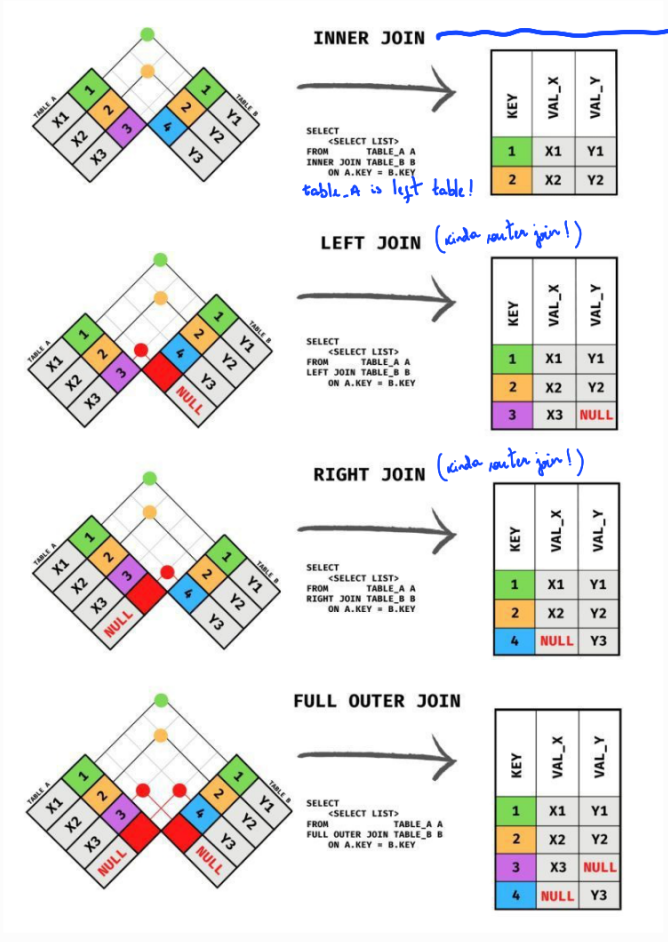
Data Eng. Roadmap

PKey: Primary Key
FKey: Foreign Key

☑ Advanced Joins SQL:



→ PKeys in tables that are FKeys on another ones.

SQL

* Alias: apelidos de tabelas

E.g.: FROM BOOK B WHERE B.BOOK_ID ...

- Foreign Keys: point to primary keys on another table.
- Primary Keys: uniquely identifies an entity/table

Data Modelling

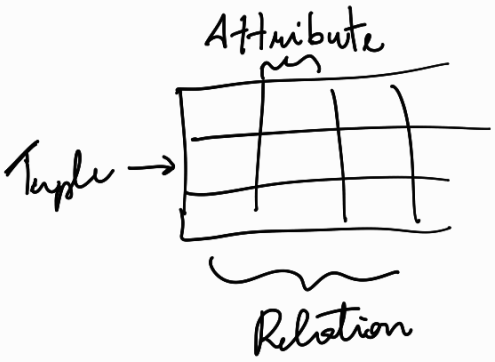
Data Eng with AWS

- Data modelling = Database modelling
- An "abstract process"

Process :

1. Gather requirements
2. Conceptual modelling (entity mapping)
3. Logical " (tables, schemas, columns)
4. Physical "
   ↳ DDL (Data Def. Language)

Relational Model



SQL - Structured Query language

- **Schema** : collection of Tables.
- The RM and relational db's were created in 60' / 70's.

Attribute → Column
(feature)

Tuple → Row

- When to use Relational DB (RDB) ?

- JOINs só podem ser feitos em RDBs
- Smaller data volumes
- Secondary Index for quick searching
  (Primary index is the primary key)
- ACID Transactions
  (Transactions são operações ACID feitas e percebidas como uma única operação)

ACID — Atomicity: each transaction as a single "unit", which either succeeds or fails completely. In fail, the db is left unchanged; rolls back to prior state.
Isolation: Concurrent transactions leave db in the same state that would have been obtained if the transactions were executed sequentially; not impacted by other transactions.
Consistency: Data integrity; recall constraints, rules, etc.

OBS: JOIN só podem ser feitos se existirem valores que dão match nas duas tabelas!

- When NOT to use RDBs ?

  . Large amounts of data

  . Quando preciso armazenar dados de diferentes formatos
    ↳ RDBs só escalam verticalmente, i.e., adicionando mais memória na máquina; NÃO É DISTRIBUÍDO!

  . Qdo necessitar de leitura rápida
    ↳ ACID torna mais lento ↓

  . Qdo precisar escalar horizontalmente
    ↳ add mais nodes/machines (melhor com NoSQL)

  . " " de alta disponibilidade

NoSQL Databases

- Simpler horizontal scaling
- Different types

. More suitable when:
- Need high availability
- " to scale fast quickly
- large amounts of data
- different data types
- high throughputs: faster reading/writing

- Partition Row store
- Document store
- key-value
- wide column ("flexible schema")

→ Apache Cassandra

- Keyspace ⇓ collection of tables
- tables ⇓ groups of partitions
- rows

- Uses its own query language: CQL

~ // ~

Relational Data Models

- OLAP (analytical processing)
  - ↳ more aggregations ↳ read-only
- OLTP (transactional processing)

⇒ Normalization & Denormalization

- Normalization
  - ↳ reduce data redundancy and increase data integrity
  - ↳ reduce copies of data

- Denorm
  - ↳ must be done to increase performance in read heavy workloads

⇒ Objetivos of Normal Form

1. Free db's from unwanted insertions, updates and deletion dependencies
2. Reduzir necessidade de restruturação do novos tipos de dados são introduzidos
3. Make more information
4. Tornar o DB neutro às queries (ñ fazem o DB p/ a query, como é no caso de NoSQL)

1NF

- Unique values on each cell (atomicity)

2NF

- All columns in the **normalized** table must depend on a **Primary Key**.

- No "partial dependencies": só com composite keys!
- Partial depend. é qdo uma non-key column depende da chave composta parcialmente; depende só de 1 campo da mesma.

3NF

- Eliminate columns that don't depend on a Primary Key
- No transitive dependencies (A → C, avoiding going through B)

- Transitive Dependency: Um campo C depende de A (chave primária), mas esse mesmo campo C também depende de (pode ser determinado por) outro B (que NÃO é chave primária)

No SQL Data Models

NoSQL = Not only SQL

- High availability → copies of data
  - → Distributed Databases

availability: make request and get a response.

VOCAB!

* stale data = dados obsoletos
* retailer = varejista
  retail = varejo

CAP Theorem

↳ Impossible for a distributed data store to simultaneously provide more than 2 of the following guarantees:

- Consistency ⇓ Every read from the db gets the latest piece of data or an error.
- Availability
- Partition Tolerance ⇓ System continues to work regardless of losing network between nodes.

- Apache Cassandra is an AP (Availability, Partition Tolerance) db.

- **Eventual Consistency in NoSQL vs Consistency in ACID SQL db's**

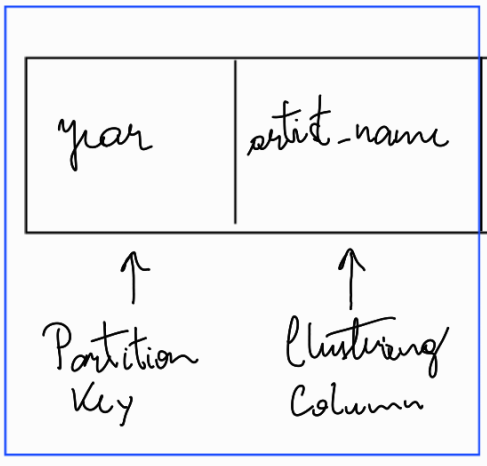
→ ACID: only transactions that obbey (abide) by constraints and db rules are committed into the db, otherwise db keeps previous state.

→ CAP: cf. definition above!

- Availability and Part. Tolerance are the biggest wins.

- **Denormalization** is a MUST in Cassandra!
  - ↳ Think queries first!
  - ↳ No JOINs!
  - ↳ 1 query per table!

- **Primary Keys** (PK)
  - First element of the PK is the PARTITION KEY
  - PK made up of either just PART. KEY or with the addition of CLUSTERING COLUMNS
  - PART. KEY will determine distribution of data across the system.

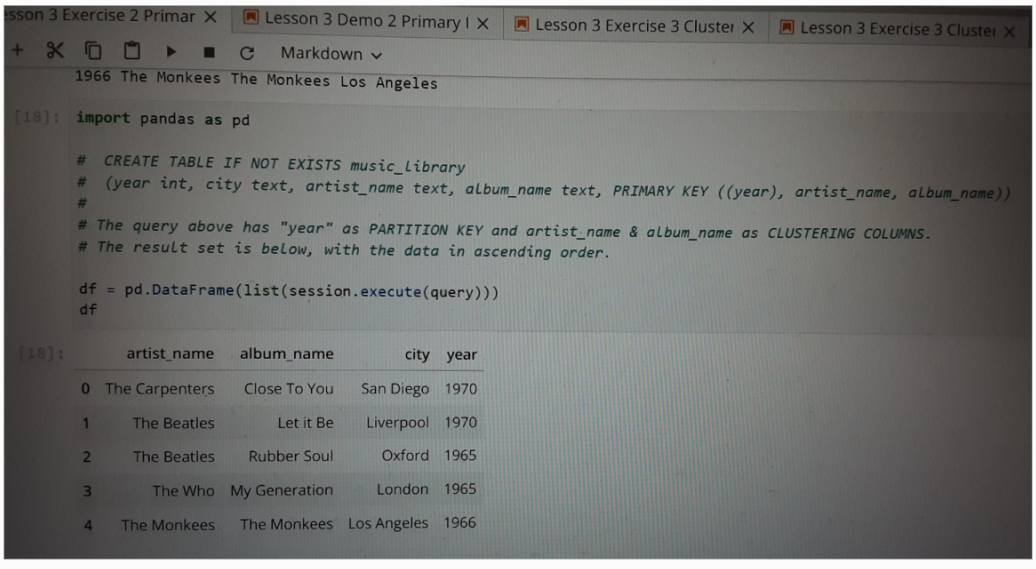



```
CREATE TABLE music_lib
(year int, artist_name text,
album_name text,
PRIMARY KEY ((year),
artist_name, album_name))
```

OBS: If PK are wrapped in "()", then the PK is composite! Otherwise, the 1st field is the Partition Key.

- Clustering columns will sort data in ASC order
- More than one column can be added.
- Clustering columns will sort in order of how they were added to the primary key.

↳ See example of query in table with PART. KEY and CLUSTERING COL. below:



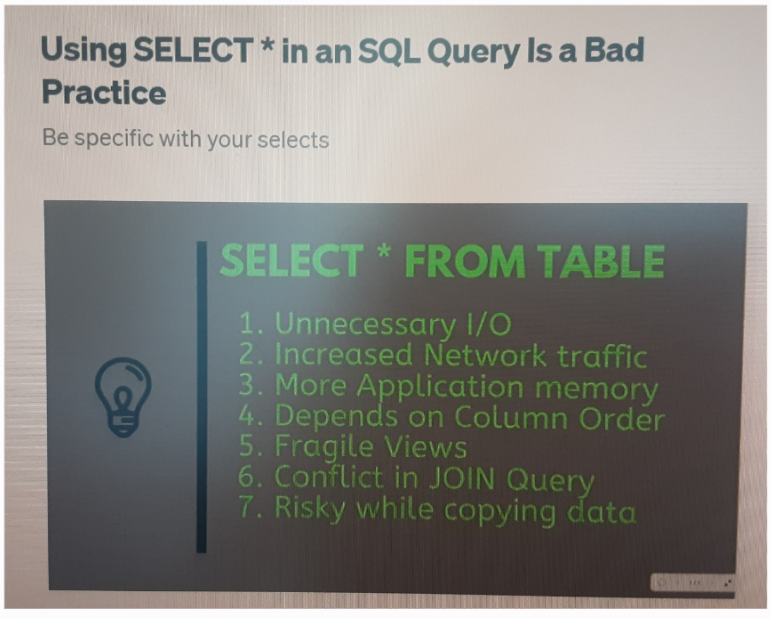
- Na query, a PK DEVE ser utilizada! Além disso, as colunas de clustering (CLUSTERING COLS.) devem ser usadas na ordem em que foram colocadas na PRIMARY KEY.

Important! - CREATE and INSERT statements must abide the order of the PK fields. Example below. FOR CASSANDRA!

Ex.: If in CREATE the PK is "PRIMARY KEY (id, orderNum)" the INSERT must be:

> INSERT INTO <table> (id, orderNum, orderBuid, ...)
VALUES (1, 999, 11, ...)

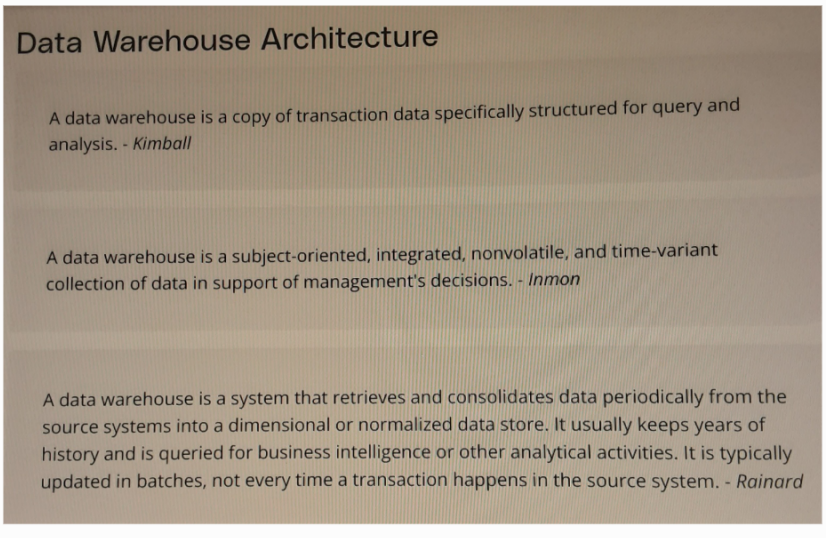
ordered according to the PK



→ Why it's a bad idea to use "SELECT *"

Data Warehouses in the Cloud

17/04 | 23

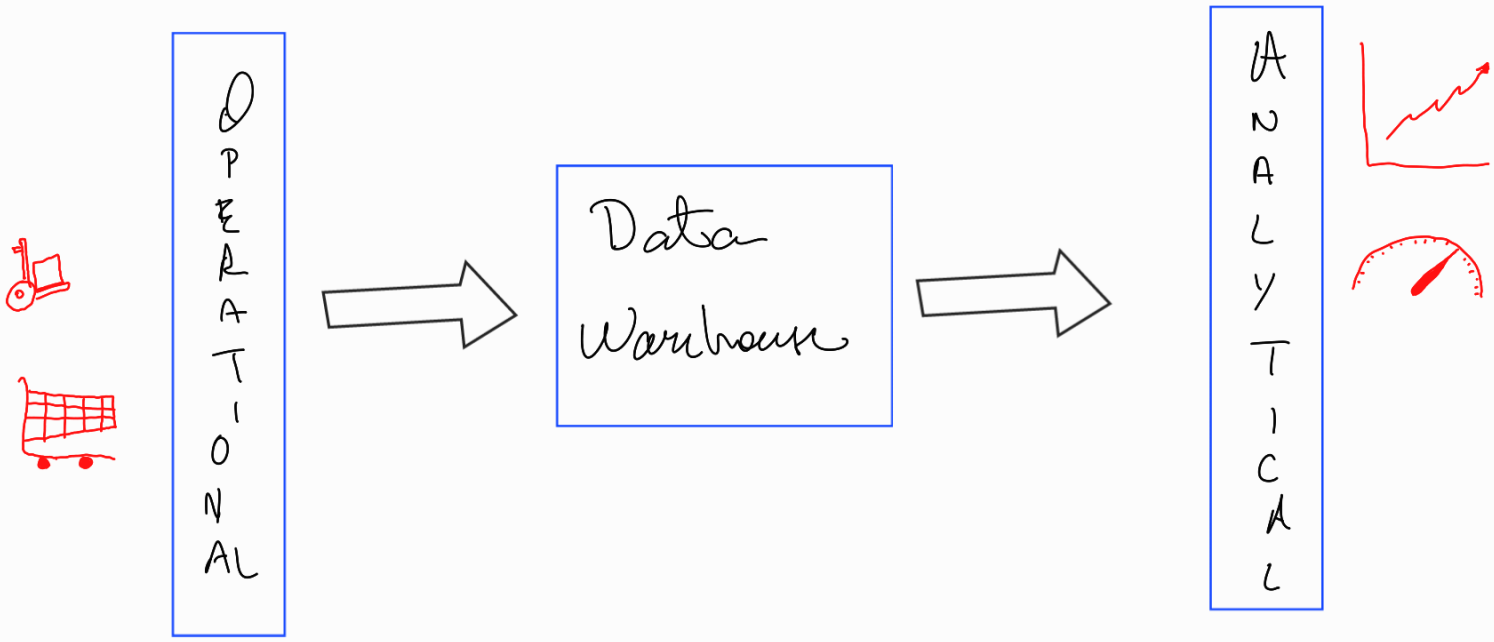


OLTP vs OLAP

Online Transactional P.
vs Analytical P.

- Operational processes: make it work
  1. Find goods and move orders
  2. Stock and find goods
  3. Pick up and deliver goods

- Analytical : what is going on?
  1. Assess performance of sales staff
  2. See the effect of different sales channels
  3. Monitor sales growth.



Ex.: track something in a retail is OLTP, but examine a market segmentation over a period of time is OLAP.

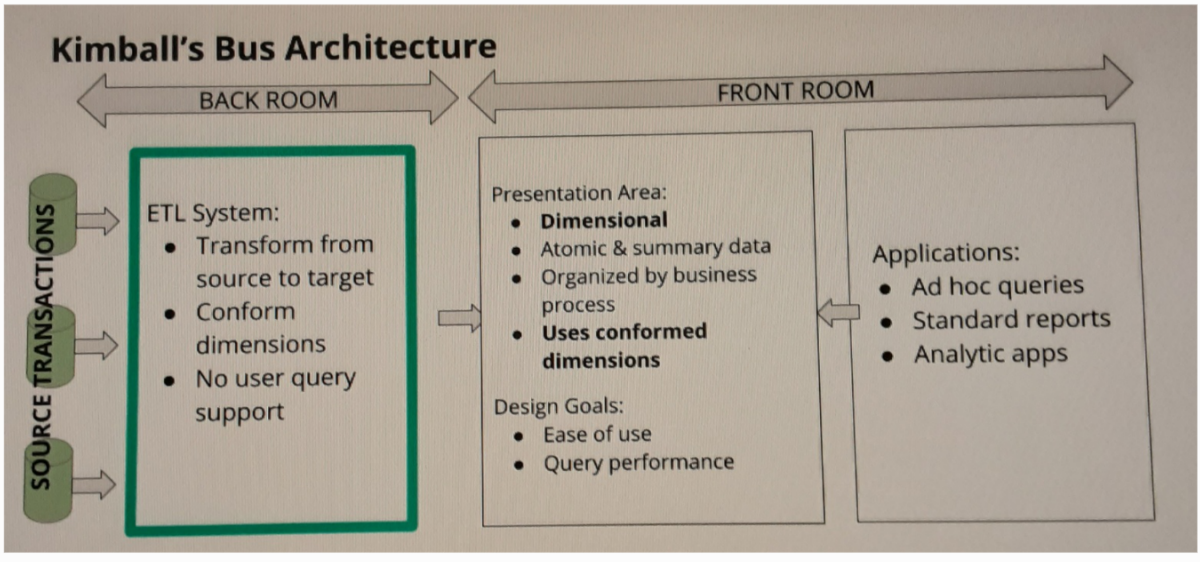
OBS: If a database is small, it may be fine for both OLTP and OLAP.

If big: queries slow; schema hard to understand!

→ Data is loaded in a DW in batches, not after very operation of a transactional system.

(?) No streaming?

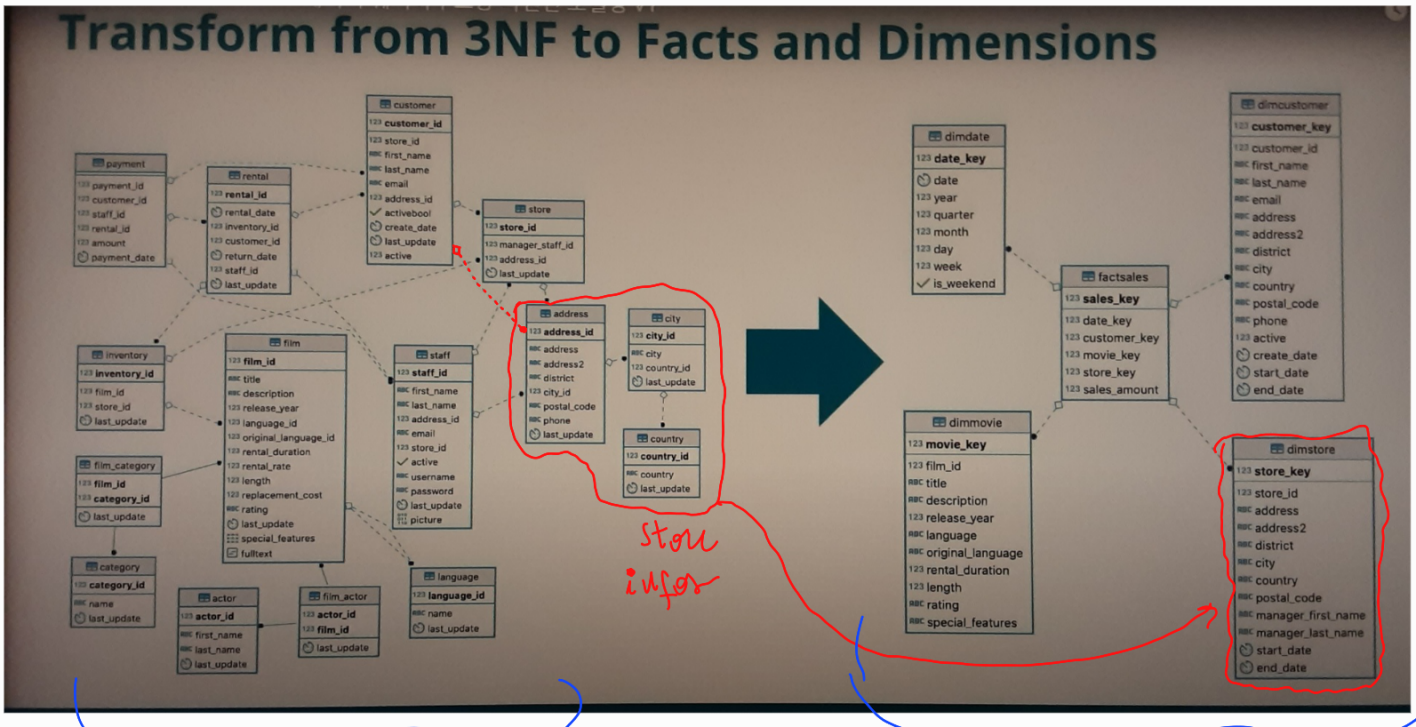
→ Data structured in a Dimensional Modeling, easy to query.



Kimball's Bus Arch:

1. Results in common data model shared by different departments
2. Data not kept at aggregated level, rather at atomic level
3. Organized by business processes, and used by different departments.

Identifying Facts and Dimensions



difficult for a business user understand

easier for a business user to understand

↝ From 3NF to ETL:

Extract

- query 3NF db

Transform

- join tables
- change types
- add new columns

Load

- insert into facts | dimensions

Facts

- Business events
- Quantifiable data
- Numeric data
- "Fato"

Examples: nº items bought by a customer;
instances of positive feedback;
sales amount;
length of phone call.

Dimensions

- Context of business events (dates, times, etc)
- Ex.: store where item was purchased, the customer, etc.

Examples: tipo de compra;
tipos de móveis buscados online;
local loja;
employee ID, etc.

OBS "Dimensional Modeling" exercise and "ETL" exercise → using "SERIAL" while creating fact/dimension tables in PostgreSQL will generate

an auto-incrementing number once records are inserted!

Grouping Sets [SQL]

SELECT ... FROM ... JOIN ...

GROUP BY grouping sets (f1, f2, (f1, f2));

See the example below using the F/Dim tables already shown before (cf. "Identifying Facts & Dimension tables"):

total revenue

```
%%sql
SELECT sum(sales_amount) as revenue
FROM factSales
```

revenue by country

```
%%sql
SELECT dimStore.country,sum(sales_amount) as revenue
FROM factSales
JOIN dimStore on (dimStore.store_key = factSales.store_key)
GROUP by  dimStore.country
order by dimStore.country, revenue desc;
```

```
 * postgresql://student:***@127.0.0.1:5432/pagila_star
2 rows affected.
```

country	revenue
Australia	33726.77
Canada	33689.74

revenue by month

```
%%sql
SELECT dimDate.month,sum(sales_amount) as revenue
FROM factSales
JOIN dimDate     on (dimDate.date_key         = factSales.date_key)
GROUP by dimDate.month
order by dimDate.month, revenue desc;
```

```
 * postgresql://student:***@127.0.0.1:5432/pagila_star
5 rows affected.
```

month	revenue
1	4824.43
2	9631.88
3	23886.56
4	28559.46
5	514.18

revenue by month & country

```
%%sql
SELECT dimDate.month,dimStore.country,sum(sales_amount) as revenue
FROM factSales
JOIN dimDate     on (dimDate.date_key         = factSales.date_key)
JOIN dimStore on (dimStore.store_key = factSales.store_key)
GROUP by (dimDate.month, dimStore.country)
order by dimDate.month, dimStore.country, revenue desc;
```

```
 * postgresql://student:***@127.0.0.1:5432/pagila_star
10 rows affected.
```

month	country	revenue
1	Australia	2364.19
1	Canada	2460.24
2	Australia	4895.10
2	Canada	4736.78
3	Australia	12060.33
3	Canada	11826.23
4	Australia	14136.07
4	Canada	14423.39
5	Australia	271.08
5	Canada	243.10

revenue total, by month, by country, by month & country All in one shot

- watch the nones

```
%%time
%%sql
SELECT dimDate.month,dimStore.country,sum(sales_amount) as revenue
FROM factSales
JOIN dimDate  on (dimDate.date_key   = factSales.date_key)
JOIN dimStore on (dimStore.store_key = factSales.store_key)
GROUP by grouping sets ((), dimDate.month,  dimStore.country, (dimDate.month,  dimStore.country));
```

```
 * postgresql://student:***@127.0.0.1:5432/pagila_star
18 rows affected.
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 27.1 ms
```

month	country	revenue
1	Australia	2364.19
1	Canada	2460.24
1	None	4824.43
2	Australia	4895.10
2	Canada	4736.78
2	None	9631.88
3	Australia	12060.33
3	Canada	11826.23
3	None	23886.56
4	Australia	14136.07
4	Canada	14423.39
4	None	28559.46
5	Australia	271.08
5	Canada	243.10
5	None	514.18
None	None	67416.51
None	Australia	33726.77
None	Canada	33689.74

→ "month & country" grouping

→ "months" grouping

→ likewise

→ likewise

sum of 2364,19 & 2460,24 above.

"( )", i.e., the total revenue regardless month & country

→ "country" grouping

☒ Data Warehouses in the CLOUD

- Why moving to cloud DW's ? = why ETL to ELT ?
  - Scalability : large amounts of data.
  - Flexibility : adding/removing data is easier.
  - Cost shifting : "transform" step (more expensive/costly) is done last, so DEngs can perform Just-in-time transformations to meet the highest priority business needs first.

ETL vs ELT

ETL
- Transformation happens on an intermediate server
- Data loaded into destination using transformed tables.

ELT
- Transformation happens in the destination server
- Data loaded ... using either raw data or staging tables.

→ ## No SQL Databases in AWS

- DynamoDB - key value
- Document DB - document
- Keyspaces - column oriented
- Neptune - graph
- Time stream - time series

AWS Data Warehouse Technologies

→ All AWS services require Identity and Access Management (IAM) policies, roles, and their associated permissions

→ **IAM User**: is an identity with long-term credentials that is used to interact with AWS in an account.

→ **Security Groups**: act as a firewall rules for e.g. a Redshift cluster to control inbound and outbound traffic.

ARN
Amazon Resource Name

Ps.: Users are different from Roles.

- Person / Application
- Uniquely associated
- Long-term credentials

- Assumable by anyone who needs
- Temporary credentials

→ **Federated Access**: external access to AWS resources

Benefits of Infrastructure as Code

- IAC é o provisionamento de infraestrutura de TI via linguagem de codificação descritiva de alto nível (e.g. json, yaml, etc).

Benefícios da IaC

Agilidade: A capacidade de automação da IaC acelera o processo de provisionamento de uma infraestrutura para desenvolver, testar e produzir aplicações. Permitindo a configuração de uma infraestrutura completa executando apenas um script. Assim possuindo os mesmos princípios seguidos na cultura DevOps no quesito da velocidade e consistência do ciclo de vida de entrega do projeto.

Consistência: Com a automatização de processos oriundos da IaC, falhas e discrepâncias que seriam criadas por um processo manual são praticamente eliminadas. A infraestrutura como código tem a capacidade de evitar esses possíveis problemas, pois seus arquivos de configuração possuem apenas uma única fonte de informação, assim garantindo a possibilidade de realizar repetidas implantações de forma consistente e sem disparidade de informações.

Segurança: Assim como todos os serviços de cloud computing, a IaC traz mais segurança para o ambiente de TI da sua empresa. As ferramentas de infraestrutura como código permitem a correção rápida de erros e a solução de problemas de forma automatizada, deixando assim uma infraestrutura mais segura e gerenciada pela empresa.

Quiz Question

Which of the following are advantages of Infrastructure-as-Code over creating infrastructure by clicking-around?

- [x] Sharing: One can share all the steps with others easily
- [x] Reproducibility: One can be sure that no steps are forgotten
- [x] Multiple deployments: One can create a test environment identical to the production environment
- [x] Maintainability: If a change is needed, one can keep track of the changes by comparing the code

↘ all are correct!

Common Terms in AWS

Terms

In the previous illustration we used specific terminology to describe how to obtain access to resources. These IAM terms are commonly used when working with AWS:

IAM Resources

The user, group, role, policy, and identity provider objects that are stored in IAM. As with other AWS services, you can add, edit, and remove resources from IAM.

IAM Identities

The IAM resource objects that are used to identify and group. You can attach a policy to an IAM identity. These include users, groups, and roles.

Principals

A person or application that uses the AWS account root user, an IAM user, or an IAM role to sign in and make requests to AWS. Principals include federated users and assumed roles.

Human users

Also known as *human identities*; the people, administrators, developers, operators, and consumers of your applications.

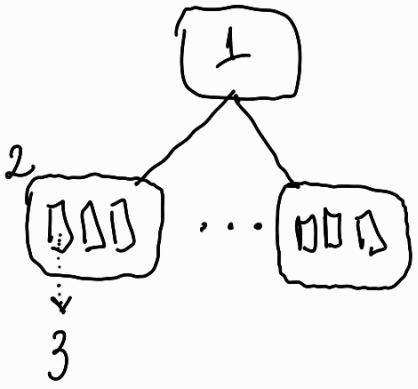
Workload

A collection of resources and code that delivers business value, such as an application or backend process. Can include applications, operational tools, and components.

Implementing a DW on AWS

Redshift Arch.

1. Leader Node
2. Compute nodes
3. Node slices



Important!

- Total # nodes in Redshift = # AWS EC2 instances used in the cluster
- slices have dedicated storage and memory
- Total # slices in a cluster is the unit of parallelism (max # of partitions / table)

Redshift Node Types and Slices

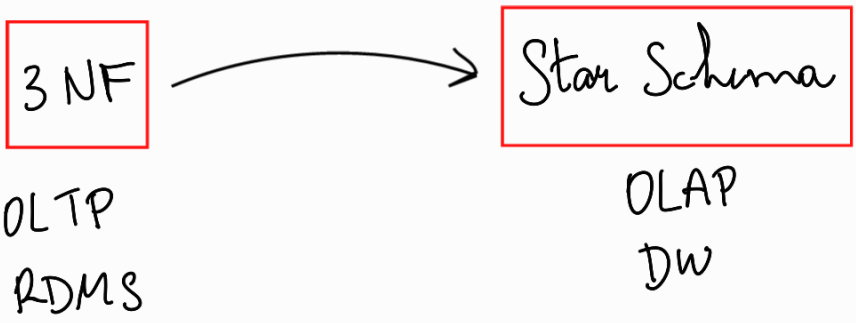
Compute Optimized Nodes:

- Start with these for lower costs and a capacity of 5 terabytes

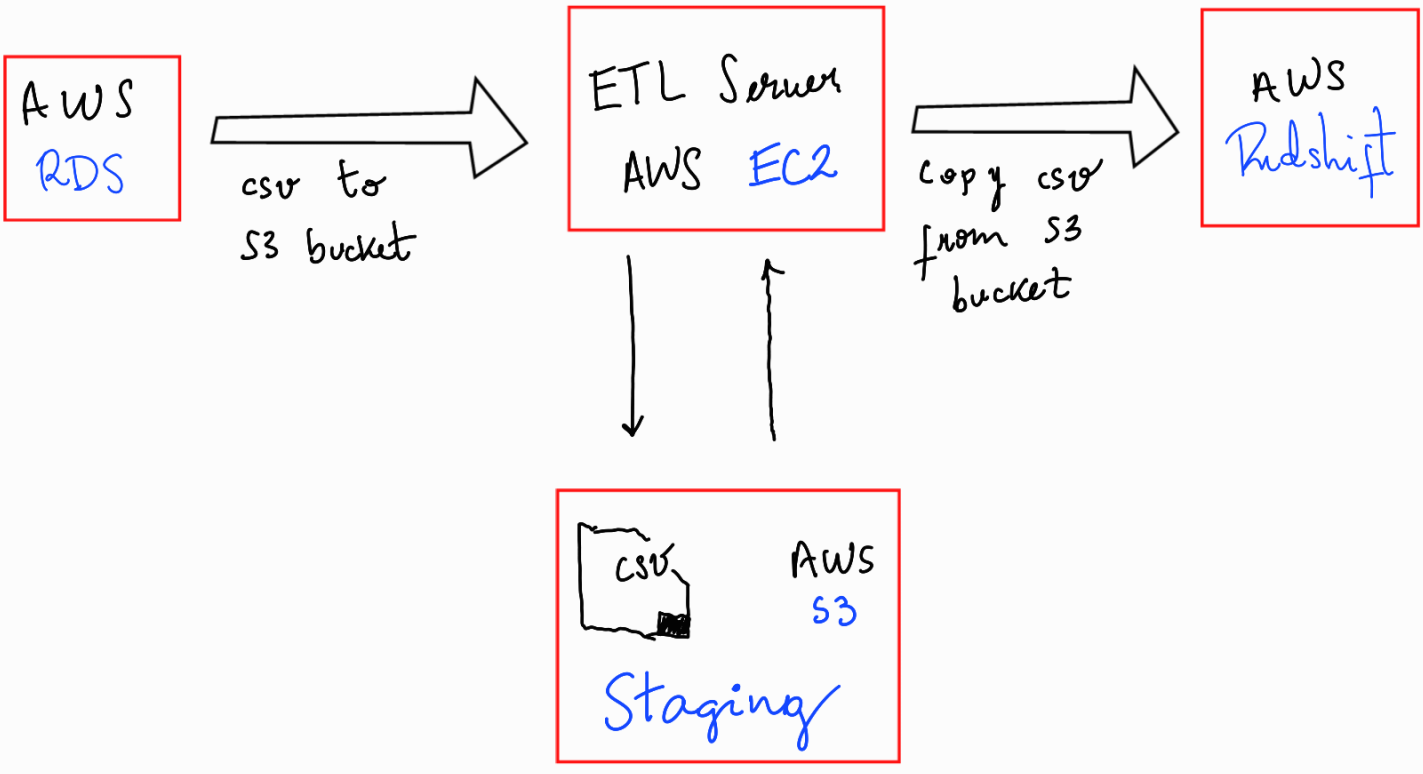
Storage Optimized Nodes:

- Higher costs, not as fast, but higher capacity

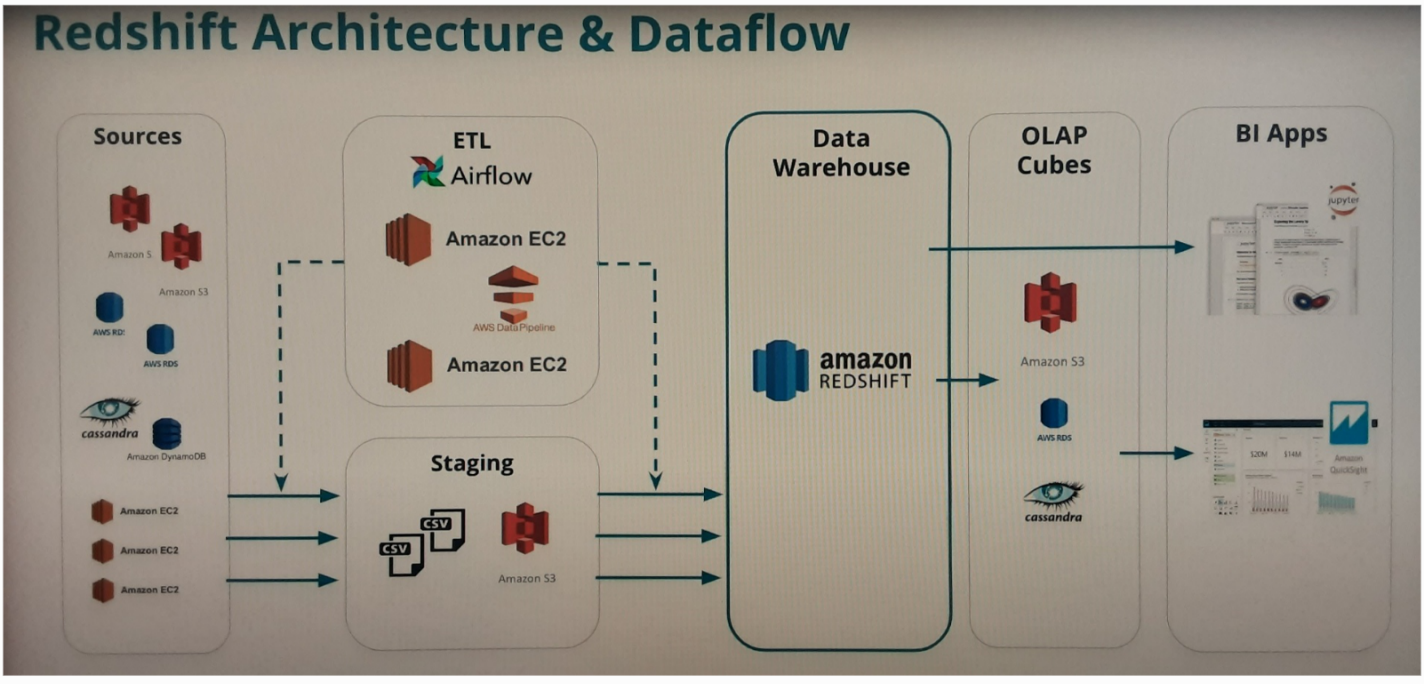
B Ingesting Data at Scale



1. Redshift SQL to SQL ETL



- Most of AWS products are able to communicate with S3 storage.
- Staging Area: tabela temporária; materialized view no banco ("área temporária") recepção de dados brutos p/ serem transformados!



- Ingesting at Scale
  - S3 staging → Redshift with COPY command → INSERT will be very slow
  - Ingest files in parallel → if file large
    - common prefix
    - manifest file (?)
  - Big files? Use compression
  - Use same AWS region to avoid network traffic

QUIZ

- Purpose of EC2 instance in ETL

  Acts as a client to RDS & Redshift to issue COPY commands

- Benefits of using S3 for ETL storage
  - Reliable, scalable and storage free of worry
  - Don't provide processing

- Small data
  - Possible to use EC2 for small data

-

Distribution Styles

- Optimizing the way a table is partitioned up into many pices and distributed accross slices on different machines

→ Column as DISTKEY sets distribution style to "KEY".

→ One can use a table attribute to set the dist. style.

```
CREATE TABLE <table> ( ... )
DISTSTYLE ALL;
```

☒ EVEN Distribution

- Round-robin method, regardless of values.
- When a table is not used in queries with joins
- Each slice with almost same # records

☒ KEY "

- Rows with similar values are placed in the same slice
- Used for tables frequently joined together.

☒ ALL "

- For small tables (e.g. lookups tables)
- Record a copy of the table/column in all slices.

☒ AUTO "

- Redshift is responsible for distributing

Redshift Cluster

- [redacted]
- [redacted]

[redacted]

Final Project Reviews

- About primary keys in staging tables

Staging tables are properly defined.

Reviewer Note

In data warehousing, staging tables serve as temporary storage for raw data being imported from source systems, before it's transformed and loaded into the final warehouse tables. They are a crucial part of the Extract, Transform, Load (ETL) process, as they allow for efficient data extraction and loading, while minimizing the impact on operational systems.

When defining staging tables, we typically avoid creating IDENTITY() columns or any kind of synthetic primary key. This is for several reasons:

1. **Preservation of Raw Data:** The main purpose of a staging table is to be a faithful reflection of the source data. Adding an IDENTITY() column or any other computed columns could distort this raw view. We want to ensure that the data in the staging tables is an exact copy of the source data, allowing us to perform transformations, cleaning, and conforming operations in subsequent stages of the ETL process.
2. **Avoidance of Unnecessary Computation:** Assigning IDENTITY() to a column means SQL Server (or any other relational database management system) would automatically generate a unique number for each row. This computation is often unnecessary in a staging table, as we are more focused on moving the data quickly and efficiently from source to staging, rather than creating new data.
3. **Preparation for Transformation:** In many cases, primary keys or unique identifiers will be generated or transformed in the process of loading data from the staging table into the final tables of the data warehouse. These identifiers are often based on the business logic and data requirements of the final tables, rather than the raw data.

Therefore, keeping staging tables as simple and as close to the source data as possible allows for more flexibility and control in the ETL process. IDENTITY() columns and similar constructs are generally more relevant in the design of the final data warehouse tables, where they can help to enforce data integrity and facilitate querying.

CREATE statements in sql_queries.py specify all columns for both the songs and logs staging tables with the right data types and conditions.