em diferentes nodes



Img 2.1 – Exemplo de uma BD partida em duas partições com duas replicas por partição

Focando-nos agora no **Leader-Follower Replication Model**:

- O líder de cada partição é atribuído a um node e os seus followers são atribuídos a outros nodes
- Cada Node pode ser o líder para algumas partções, mas um follower para outras!



Note: the choice of partitioning scheme is mostly independent of the choice of replication scheme, so we will keep things simple and ignore replication in this section

Img 2.2 – Exemplo de Leader-Follower Replication com Partition

III. Partitioning de Dados

O desafio que nos é apresentado é o de **Particionar uma grande quantia de dados**

O objetivo é **espalhar os dados e query loads pelos nodes de forma uniforme**

Obviamente, vamos ter problemas:

- **Skewed**
 - Algumas partções vão ter mais dados ou queries que outras

- **Hot Spot**

- Quando uma partição tem uma carga muito desproporcionalmente maior que as outras (caso extremo de skewed)

Vamos, claramente, querer evitar Hot spots ao maximo.

A forma mais simples disso é **atribuir records a nodes de forma aleatoria** (probabilisticamente, os dados devem acabar com uma distribuição uniforme)

Uma desvantagem disto, porém, é que vamos precisar de realizar queries a todos os nodes em paralelos para termos itens... (cus we don't rlly know where they were placed)

IV. Partitioning de Dados Key-Value

Nesta secção vamos olhar para alguns algoritmos de particionamento de dados com as assumções que:

- Estamos a usar um simples Key-Value Data Model
- Acedemos sempre a um record usando a sua Primary Key

Temos então as approaches:

- **Partitioning por Key Range**
- **Partitioning por Hash da Key**
- **Skewed Workloads**

IV.I Partitioning por Key Range

Esta técnica consiste em **atribuir ranges continuos de chaves a cada partição**

Com isto vamos:

- **Saber as fronteiras entre ranges**
- Escolher as **fronteiras automaticamente ou manualmente**
- Ser capazes de facilmente **determinar que partição contém dada Key**
- Ser capazes de **realizar requests diretamente ao node** apropriado



Img 4.1.1 – Exemplo de Volumes de Enciclopedia. Temos 12 partições cada uma com um range alfabetico

Note-se que os **ranges das keys podem não estar, necessariamente, espaçados de forma uniforme...**

Os dados, portanto, podem não ser distribuidos igualmente

No exemplo anterior, o Volume 1 vai conter palavras de A até B, mas o volume 12 vai conter palavras de T até Z

Dentro de cada partição, podemos manter as keys ordenadas (com SSTables e LSM-Trees).

Desta forma, os **range scans vão ser mais fáceis**

Uma chave pode ser um index concatenado de forma a conseguir dar fetch de varios related records numa query

P.ex um timestamp “ano-mes-dia-hora-minuto-segundo”

O problema com isto é que certos padrões podem-nos levar a **hot spots**

Esta técnica é usada em MongoDB (<v2.4) , BigTable, Hbase, RethinkDB

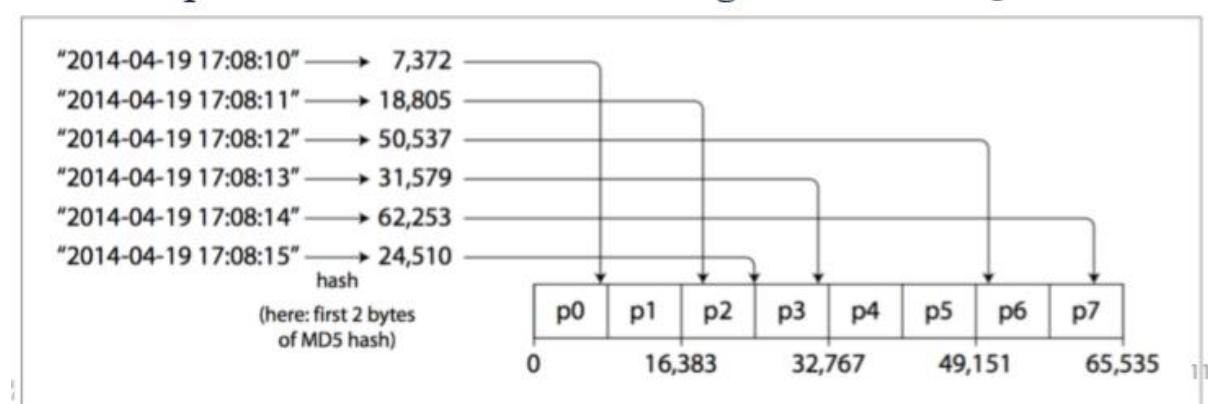
IV.II Partitioning por Hash da Key

Usam-se **Hash Functions** para determinar a partição para uma dada key.

Reduz o risco de skew e hot spots, e por isso é muito usada por datastores distribuidos

Uma boa hash function torna dados skewed e transforma-os numa distribuição uniforme!

P.ex Cassandra e MongoDB usam MD5, enquanto que Voldemort usa Fowler-Noll-Vo



Img 4.2.1 – Exemplo do uso de partição por Hash de Key

A ideia é que:

- Atribuimos a Key à partição 0 se $0 \leq \text{hash}(\text{key}) < b_0$
- Atribuimos a Key à partição 1 se $b_0 \leq \text{hash}(\text{key}) < b_1$
- And so on...

Os **problemas** deste método incluem:

- Perda de efficient range queries
- Perda de keys ordenadas

Exemplos de uso incluem:

- ❖ MongoDB with hash-based sharding enabled
 - range query are sent to all partitions
- ❖ Riak, Couchbase and Voldemort
 - range queries on the primary key are not supported
- ❖ Cassandra - compromise between two strategies
 - a table can be declared with a *compound primary key* consisting of several columns. Only the first part of that key is hashed to determine the partition, but the other columns are used as a concatenated index for sorting the data in SSTables
 - can search by a fixed value for the first column and perform an efficient range scan based on the other columns of the key
 - allows an elegant data model for one-to-many relationships



Img 4.2.2 – Exemplos de IRL use de Hash of Key

12

IV.III Skewed Workloads

O método anterior **não elimina hot spots inteiramente**

Num caso muito extremo, todos os reads e writes são feitos à mesma Key, pelo que todos os requests vão ser routed para a mesma partição...

Este tipo de workload não usual...Mas também não é impossível

P.ex a conta do insta do DS, com milhoes de seguidores vai causar uma storm de atividade quando este fizer alguma coisa controversa...its bound to happen

Alguns data systems não são capazes de automaticamente detetar e compensar tal highly skewed workload, pelo que a **responsabilidade de reduzir o skew vai ser da aplicação**

Uma possível approach seria adicionar um numero aleatório ao inicio ou ao fim da chave

P.ex com um Numero de dois digitos iríamos partitir os writes a uma chave de forma uniforme por 100 chaves diferentes

V. Partitioning e Secondary Indexes

Records são acedidos via primary key permitem-nos determinar em que partição é que vamos ter que efetuar a escrita e leitura

Mas e se estivermos a usar Secondary **Indexes**?

❖ DB Scenario

- well supported by relational databases and common in document databases
- key-value stores (e.g. HBase and Voldemort) avoided secondary indexes because of their added implementation complexity
 - Riak support them because are useful
- *raison d'être* of search servers such as Solr and Elasticsearch

Img 5.1 – Casos do uso de Secondary Indexes em BDs reais

Temos duas formas de particionar uma base de dados com secondary indexes:

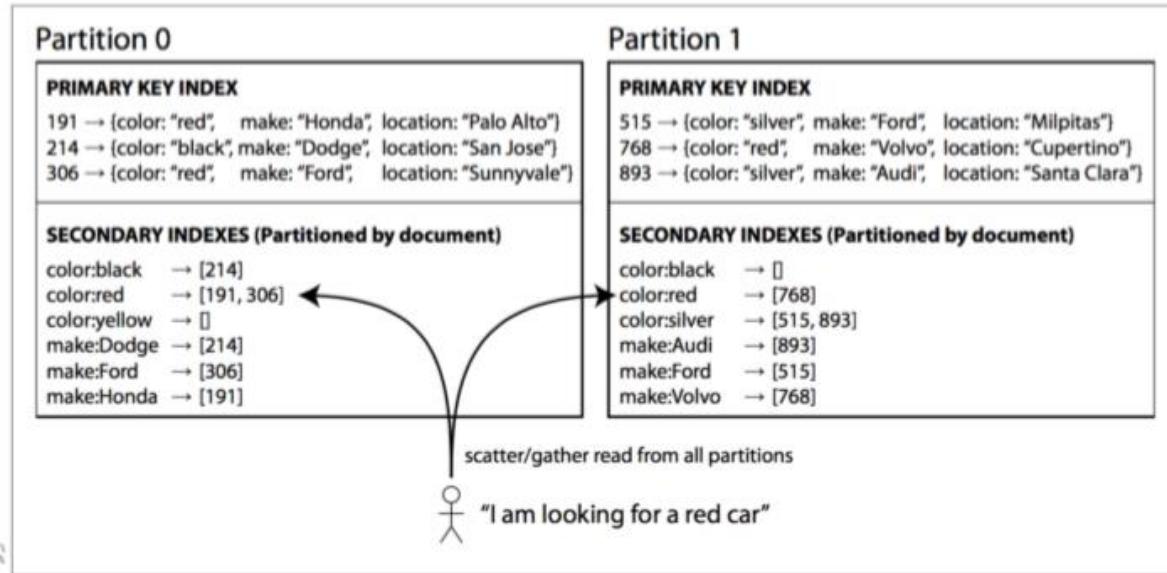
- **Document-Based Partitioning**
- **Term-Based Partitioning**

V.I Document Partitioned Index

Características:

- **Index por partição** (conhecido como **local index**)

Exemplo:



Img 5.1.1 – Exemplo – DB para vender carros usados. Os UniqueID (document ID) dos carros são usados para particionar a DB. Temos secondary indexes para cor e fabricante

Cada partição vi operar do mesmo modo:

- **Writing (Add, Remove ou Update)**
 - Só precisamos de lidar com a partição que contenha o **Document ID**
 - Cada partição mantém o seu próprio secondary index, que cobre apenas os documentos naquela partição
- **Reading**
 - Requer que se mandem **queries a todas as partições** e que se combinem os resultados obtidos
 - Esta query approach é conhecida como **scatter/gather**
 - É um processo caro mesmo se fizermos as queries às partições em paralelo, visto que com isto a latencia é proporcional (ou piça) a aumentar

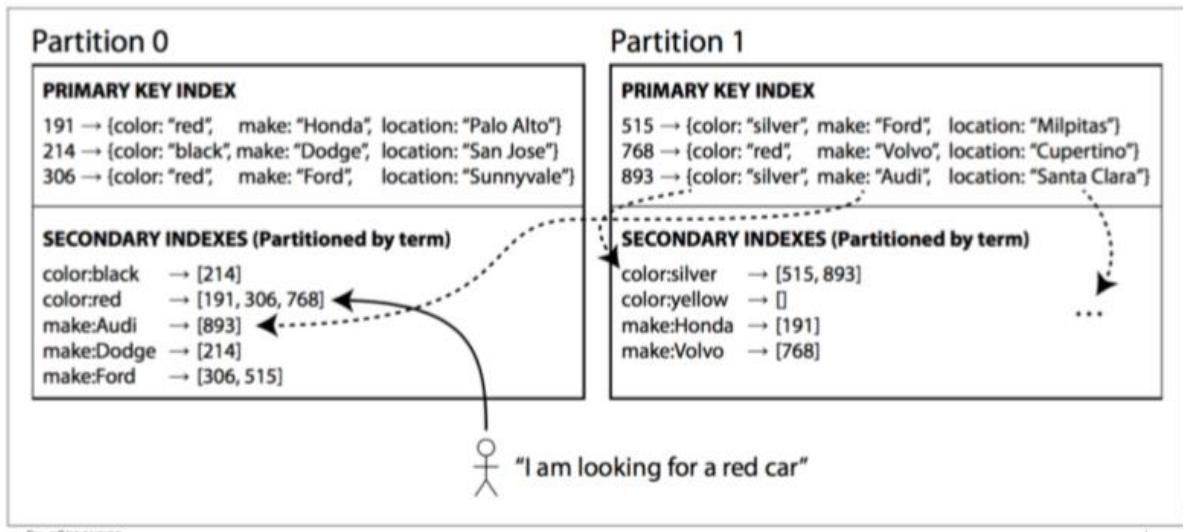
Usado por MongoDB, Riak, Cassandra, Elasticsearch, SolrCloud e VoltDB

V.II Term-Based Index

Características:

- **Index Global** que cobre dados em todas as partições
- **Particionado de forma diferente** do que a **primary key index**
 - Distribuido pelas partições por Termo ou Hash
 - Se guardassemos o index apenas num node, estariamos a criar um bottleneck...big F

Exemplo:



Img 5.2.1 – Exemplo – DB para vender carros usados. Os UniqueID (document ID) dos carros são usados para particionar a DB. Temos secondary indexes para cor e fabricante

Particionar Secondary Indexes por term ou por hash também é possível com as vantagens e desvantagens que já foram discutidas

Vantagens de Term-Partitioned Indexes sobre Document-Partitioned Indexes:

- **Reads mais eficientes**
 - Cliente só tem que fazer um request à partição que contem os termos

Desvantagens de Term-Partitioned Indexes sobre Document-Partitioned Indexes:

- **Writes mais lentos e complicados**
 - Escrever um documento pode agora afetar multiplas partições

Updates a global secondary indexes são, por norma **assincronos**.

P.ex, Amazon DynamoDB diz que os seus global secondary indexes são atualizados numa fração de segundo

VI. Rebalancionamento de Partições

Numa base de dados, as coisas podem mudar com o decorrer do tempo, e estas mudanças requerem counteractions

- Query Throughput pode aumentar
 - Precisamos de adicionar mais CPUs para
- Tamanho do dataset aumenta
 - Precisamos de adicionar mais discos e RAM para os guardar
- Maquina Falha
 - Outras maquinas tem de assumir as responsabilidades

Rebalancing é o processo de **mover dados** (partições) **entre nodes** numa cluster

Independentemente do esquema de particionamento usado, rebalancing tende a ter os seguintes minimum requirements:

- **Depois do rebalancing**, a **carga** (data storage, read e write requests) deve estar **partilhada de forma justa** pelos nodes numa cluster

- **Enquanto o rebalancing** está a ocorrer, a **BD** deve **continuar a operar** (aceitando reads e writes)
- **Não devemos mover dados** entre nodes quando **não é necessário**, de forma a evitarmos sobrecarregarmos a network

Note-se que **rebalancing é uma operação cara**:

- Requer re-routing requests e a movimentação de largas quantidades de dados entre nodes
- Pode sobrecarregar a network ou os nodes, causando problemas de performance para outros sistemas

Em combinação com **automatic failure detection** pode ser perigosa porque podemos causar uma **cascading failure**

O **Rebalancing** pode ser **Automático** ou **Manual**:

- **Fully Automated Rebalancing**
 - É conveniente porque requer menos trabalho operacional/gestão
 - Infelizmente é unpredictable podendo causa resultados não desejados
- **Human controlled Rebalancing** (recomendado)
 - Recomenda-se que um humano controle o processo de rebalancing
 - Vai tornar-se um processo mais lento, MAS vai prevenir surpresas operacionais
 - Chouchbase, Riak e Voldemort geram uma suggested partition assignment automatically, mas requerem que um administrador dê commit

Vamos agora ver 3 estratégias de rebalancing:

- **Hash mod N**
- **Fixed Number Partitions**
- **Dynamic Partitioning**

VI.I Hash mod N

Quando temos **particionamento por hash da key**:

- Atribuimos a Key à partição 0 se $0 \leq \text{hash}(\text{key}) < b_0$
- Atribuimos a Key à partição 1 se $b_0 \leq \text{hash}(\text{key}) < b_1$
- ...

A estratégia de rebalanciamento vai ser portanto:

- **Round Robin Hash(key) mod N**
 - $N = \text{numero de nodes}$
 - P.ex para $N=10$, vai retornar um numero entre 0 e 9

Quando o numero total de nodes, N , mudar, a maioria das keys vão ter que ser movidas de um node para outro

Torna rebalancing excessivamente caro

VI.II Fixed Number of Partitions

Nesta approach vamos considerar:

- **Numero fixo de partições**
 - Consideramos que o numero de partições não muda
 - O que muda é o assignment de partições a nodes
 - Apenas partições inteiras são movidas entre nodes
- **Temos mais partições que nodes**
 - Vamos atribuir várias partições para cada node

Exemplo:

Uma base de dados a correr numa cluster de 10 nodes pode ser split em 1000 partições, e 100 partições serem dadas a cada node

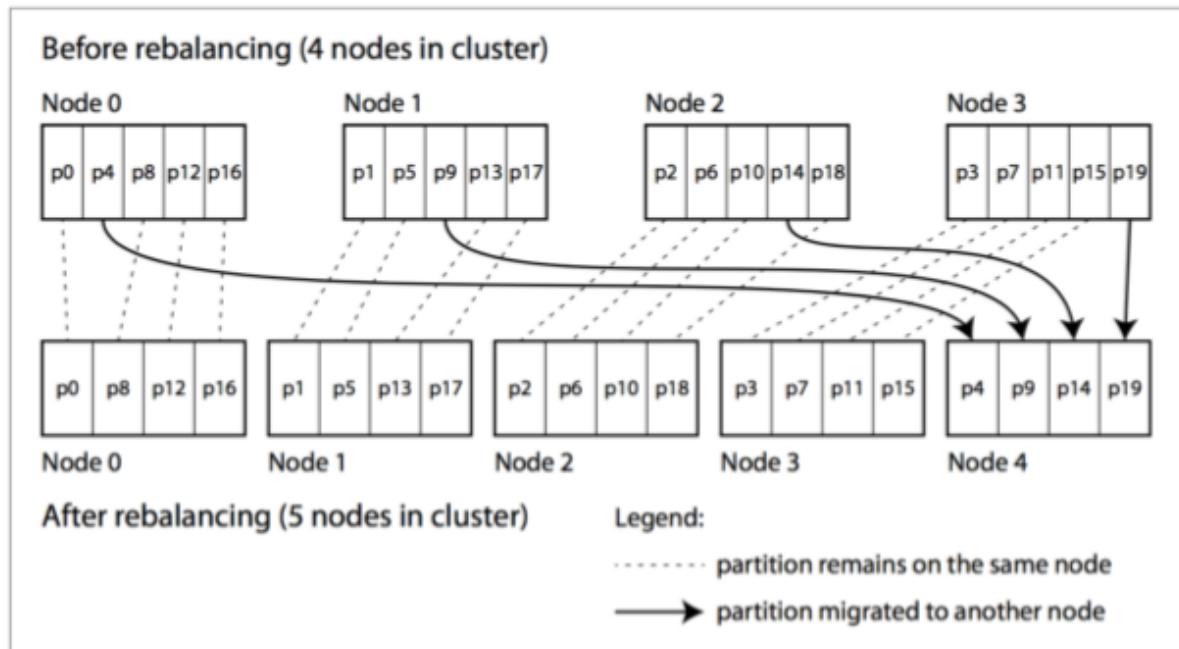
Técnica usada por Riak, Cassandra (>v1.2) Elasticsearch, Couchbase e Voldemort

Adicionar um novo node:

- Rouba partições de todos os existing nodes até que as partições estejam justamente distribuidas outra vez

Remover um node:

- Processo inverso do de adicionar



Img 6.2.1 – Exemplo de como funciona o rebalanciamento

VI.III Dynamic Partitioning

Quando temos **particionamento por key range**:

- Vamos ter o problema de Skew e Hot spots

A solução é **Criar Partições de forma Dinâmica**

Partition Split

- Quando a partição crescer e exceder o tamanho configurado
- **Partimos** a partição em duas com metade do tamanho cada
- Após o split, a partição pode ser transferida para outro node

Partition Merge

- Quando muitos dados são apagados, podem ser merged com a partição adjacente

Usado em BDs como a Hbase e RethinkDB

Vantagens:

- Número de partições adapta-se ao volume dos dados

Limitações:

- Quando o dataset for pequeno funciona com uma única partição
- Isto implica que todos os writes vão ser processados por um único node enquanto que os outros não fazem piço...
- Para mitigar este problema, Hbase e MongoDB permite dar setup de um número inicial de partições numa BD vazia (pre-splitting)

Partitioning Dinamico também pode ser usado com **hash-partitioned data**

O MongoDB, desde a versão 2.4, suporta ambos Key-Range e Hash-Partitioning, mas parte as partições dinamicamente em ambos os casos

VII. Routing Process

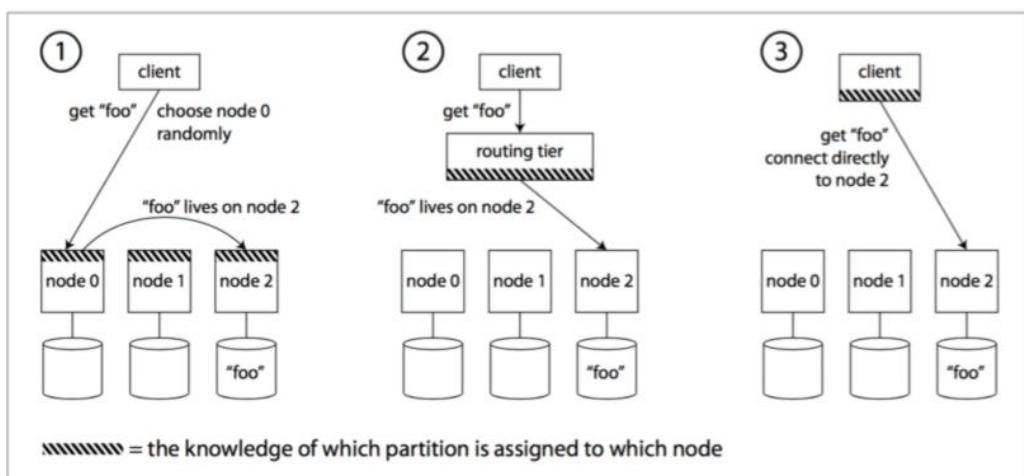
Imagina o cenário:

- Temos o dataset particionado por multiplos nodes a correrem em multiplas máquinas
- As partições estão rebalanced (o assignment de partições aos nodes pode ser mudado)

Como é que um cliente sabe a que node se deve conectar? :(

A isto chama-se o **Service Discovery Problem**

- Não é exclusivo a BDs
 - Muitas empresas desenvolvem e usam in-house service discovery tools
 - Muitas destas tools são lançadas como open source (nice)
- ❖ Examples of different ways of routing a request to the right node (1. node; 2. routing tier; 3. client)



Img 7.1 – Diferentes approches de Routing

Temos várias formas distintas de Routing Approaches, mas todas têm o mesmo problema:

- Como é que componente que toma as routing decisions...as toma?

Protocolos para atingir consensus num distributed system são dificeis de implementar corretamente :(

Muitos distributed data systems dependem de **separate coordination services** para darem keep track de metadados da cluster

Alguns exemplos destes serviços incluem:

❖ ZooKeeper

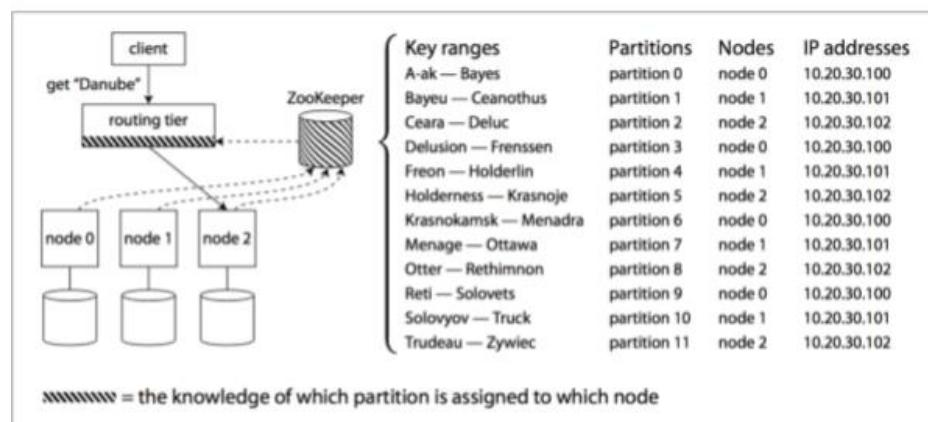
- HBase, SolrCloud and Kafka also use ZooKeeper to track partition assignment
- LinkedIn's Espresso uses Helix (relies on ZooKeeper)
- MongoDB has a similar architecture, but relies on its own *config server* implementation and *mongos* daemons as routing tier

❖ Cassandra and Riak take a different approach (~ approach number 1 of first figure)

- gossip protocol among the nodes to disseminate and agree on any changes in cluster state
- requests can be sent to any node, and that node forwards them to the appropriate node for the requested partition
- disadvantages: more complexity in the database nodes
- advantages: avoids the dependency on an external service

❖ Couchbase does not rebalance automatically, which simplifies the agreement protocol between nodes

Img 7.2 – Market Solutions para o Routing Process



Img 7.3 –
ZooKeeper Example

Processamento de Dados Distribuidos - MapReduce

I. Mapreduce e Hadoop - Introdução

O MapReduce foi uma técnica desenvolvida na Google

MapReduce é uma **framework** para facilmente **escrever aplicações** que processam **largas quantias** de **dados** em **paralelo** numa **cluster**.
Fornece:

- **Paralelização e Distribuição Automática**
- **Fault Tolerance**
- **I/O Scheduling**
- **Monitoring e Status Updates**

Usa um modelo de programação do Lisp (e outras linguagens funcionais)

Pode ser pensado como uma **Solução Padrão** visto que muitos problemas podem ser fraseados dessa forma (aplicações que processam largas quantias, blah blah blah)

Facil de distribuir pelos nodes

Boas retry/failure semantics

Existem duas implementações:

- **Hadoop MapReduce**

- O Mapper e o Reducer são ambos Classes Java que implementam uma interface particular

- **MongoDB e CouchDB**

- Mappers e Reducres são funções de Javascript

○ Hadoop é uma implementação do MapReduce



- Doug Cutting and Mike Cafarella created the Apache Nutch Web crawler (2002)
- Doug used MapReduce in Nutch (2006)
- Became **Hadoop** (name that Doug's kid gave a stuffed yellow elephant toy) @ Yahoo
- Hadoop has evolved into an open source ecosystem for handling and analyzing Big Data (2007)

Img 1.1 – Logo do Hadoop e Historia

É uma técnica de **Batch Processing Distribuido**:

- **Reads e Writes** de ficheiros num filesystem distribuido
- Implementação de Hadoop sobre **HDFS** – Hadoop Distributed FileSystem suporta integrações (Cassandra como Data Input)

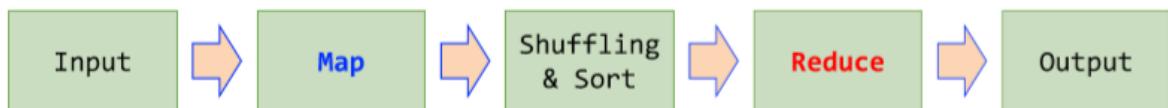
Foi desenhado para **Large Scale Data Processing**, sendo que procura utilizar 1000s de CPUs mas como management simplificado

More on this later in this chapter

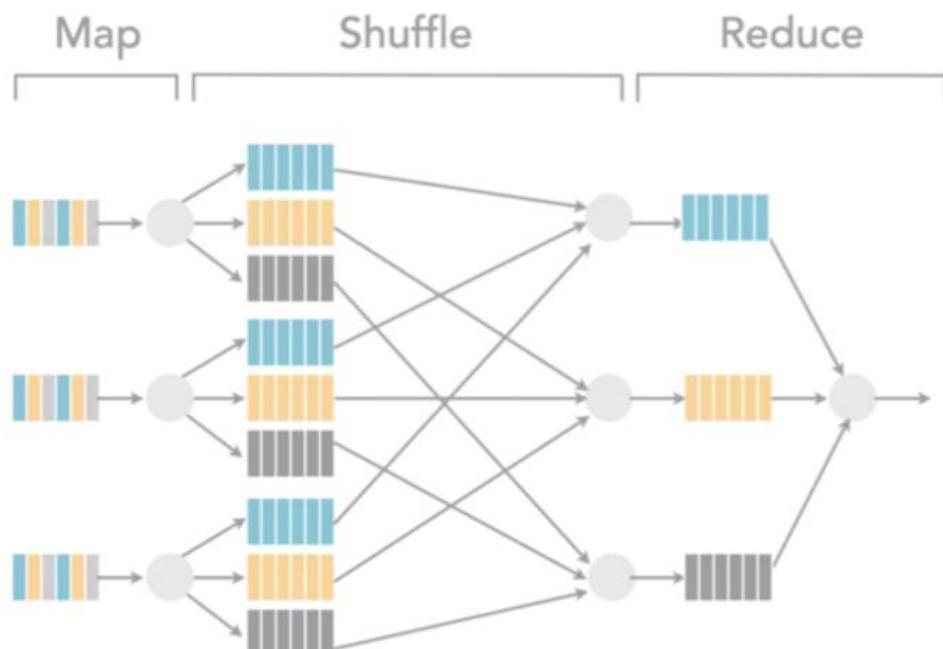
II. Mapreduce DataFlow

Em MapReduce os dados fluem da seguinte forma:

1. Ficheiros/Documentos de Input são partidos em **records**
2. **Map Function** é chamada **uma vez por** cada input **record**
 - a. Vai ter a função de **extrair** (um ou mais) **key-values** do input record
3. A Framework MapReduce **coleciona** todos os Key-Value **pairs com a mesma key**
 - a. É feita uma Sort operations por Key
4. A **Reduce Function** é um iterador sobre a coleção de valores com a mesma key e vai produzir **output records**
 - a. Outputs podem, p.ex, ser o numero de ocorrências da mesma key



Img 2.1 – Dataflow



Img 2.2 – Visualização das Map e Reduce Functions

III. Mapreduce Framework

No MapReduce vamos ter dois componentes principais:

- **Mappers**
 - Processam os dados num dado node de uma cluster
- **Reducers**
 - Pegam nos resultados produzidos pelos mappers e combinam-nos como necessário para resolver o problema em questão

A MapReduce Framework vai ter que **escalonar, monitorizar e re-executar failed tasks/components** (mappings e reduces)

A pipeline das Main Tasks é então:

1. Dividir o data-set de input em chunks independentes
2. Mapear processos de chunks de forma paralela
3. Dar sort dos outputs dos maps
4. Aplicar o Reduce Process aos sorted Mapper Outputs

IV. Programming Model

Vamos ter como input e output um conjunto de Key/Value Pairs

Programaticamente vamos possuir duas funções principais:

- **map(in_key, in_value)**
 - Retorna **list(out_key, intermediate_value)**
 - Processa o input key/value pair
 - Produz um set de pares intermediarios
 - Normalmente com de um dominio diferente
 - Keys não têm de ser unicas
- **reduce(out_key, list(intermediate_value))**
 - Retorna **list(out_value)**

- Combina todos os intermediate values para uma particular key
- Produz um set de merged output values (normalmente, só um)
 - Do mesmo domínio

Vamos ver o seguinte exemplo – **Word Count** (fizemos esta merda em Computação Distribuída)

❖ Input

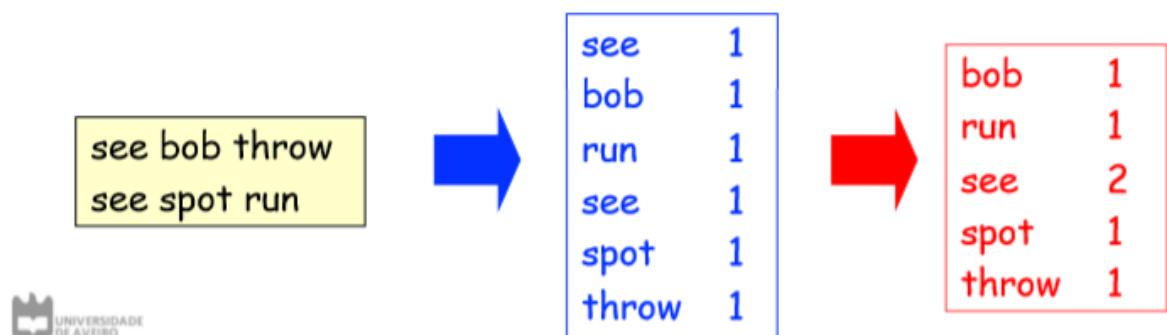
- (url, contents) pairs

❖ **map**(key=url, val=contents):

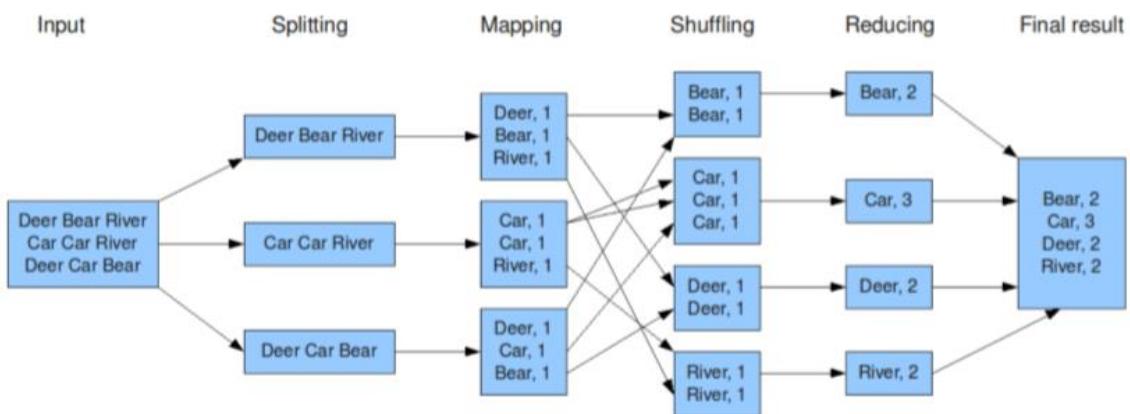
- for each word w in contents, emit (w, “1”)

❖ **reduce**(key=word, values=uniq_counts):

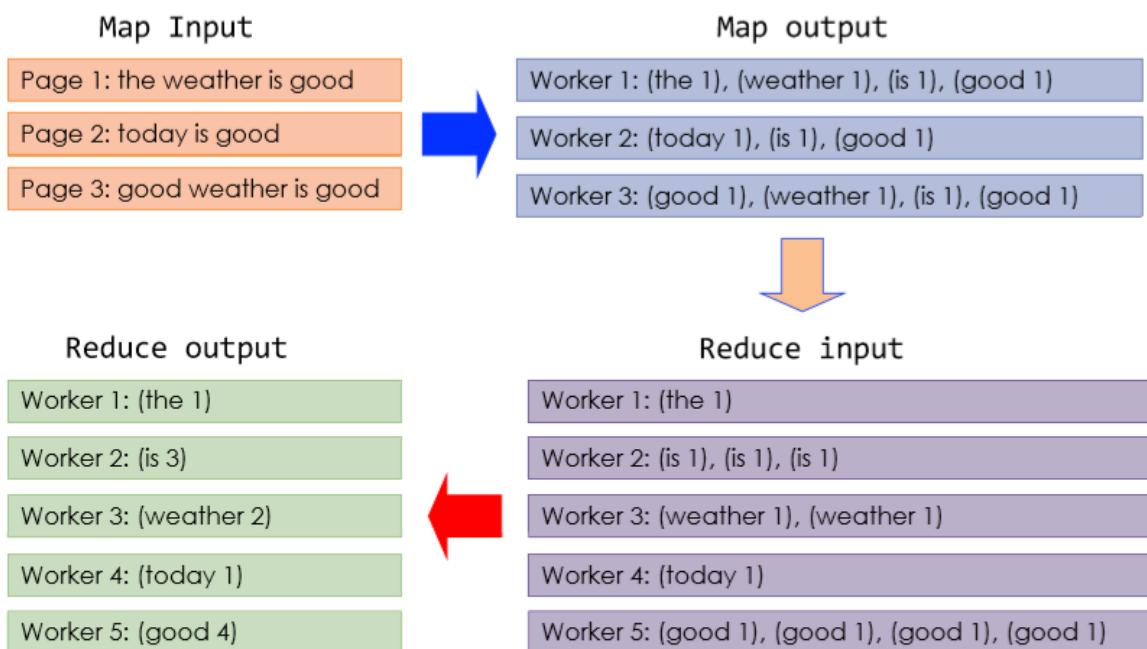
- sum all “1”s in values list
- emit result “(word, sum)”



Img 4.1 Exemplo dos inputs, e funcionamento de cada função



Img 4.2 Exemplo do



Img 4.3 Exemplo da ordem da execução das tasks

V. Aplicações do MapReduce

O MapReduce pode ser aplicado a diversos tipos de aplicações

- ❖ words count / histogram
- ❖ distributed grep
- ❖ distributed sort
- ❖ web link-graph reversal
- ❖ term-vector per host
- ❖ web access log stats
- ❖ inverted index construction
- ❖ document clustering
- ❖ machine learning
- ❖ statistical machine translation



...

Img 5.1 Exemplo da aplicações do MapReduce

Vejamos mais dois exemplos:

- **Distributed Grep com Map-Reduce**

- ❖ In Unix:

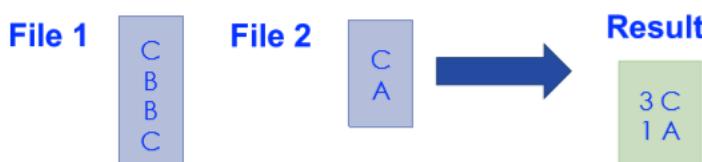
```
> grep -Eh <regex> <inDir>/* | sort | uniq -c | sort -nr
```

- counts lines in all files in <inDir> that match <regex> and displays the counts in descending order

- ❖ Usage Example

- Analyzing web server access logs to find the top requested pages that match a given pattern

```
> grep -Eh 'A|C' in/* | sort | uniq -c | sort -nr
```



Img 5.2 Funcionamento do GREP em Unix

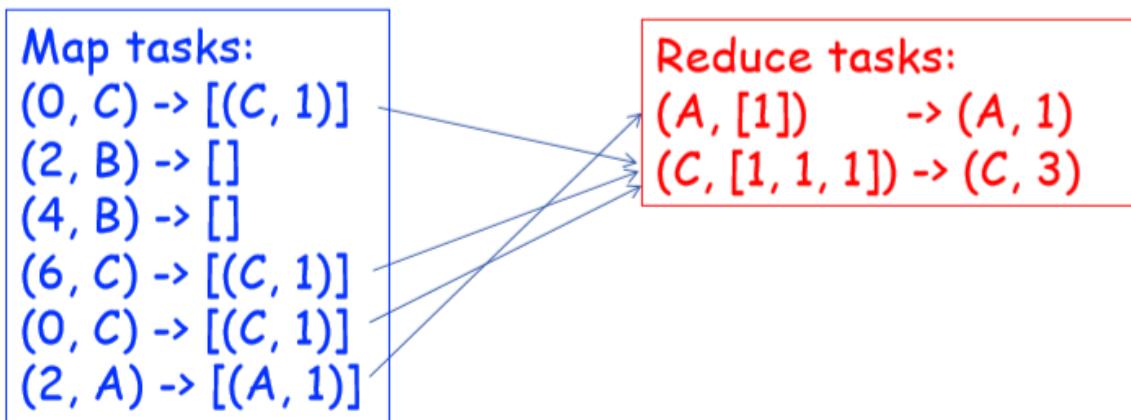
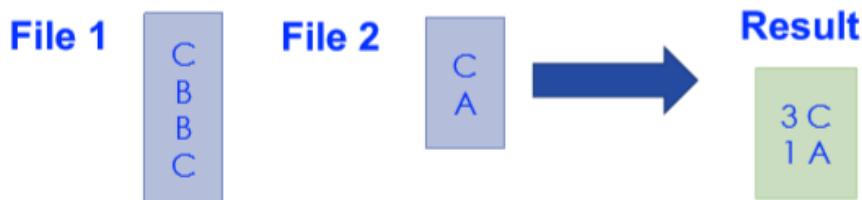
❖ Map function

- input
(file offset, line)
- output is either
 - 1. empty list [] (if line does not match)
 - 2. key-value pair [(line, 1)] (if line matches)

❖ Reduce function

- input
(line, [1, 1, ...])
- output
(line, n) (n is the number of 1s in the list)

Img 5.3 - Como implementar GREP com MapReduce



Img 5.4 – Exemplo de execução

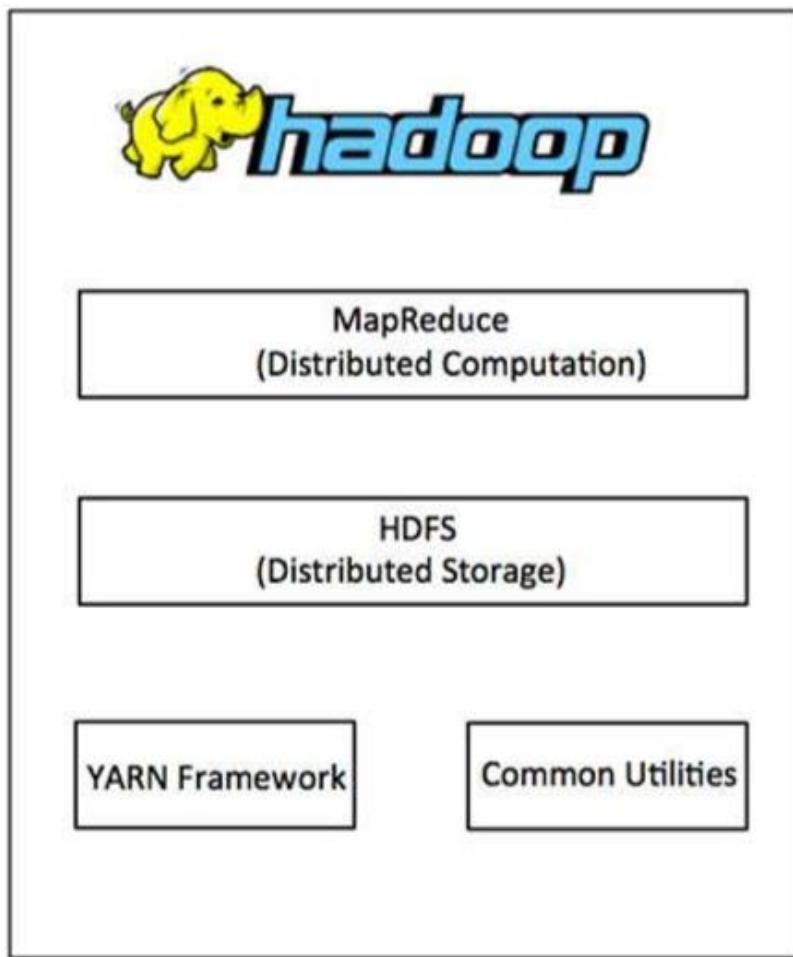
- **Geração de PDFs em Larga-Escala**

- O New York Times precisa de gerar ficherios PDF de 11.000.000 artigos (todos os artigos de 1851 até 1980) sob a forma de imagens scanadas do papel original
- Cada artigo é composto por inumeras TIFF Images que são escaladas e coladas juntas
- The Problem: **Precisamos de Código para gerar o PDF de forma straightforward**
- Para resolvemos isto vamos usar:
 - **Amazon Simple Storage Service – S3**
 - Internet storage escalavel e barato que pode guardar e ir buscar quaisquer quantidades de dados a qualquer altura de qualquer local da web
 - Sistema assincrono e decentralizado que procura reduzir scaling bottlenecks com unicos points of failure
 - **Amazon Elastic Compute Cloud – EC2**
 - Virtualização do Computing Environment desenhado para ser usado com outros serviços da Amazon
 - **Hadoop**
 - Open Source implementation do MapReduce
- Implementando estas tecnologias vamos obter os resultados:
 - 4 TB de artigos scanados enviados para o S3
 - Cluster de EC2 Machines configuradas para distribuir a geração de PDF via Hadoop
 - Usando 100 EC2 Instances e 24 horas, o NYT conseguiu converter 4 TB de artigos em 1.5 TB de documentos pdf

VI. Apache Hadoop

Voltando ao Hadoop, esta é uma **Open Source Framework** feita em Java cujos componentes são:

- **Hadoop Common**
- **Hadoop Distributed File System – HDFS**
 - File System distribuído, escalável e portável
- **Hadoop Yet Another Resource Negotiator – YARN**
- **Hadoop MapReduce**
 - Implementação de um MapReduce Programming Model



Img 6.1 – Componentes do Hadoop

VI.I Overview

Trabalhos dos Inputs e Outputs estão normalmente guardados no **HDFS** – Um Shared File System

O HDFS é distribuído pelos nodes numa cluster.

É fornecida uma unified interface para distributed files

Fornece **Fault Tolerance**

- HDFS distribui multiplas copias dos data files para dentro de diferentes nodes

MapReduce consegue escalar processos de acordo com a distribuição de dados pelos nodes

Isto minimiza os network data movements!

VI.II Basic Commands

❖ Basic help for all the commands

> `hadoop`

❖ Distributed file system commands

> `hadoop fs`

❖ Execution of MapReduce jobs

> `hadoop jar`

VI.III HDFS Commands

- ❖ Help
 - > hadoop fs -help
- ❖ Creating a directory
 - > hadoop fs -mkdir /myhdfsdir
- ❖ Listing a directory
 - > hadoop fs -ls
 - > hadoop fs -ls /myhdfsdir
- ❖ Copy a file/dir from local file system to HDFS
 - > hadoop fs -put /myhdfsdir/ /user/xpto/
- ❖ Get a file/dir from HDFS to local file system
 - > hadoop fs -get /user/xpto/ /myhdfsdir/
- ❖ more commands ...



https://www.tutorialspoint.com/hadoop/hadoop_command_reference.htm

VI.IV Compilação do Java Program

- ❖ Compile the JAVA program
 - > mkdir classes
 - > javac -classpath
/usr/local/hadoop/share/hadoop/common/
hadoop-common-2.9.0.jar:
/usr/local/hadoop/share/hadoop/mapreduce/
hadoop-mapreduce-client-core-2.9.0.jar
-d classes/ MyApp.java
 - > jar -cvf MyApp.jar -C classes/ .

VI.V Execução de um MapReduce Job

❖ Run the job

```
> hadoop jar WordCount.jar  
/myhdfsdir/myappdir/input1  
/myhdfsdir/myappdir/output1
```

❖ Retrieve and explore the job result

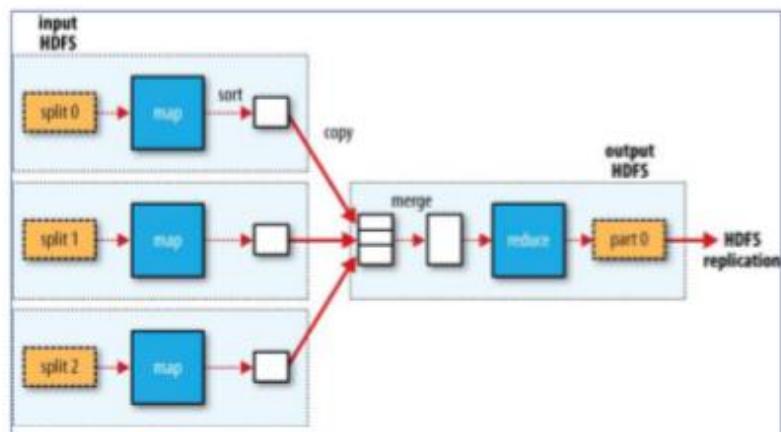
```
> hadoop fs -copyToLocal  
/myhdfsdir/myappdir/output1/part-r-00000 result.txt  
> cat result.txt
```

❖ Clean the output HDFS directory

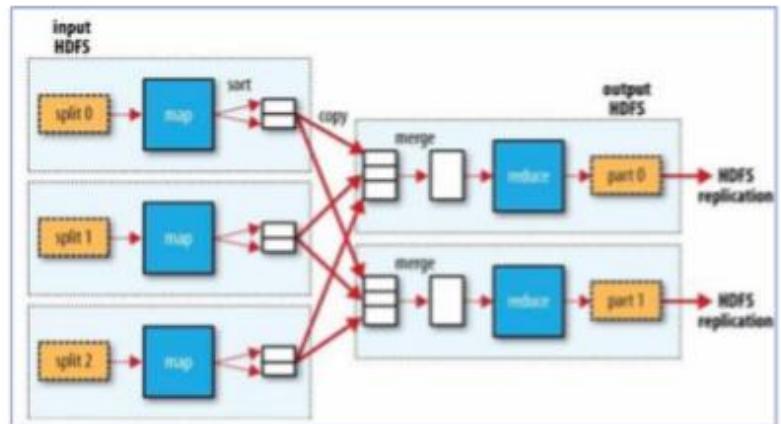
```
> hadoop fs -rmr /myhdfsdir/myappdir/output1/
```

VI.VI Task Execution - Configurações

single reduce task



multiple reduce task



VI.VII Number of Mappers e Reducers

- **Numero de Mappers**
 - Tipicamente controlado pelo YARN
 - **Um mapper por bloco HDFS**
 - Pelo menos um por cada node que contenha dados
 - P.ex
 - File Size = 500 mB ; HDFS File Size = 50 mB
 - #Blocks (physical storage) usado no HDFS = $500/50 = 10$ blocos = 10 Mappers
- **Numero de Reducers**
 - Tipicamente controlado pelo User
 - Default = 1
 - Executa em apenas alguns nodes
 - Numero ideal = 0.95 ou $1.75 * \#nodes * \#maximo\ de\ containers\ por\ node$

Num caso extremo Podemos ter 1 Mapper e 1 Reducer. Neste caso não haverão processos distribuidos

❖ Example

- 5 Mappers
- 2 Reducers

❖ Option 1: command line

-D mapred.map.tasks=5 -D mapred.reduce.tasks=2

❖ Option 2: Java code

```
job.setNumMapTasks(5);  
job.setNumReduceTasks(2);
```

VI.VIII Project Creation com IDE

O exemplo que vamos apresentar assume que se utiliza o Eclipse ou Netbeans como IDE

1. Create a new Java project
2. Make local copies requested Hadoop libraries*:
`/usr/local/hadoop/share/hadoop/common/hadoop-common-2.9.0.jar`
`/usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.9.0.jar`
3. Add both libraries to the project
4. Create the Java application with Map and Reduce functions
5. Build the project and create a jar distribution

❖ Mapper class

- Implementation of the **map function**
- Template parameters
 - KEYIN, VALUEIN** – types of input key-value pairs
 - KEYOUT, VALUEOUT** – types of intermediate key-value pairs
- Intermediate pairs are emitted via **context.write(k, v)**

```
class MyMapper extends Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {  
    @Override  
    public void map(KEYIN key, VALUEIN value, Context context)  
        throws IOException, InterruptedException  
    {  
        // Implementation  
    }  
}
```

❖ Reducer class

- Implementation of the **reduce function**
- Template parameters
 - KEYIN, VALUEIN** – types of intermediate key-value pairs
 - KEYOUT, VALUEOUT** – types of output key-value pairs
- Output pairs are emitted via **context.write(k, v)**

```
class MyReducer extends Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {  
    @Override  
    public void reduce(KEYIN key, Iterable<VALUEIN> values, Context context)  
        throws IOException, InterruptedException  
    {  
        // Implementation  
    }  
}
```

VI.IX Wordcount Example

```
public class WordCount extends Configured implements Tool {  
    private static IntWritable ONE = new IntWritable(1);  
    @Override  
    public int run(String[] args) throws Exception {  
        FileSystem fs = FileSystem.get(getConf());  
        Job job = Job.getInstance(getConf());  
        job.setJarByClass(WordCount.class);  
        job.setJobName("WordCount");  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(IntWritable.class);  
        job.setMapperClass(Map.class); ← Mapper class  
        job.setCombinerClass(Reduce.class);  
        job.setReducerClass(Reduce.class); ← Reducer class  
        job.setInputFormatClass(TextInputFormat.class);  
        job.setOutputFormatClass(TextOutputFormat.class);  
        FileInputFormat.setInputPaths(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
        if (fs.exists(new Path(args[1])))  
            fs.delete(new Path(args[1]), true);  
        boolean success = job.waitForCompletion(true);  
        return success ? 0 : 1;  
    }  
}
```

Img 6.9.1 – Configuração

```

...
    public static void main(String[] args) throws Exception {
        int ret = ToolRunner.run(new WordCount(), args);
        System.exit(ret);
    }

    public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
        @Override
        public void map(LongWritable key, Text value, Context context) throws
                IOException, InterruptedException {
            String line = value.toString();
            String[] words = line.split("[\t\n.,;?!-/\\[\\]\\\"']+");
            for (String word : words) {
                if (word.trim().length() > 0) {
                    Text text = new Text();
                    text.set(word);
                    context.write(text, ONE);
                }
            }
        }
    } // end class Map

```

Img 6.9.2 – Mapper Class

```

...
    public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
                throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values)
                sum += val.get();
            context.write(key, new IntWritable(sum));
        }
    } // end class Reduce
} // end class WordCount

```

Img 6.9.3 – Reducer Class

```
Compile
> $ javac -classpath $CLASSPATH -d WordCountDir WordCount.java

Create JAR
> $ jar -cvf WordCount.jar -C WordCountDir/ .

Run WordCount App
> $ hadoop jar WordCount.jar /in/test.txt /out/wc

See the Result
> $ hadoop fs -cat /out/wc/part-r-00000

ABOUT 1
ACCOUNT 2
ACTUAL 1
ADDITIONAL 1
ADVANCING 2
...
```

Img 6.9.3 – Resultados



All work and no play makes Jack a dull boy