DATA ENGINEER NANODEGREE — UDACITY

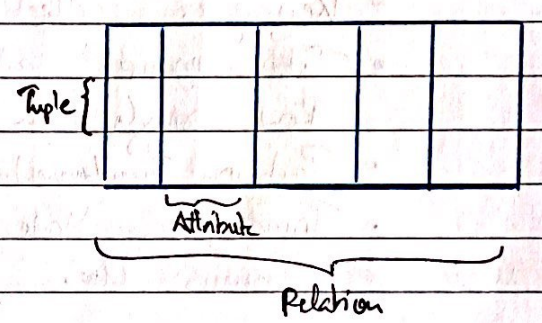
1] DATA MODELING

① INTRO

DB: Structure repo that is stored & retrieved electronically for use in apps
DBMS: DB Management System. Software used to access the DB

→ Relational DB: A DB based on the Relational Model
↳ Relational model

Types (common):
- Oracle
- Postgre SQL
- Teradata
- Sqlite
- MySQL



Basics:
- DB / Schema: collection of tables
- Table / Relation: group of rows w/ the same labeled elements
- Column / Attribute: labeled element
- Rows / Tuple: single item

ⓨ When to use a Relational DB?

- ✓ Smaller data volumes
- ✓ ACID Transactions
- ✓ Easier to change to bss requirements
- ✓ Flexibility for writing SQL queries
- ✓ Modeling data, not queries
- ✓ Ability to do JOINS
- ✓ Ability to do aggregations & analyt.
- ✓ Secondary indexes available

ⓝ When not to use a Relational DB?

- ✗ Large amounts of data
- ✗ Need to be able to store diff. data type
- ✗ Need ↑ throughput - fast reads
- ✗ Flexible schema
- ✗ High availability
- ✗ Need horizontal scalability

ACID Transactions
- Atomicity: a whole tx is processed or nothing is processed
- Consistency: only tx that abide by constraints are written in the DB
- Isolation: Tx are processed independently and securely. Order does not matter.
- Durability: Completed tx are saved to DB even in cases of syst failure

→ NoSQL DB: Not only SQL or Non Relational DB

Types (common):
- Apache Cassandra (Partition Row store)
- MongoDB (Document store)
- DynamoDB (Dynamo Key-value store)
- Apache HBase (Wide column store)
- Neo4J (Graph DB)

BASICS [APACHE CASSANDRA]: → Scalability ⊕ High availability
- Keyspace: collection of tables
- Table: group of partitions
- Row: single item
- Partition: fundamental unit of access (collection of rows
- Primary Key: Made up of a partition key ⊕ clustering column
- Columns: Clustering & data. Labeled element

Ⓥ When to use a NoSQL DB? → (⊖ when NOT TO use Relational DB)
⊕Ⓥ Users are distributed → low latency
Ⓝ When NOT to use a NoSQL DB? → Scale up linearity
→ (⊖ when to use a Rel. DB).

② Relational Data Models

DB: A set of related data and the way it's organised
DBMS: Computer software that allows users to interact w/ DB.

Relational Importance
- Standardization of data model: Once the data is transformed into rows & columns format → queryable)
- Flexibility in adding and altering tables:
- Data integrity
- Structured Query Language (SQL): Lang. to access data
- Simplicity: Data is systematically stored and modelled in tabular
- Intuitive Organization: as a spreadsheet.

OLAP: Online Analytical Processing
- DBs optimized for these workloads allow for complex analytical and ad hoc queries. OPTIMIZED for READS

OLTP: Online Transactional Processing
DBs optim. for these workloads allow for less complex queries in large volumes. The types of queries of these DB: READ, INSERT, UPDATE and DELETE → Very fast query processing

STRUCTURING your DB:

- Normalization: Reduce data redundancy and increase data integrity
- Denormalization: Must be done in read heavy workloads to ↑performance

OBJECTIVES OF NORMAL FORM
1. To free the DB from unwanted insertions, updates and deletions dependencies.
2. To reduce the need for refactoring the DB as new types of data are introduced.
3. To make the relational model more informative to users
4. To make the DB neutral to the query statistics
   ↳ opposite of NoSQL

NORMALIZATION

→ Process of normalization
1NF → 2NF → 3NF

[NORMALIZATION]

1NF:
- Atomic values: Each cell contains unique and single values
- Be able to add data w/o altering tables
- Separate different relations into different tables
- Keep relationships btw tables together w/ foreign keys

2NF
- Have reached 1NF
- All columns in the table must rely on the Primary Key

Ex.

[Store_Custom_ID]

Store ID	Customer ID
001	004
001	005
003	006

1NF → 2NF

Store ID	Customer ID	Name	...
001	004	...	...
001	005	...	...
003	006	...	...

[Customer table]

Customer ID	Name	...
004	...	...
005	...	...
006	...	...

3NF:
- Have reached 2NF
- No transitive dependencies

↓

When we want to update data, we want to be able to do it in 1 place.

Ex:

Music Award	Year	Winner Rec.	Lead Singer
Grammy	1965	The Beatles	John Lennon
CMA	2000	Faith Hill	Faith Hill
Grammy	1970	The Beatles	John Lennon

[Awards table]

Music Aw.	Year	Winner Rec
Grammy	1965	The B.
CMA	2000	Faith Hill
Grammy	1970	The B.

[Lead Singer]

Band Name	Lead Singer
The Beatles	John Lennon
Faith Hill	Faith Hill

DENORMALIZATION
- Process to improve the read performance of a DB at the expense of losing some write performance by adding redundant copies of data.

JOINS allow for outstanding flexibility BUT extremely slow.

Logical design change:
- The designer is in charge of keeping data consistency
- Reads will be faster (select)
- Writes will be slower (insert, update, delete)

FACT TABLES: Measurements, metrics or facts of a business process. (Transactions)

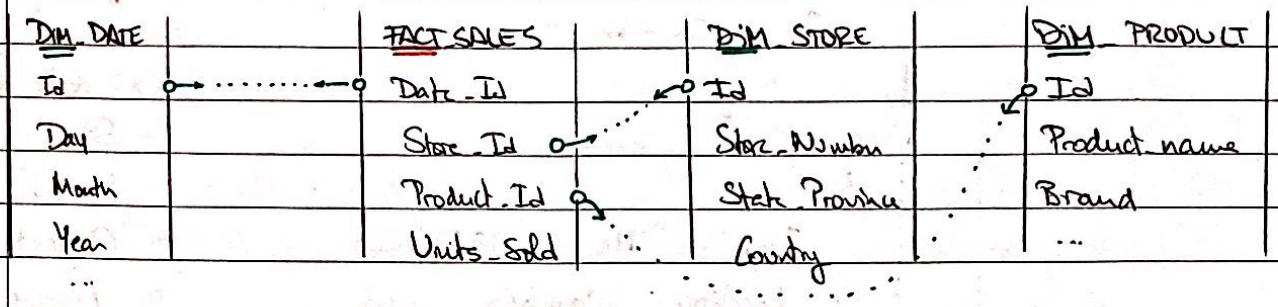
DIMENSION TABLES: Categorizes facts & measures. Enable users to answer biss questions. People, products, place and time

[DATA MART SCHEMAS]

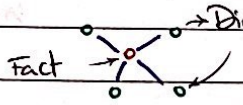
STAR : simplest style of data mart schema. 1+ fact tables referring any # dim tables

SNOWFLAKE: Arrangement of tables in a multidim. dB. Centralized facts conn. to mult dimensions.

Ex: STAR ★

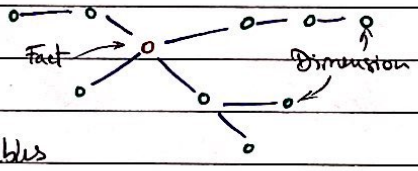


Benefits of ★ schema
- Denormalized
- Simplified queries
- Fast aggregations



Drawbacks:
- Issues that com w/ denormalization
- ↓ query flex
- Data integrity
- Many to many relationships

Ex: SNOWFLAKE ✳



- 1 fact table with multiple dimension tables with more child dimension tables
- Star Schema is a simplified case of a snowflake sch.

NOTE: While using SQL to create STAR/SNOWFLAKE schemas, there are some data definitions and constraints to take into account → CREATE statement

- **NOT NULL**: The column cannot contain a NULL value. This occurs w/ a composite keys (normally)
- **UNIQUE**: The data across all rows in 1 column is unique.
- **PRIMARY KEY**: Defined on a single column. The values in this column identify the table. If a group of columns are defined as a PK, they are called COMPOSITE KEY.
  - ↳ NOT NULL + UNIQUE constraints

→ UPSERT : inserting a new row or updating an existing one

table: customer_address
- customer_id
- customer_street
- customer_city
- customer_state

INSERT:
```
INSERT INTO customer_address (cust_id, cust_street, cust_city, cust_state)
VALUES (432, '123 Ab Street', 'Albany', 'NY')
```
UPDATE:
```
ON CONFLICT (cust_id)
DO UPDATE
    SET cust_street = EXCLUDED.customer_street
```

USEFUL Jupyter Notebooks / Python scripts:

JSON: Great to represent data w/ complex data hierarchy

```
{                                          ← object
                                  ← string
  "Directed by": "Steven Spielberg",
  "Produced by": [                ← array
      "Kathleen Kennedy",
      "Steven Spielberg"
  ],
  "Release": [                    ← nesting
      {
        "Date": "May 26, 1982"
        Location: "Cannes"
      },
      {
        "Date": "June 11, 1982"
        "Location": "USA"
      }
  ]
}
```

Values for both JSON objects and arrays can be any valid JSON data types

- String
- Number
- Object
- Array
- Boolean
- null

③ DATA MODELING WITH POSTGRES

INTRO : Sparkify : music streaming app. Analyze data : SONGS & USER act.
What songs users are listening | JSON logs
Creat Postgres DB to optimize queries | JSON metadata
① DB Schema ③ Test DB & ETL
② ETL pipeline

Description : • Define fact and dimension tables for a star schema using Python
• Write an ETL pipeline that transfers data : files → DB

Song Dataset : Real data from Milion Song Dataset
Each JSON file : metadata about a song and an artist
Files partitioned by the 3 first letters of each song's track ID
Ex: song_data/A/B/C/TRABCET128F424C983.json

→ Ex: 1 "num_songs" : 1
"artist_id" : "ARJIE...B7"
"artist_latitude": null
"artist_longitude": null
"artist_location" : ""
"artist_name" : "Line Renaud"
"song_id" : "SOUPI...E1"
"title" : "Der Kleine Dampfaff"
"duration" : 152.92036
"year" : 0 }

Log Dataset: Log files (JSON) by an event simulation based on the songs (↑).
Log files partitioned by year & month
Ex: log_data/2018/11/2018-11-12-events.json

→ To look at the JSON data in log_data files → pandas dataframe

df = pd.read_json(filepath, lines=True)
↑
'data/log_data/2018/11/2018-11-01-events.json'

STAR SCHEMA

FACT TABLE : ① songplays : records in log data. Records page NextSong
- col: songplay_id, start_time, user_id, level, song_id, artist_id, session_id, location, user_agent

DIMENSION TABLES :
② users :
- col: user_id, first_name, last_name, gender, level

③ songs :
- col: song_id, title, artist_id, year, duration

④ artists :
- col: artist_id, name, location, latitude, longitude

⑤ time : timestamps of records in songplays broken down into specific units
- col: start_time, hour, day, week, month, year, weekday

STEPS

① Create tables : - CREATE in 'sql_queries.py'
- DROP in 'sql_queries.py'
- 'run' 'create_tables.py'
- 'run' 'test.ipynb' confirm creation
- Restart kernel to close connection to DB after running 'test.ipynb'

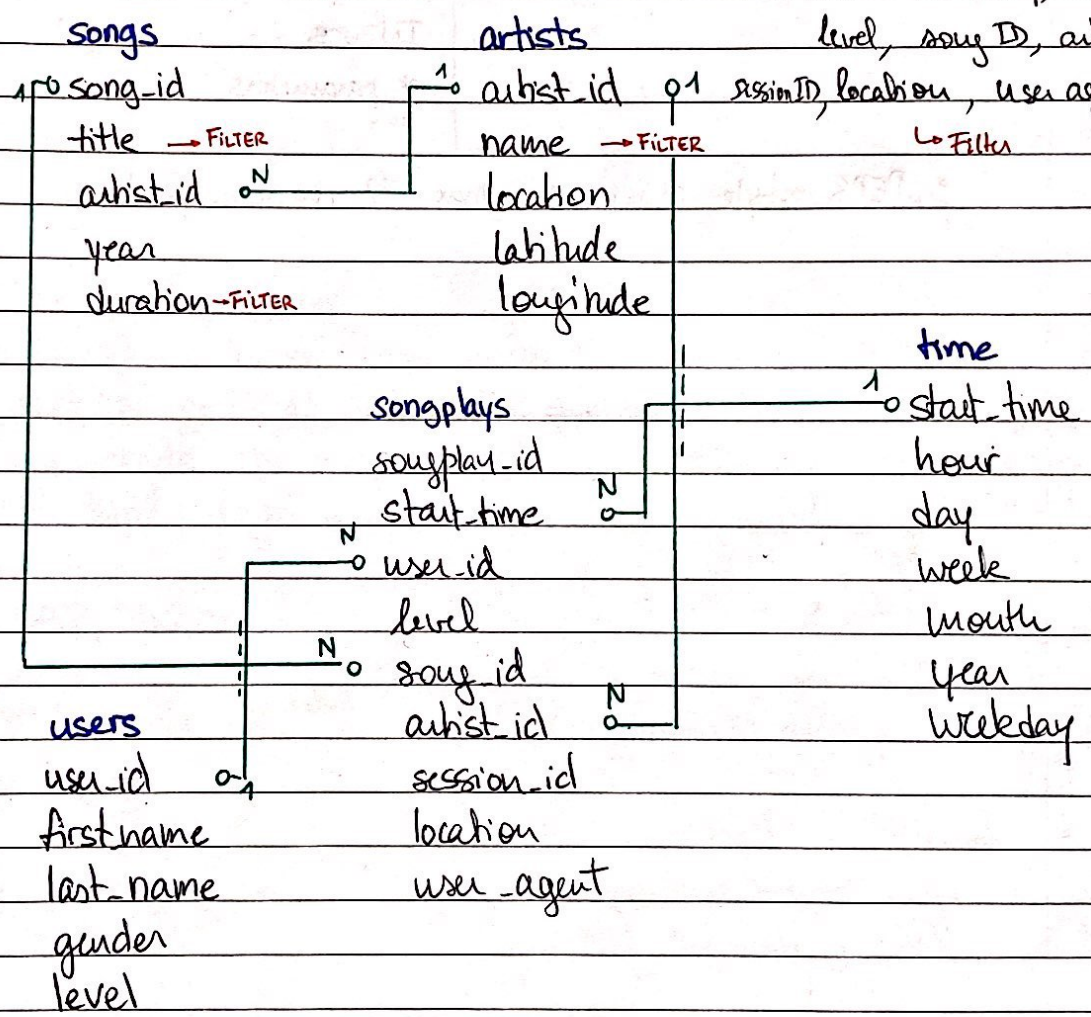
TIPS : ◦ CREATE statement : CREATE TABLE IF NOT EXISTS table_name (column data type, ...);
◦ DROP statement: DROP TABLE IF EXISTS table_name;
◦ File > New > Terminal # python create_tables
◦ Restart kernel w/ [C] button

② Build ETL processes - Instructions in 'etl.ipynb'
↳ At the end of each table section : 'run' 'test.ipynb'
↳ TIP! Rerun 'create_tables.py' to reset tables

TIPS:
- 'sql_queries.py' : INSERT INTO table_name (column, ...) VALUES (%s, %s, ...) : (+) ON CONFLICT ...
- song_data = df[['song_id', 'title', 'artist_id', ...]]
song_data_list = song_data.values[0].tolist()
cur.execute(song_table_insert, song_data_list)

- FILTERING w/ Pandas: `df[df.column == 'Value']`
- COPY df: `t = df.copy()`
- DATETIME: `t['ts'] = pd.to_datetime(df['ts'], unit='ms')`
- Extract time: `t['ts'].dt.hour` OR `t['ts'].dt.week`
- `frame = {'column_1': t['ts'].(...), 'column_2': t['ts'].(...), ...}`
  `time_df = pd.DataFrame(frame)`

songplays TABLE: song_select query: ① find song_id, artist_id based on the title, artist name & duration.
② Select timestamp, user ID, level, song ID, artist ID, sessionID, location, user agent



Project Review

3 specifications:

- Add NOT NULL constraints:
  Ex: songplay table : start_time NOT NULL
- ⊕ Add FOREIGN keys
  → Added songplay.start_time NOT NULL REFERENCES time(start_time)
  songs.artist_id NOT NULL REFERENCES artists(artist_id)
- Add doc strings → def function(parameters...):

```
"""
Description
INPUTS
* parameters
"""
```

- PEP8 style guidelines: max 72 characters per line

④ NoSQL Data Models

NoSQL and Non-Relational are interchangeable terms
↳ Not Only SQL

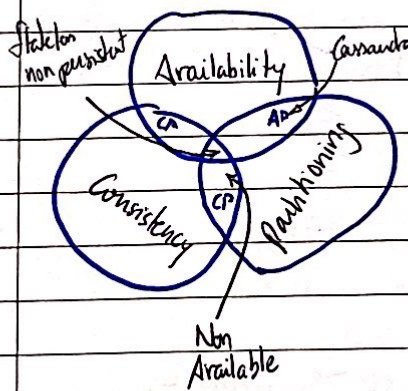
Apache Cassandra
↓
Distributed DBs
- In order to have high availability, you will need copies of your data
  ① A DB made up of multiple machines
  ② High avail.: (almost) no downtime → with ⊗ nodes
  ③ Because 1 node can go down ⇒ need copy

- Open Source NoSQL DB
- Masterless Arch
- High Availability
- Linearly Scalable (adding nodes)

User	Netflix
Hulu	Twitter

▷ **Eventual Consistency**
Consistency model used in distrib. computing to achieve HIGH AVAILABILITY that informally guarantees that, if no updates are made to a given data item, eventually all accesses to that item will return the last updated value.

CAP Theorem



- Consistency: Every read from the DB gets the latest (and correct) piece of data or an error.
- Availability: Every request is received and a response is given - w/o the guarantee that the data is latest update
- Partition tolerance: The system continues to work regardless of losing network connectivity between nodes

Apache Cassandra is...
- highly available DB
- Linearly scalable
- An open source project (Apache Foundation)

DENORMALIZATION

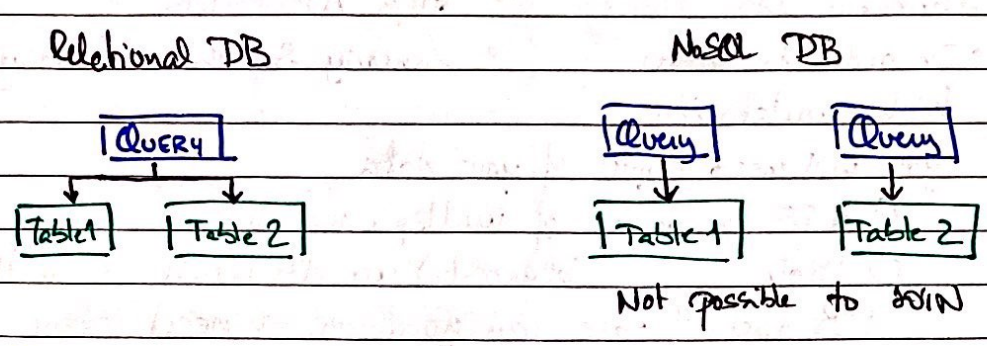
↳ It's a must w/ Apache Cassandra

- Done for fast reads
- A. Cassandra has been optimized for fast writes

ALWAYS think queries first

- Good strategy: 1 table per query

→ A. Cassandra does NOT allow for JOINs btw tables.



CQL Cassandra Query Language

Language to interact w/ DB. Very similar to SQL

No: JOINS, GROUP BY or subqueries not in CQL, not supported by CQL

Python wraper/driver : pip install cassandra-driver

import

```
import cassandra
```

connection to DB

```
from cassandra.cluster import Cluster
try:
    cluster = Cluster(['127.0.0.1'])   # locally installed Ap. Cass. instance
    session = cluster.connect()
except Exception as e:
    print(e)
```

create KEYSPACE

```
try:
    session.execute("""
    CREATE KEYSPACE IF NOT EXISTS udacity
    WITH REPLICATION =
    {'class': 'SimpleStrategy', 'replication_factor': 1}""")
except Exception as e:
    print e
```

connect

```
(try/except) session.set_keyspace('udacity')
```

query

```
query = "CREATE TABLE IF NOT EXISTS music_library"
query += "(year int, artist_name text, album_name text,
    PRIMARY KEY (year, artist_name)
```

PRIMARY KEY : ① PARTITION KEY

② PARTITION KEY + CLUSTERING COLUMNS

↓

PARTITION KEY : Determine the distribution of data accross the system

↳ The partition key's row value will be hashed (→ #) and stored on the node in the system that holds that range of values.

PK

- Must be UNIQUE
- Hashing on this value results in placement on a particular node in the syst.
- Data distrib. by this partition key
- simple or composite
- 1+ clustering columns (optionals)

- SIMPLE: ①

```
CREATE TABLE music_library
(year int, artist_name text, album_name text,
    PRIMARY KEY (year) )
                 ↑ partition key
```

- COMPOSITE: ②

```
CREATE TABLE music_library
(year int, artist_name text, album_name text,
    PRIMARY KEY (year, artist_name ))
                 ↑ partition key    clustering column
```

CLUSTERING COLUMNS:

- Will SORT the data in ASC.
- 1+ (or none) can be added
- The columns will be sorted in order of how they were added to the PK.

```
CREATE TABLE music_library
(year int, artist_name text, album_name text,
    PRIMARY KEY [((year), artist_name, album_name)] )
                  ↑ partition key   1st clust. column (ASC)   2nd clust. column
```

PRIMARY KEY

↳ for each artist_name it orders the album names ASC.

year	artist_name	album_name
1965	Elvis	Blue Hawaii
1965	The Beatles	Rubber Soul
1965	The Beatles	Test
1965	The Monkees	Meet the Monkeys

ASC

'WHERE' CLAUSE:

- Must be included to execute queries.
- Recommended that 1 partition be query at a time (performance)
- Possible: 'select * from table' if you add a config: ALLOW FILTERING
  ↑ RISKY

→ PARTITION KEY must be included

→ CLUSTERING COLUMNS can be used in order they appear
  ↳ We cannot use the 3rd clust. col. if we do not add the 1st & 2nd.

⑤ DATA MODELING WITH APACHE CASSANDRA

INTRO: ⊖ POSTGRES

Create A Cassandra DB → song play data

EVENT DATA

Directory of CSV files partitioned by date

Ex: 'event_data/2018-11-08-events.csv'

CSV: Comma Separated Value

Jupiter notebook: ① Process: 'event_datafile_new.csv'
② Model data tables (w/ queries in mind)
③ Queries needed to keep in mind
④ Load data into tables + Run queries

STEPS

① Modeling NoSQL /Apache Cassandra DB

event_datafile_new.csv

1 artist	ABC	5 lastName	ABC	9 sessionId	123
2 firstName	ABC	6 length	1.234	10 song	ABC
3 gender	ABC (1)	7 level	ABC	11 userId	123
4 itemInSession	123	8 location	ABC		

↳ in array [-1]

QUERY 1: SELECT 1, 10, 6 FROM music_library WHERE 9 AND 4

QUERY 2: SELECT 1, 10, 2, 5 FROM music_library WHERE 11 AND 9
↑ SORTED BY itemInSession [4]

QUERY 3: SELECT 2, 5 FROM music_library WHERE 10

Columns:		
	1	normal
	2	normal
	4	PRIM. KEY
	5	normal
	6	normal
	9	PRIM. KEY
	10	
	11	

Q1) 1
4
6 CC1
9 PK
10 CC

Q2) 1
2
4
5
9
10
11