it's not luck
it's hard
work

DATA ENGINEERING

Learn to design DATA MODELS, build DATA WAREHOUSES and DATA LAKES, automate DATA PIPELINES and work with massive datasets.

→ vital role in storing, organizing + making use of data.
↑ speed + volume of INFO Generated every day

HOW TO BE **SUCCESSFUL** IN THE PROGRAM

- BE an ACTIVE STUDENT : Take notes
- write the CODE YOURSELF
- Time management : GO to class as a rutine

? how to address BLOCKERS
→ Research → Read → Ask 4 help

The Data science Hierarchy of Needs

Role	Stage	Examples	Area
Research-DS, ML Engineer		AI, Deep learning	ML + AI
Data scientist	Learn + Optimize	A/B testing, experimentation, machine learning algorithms	ML + AI
Analyst	Aggregate + Label	Analytics, metrics, segments, Aggregates, Features, training data	Reporting + Analytics
Data Engineer	Explore + Transform	cleaning, anomaly detection, prep	Data storage + Processing
	move + store	Reliable data flow, infraestructure, pipelines, ETL, structured unstructured data	Data storage + Processing
Software Dev.	collect	Instrumentation, logging, sensors, external data, user gen content	system Architecture

Roles + Skills

Soft Dev		DE	DS	
HTML	web Services	Hadoop	SQL	R
CSS	SW Dev.	Data wareh.	Big data	Statistics
XML	Agile Meth.	ETL	Data analysis	ML
JQuery	MySQL	Hive	BI	SAS
PHP	Linux	Unix	Algorithms	
.NET	Java Python	oracle	Data Mining	
	JS	Databases		
	C	SQL		

Common Data engineering Activities :

- Ingest data from a data source
- Build + Maintain a data warehouse
- Create a data pipeline
- Create an analytics table for a use case
- Migrate data to cloud
- Schedule + automate pipelines
- Backfill data
- Debug data quality issues
- Optimize queries
- Design a Database.

DATA MODELING

→ Abstaction that organizes elements of data and how they will relate to each other.

→ The process of creating data models for an information system.
Organize data into db systems to ensure that data is persisted and easily usable.

WHAT?

- Process to support business and user needs
- Gather requirements from apps team to users

① • Conceptual data modeling w/ entity mapping
→ map concepts of data that you have/will have and relate data concepts

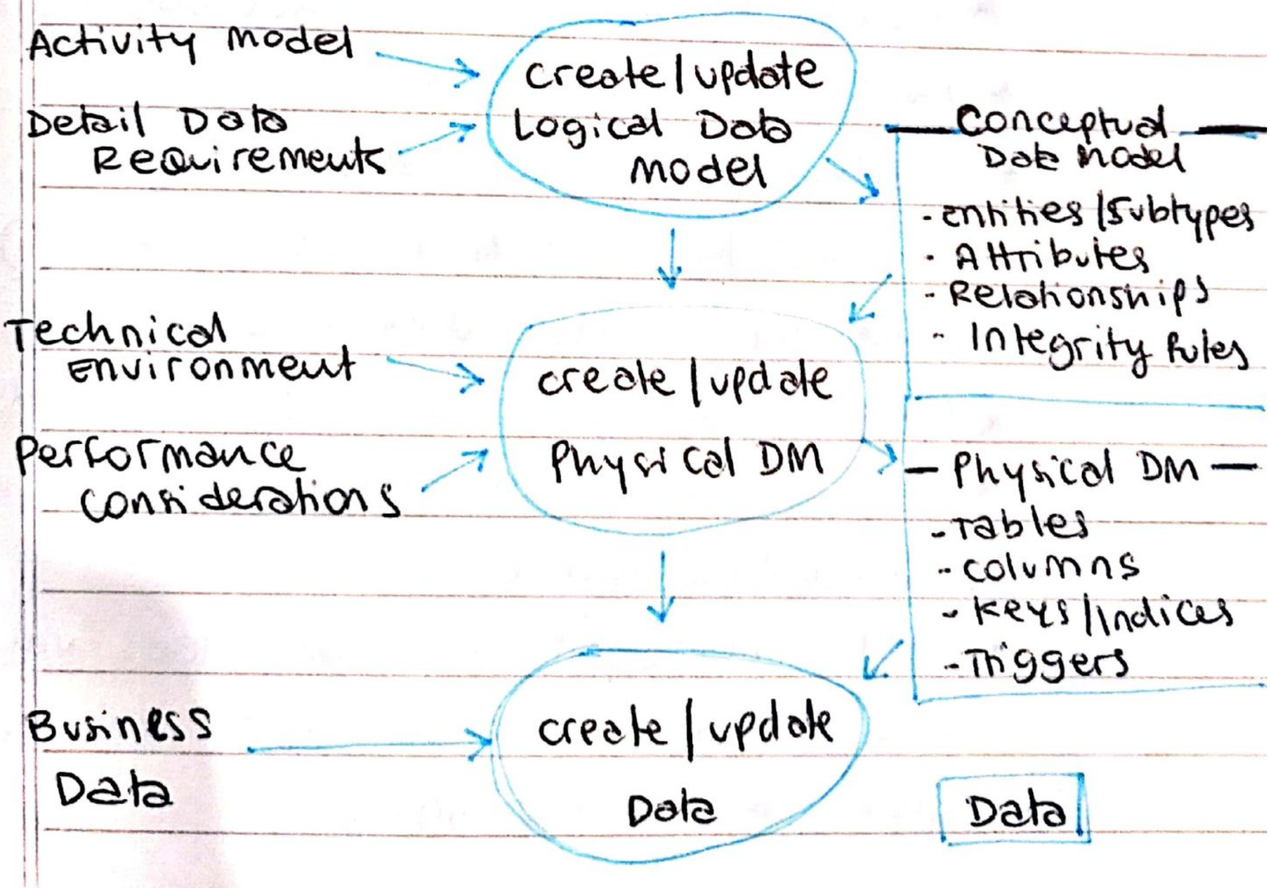
② • Logical data modeling
→ Map concepts to logical models using schemas, tables, columns...

③ • Physical data modeling
→ DDL: Data definition language
code to persist our data

* Data organization is critical - makes everyone's life easier.
* Organized data determines later data use
* Prior step to building out app, business logic, and analytical models. - start early.
* Iterative process - as new reqs + data come in.

WHO?

→ Everyone who works with data.



Relational Model

This model organizes data into one or more tables of columns + rows, with a unique key identifying each row. Each table represents an entity type. (Generally)

Columns = Attributes
Rows = single items

WHY USE RDBMS?

Ease of use:

* Flexibility for writing SQL queries
* Ability to use JOINS
* Ability to do agg and analytics (Group by, Sum, Min, MAX, AVG...)
* Great for smaller data volumes
* Easy to change business requirements
* You model the data, independent of queries.
* Secondary Indexes
* ACID Transactions = Guarantees validity even in the event of errors, power failures, → ⇒ Data integrity

Atomicity: The whole transaction is processed or nothing is processed.
→ All or nothing.

Consistency: Only transactions that abide by constraints and rules is written into the db, otherwise DB keeps previous state. → Data should be correct accross all rows and tables

Isolation: Transactions are processed independently and securely, order does not matter. → Reduces changes of concurrency effects.

Durability: Completed transactions are saved to DB even of cases of system failure → Transactions are recorded on non-volatile memory

Scanned by CamScanner

WHY NOT USE RDBMS ?

* Large amount of data: can only scale vertically ⇒ They are not distributed. Limited by the host server.
* Need to store diff data types formats.
* Need high throughput = fast reads. ACID transactions slow down r/w process.
* Need flexible schema ∴ columns added to tables, but not used on every row.
* Need high availability = As they are not distributed → SPOF. If DB goes down, failover backup sys. must happen ⇒ Takes time.
* Need horizontal scalability

HA : very little downtime of the system, it is always on and functioning.

H.S : ability to add servers and nodes to improve performance

POSTGRE SQL


```
import psycopg2

try:
    conn = psycopg2.connect("host=127.0.0.1
        dbname=studentdb   user=student   password=")
except psycopg2.Error as e:
      print (" Error   caught")
       print ("e")


cur = conn.cursor()
conn. set_session (autocommit =True)

cur. execute(" create   database   <name>")
conn. close ()

cur. execute ('CREATE  TABLE  IF  NOT  EXISTS
     <table_name>  (<col_name>   <col_type>,
                              ... )');

        ('INSERT INTO <table_name> (<col_name>, ..)
           VALUES (%s,  ...) ,
                      (<value_1>,  .. ));
```

NO SQL

$\underbrace{\text{NO}}_{\text{Not only}}$ = Non Relational DB

Simple design, horizontal scaling, finer control of availability.
Data structures <> RDB
Faster operations.
- Different types of NoSQL & implementations

Common types:

DB	Type
Apache cassandra	Partition row store
Mongo DB	Document store
Dyna mo DB	Key-Value store
Apache HBase	Wide Column store
Neo 4j	Graph DB

CASSANDRA

Data distributed by partitions across nodes or servers and the data is organized in the columns and rows format. → Own language = CQL

Keyspace : Collection of tables [≃DB]
Table : A group of partitions
Rows : A single item
Partition : Fundamental unit of access.
Collection of row(s). - How data is distrib.
Primary Key: PK is made of a partition key + clustering columns.
Columns : Clustering and Data
Labeled elements.

→ Provides scalability and high availability without compromising performance.
Linear scalability + Fault-tolerance on comodity Hw / cloud infra
⇒ Perfect platform for mission-critical data.

Good for
- Transactional logging (retail, health care)
- IOT
- Time series data
- Heavy on writes workloads.

Bad for
- Analytics
- Group by's
- Ad-hoc queries
- Joins

WHEN USE NOSQL
* Large amount of data
* Need horizontal scale → + Nodes + Performance
* High troughput - reads.
* Need flexible schema
* Need high availability
* Different data type formats
* Distributed users - low latency

WHEN NOT USE NOSQL

* ACID Transactions
* Need to use JOINS
* Aggregations + Analytics
* Changing business requirements
* Small data set
* Queries not available, flexible.

→ This properties are by design. Some products can be customized to add this support.

```
import cassandra

From cassanda.cluster import Cluster
cluster = cluster(['127.0.0.1'])
session = cluster.connect()
session.execute("""
  CREATE KEYSPACE IF NOT EXISTS <name>
  WITH REPLICATION =
  { 'class':'simpleStrategy',
  'replication.factor': 1 }
""")

session.set_keyspace('name')
```

Scanned by CamScanner

RELATIONAL DATA MODEL

IMPORTANCE

- Standarization of data model
- Flexibility in adding/altering tables
- Data Integrity
- SQL
- Simplicity
- Intuitive Organization

OLAP - Online Analitical Processing

→ DB optimized for analytical + ad hoc queries.

OLTP - Online Transactional Processing

→ DB optimized for less complex queries in large volumes

Normal Forms

copies of data

Normalization → to reduce data redundancy and increase data integrity.
correctness of data

Dernormalization → must be done in read heavy workloads to increase perform.

OBJECTIVES

1. Free DB of unwanted insertions, updates, deletion dependencies
2. Reduce need of refactoring the DB as new types of data are introduced.
3. Make data model informative to users.
4. Make dB neutral to query statistics.

normal forms

1NF

- Atomic values : each cell must contain unique single values (No sets, arrays, etc)
- Be able to add data w/o altering tables.
- Sepparate different relations into diff tables.
- Keep relation between tables together with FK

2NF

- ⊕ 1NF
- All columns in the table must rely on the PK

3NF

- ⊕ 2NF
- No transitive dependencies

⇒ to get from A → C you shouldn't need to go through B

: when you want to update data, we want to be able to do in just 1 place.

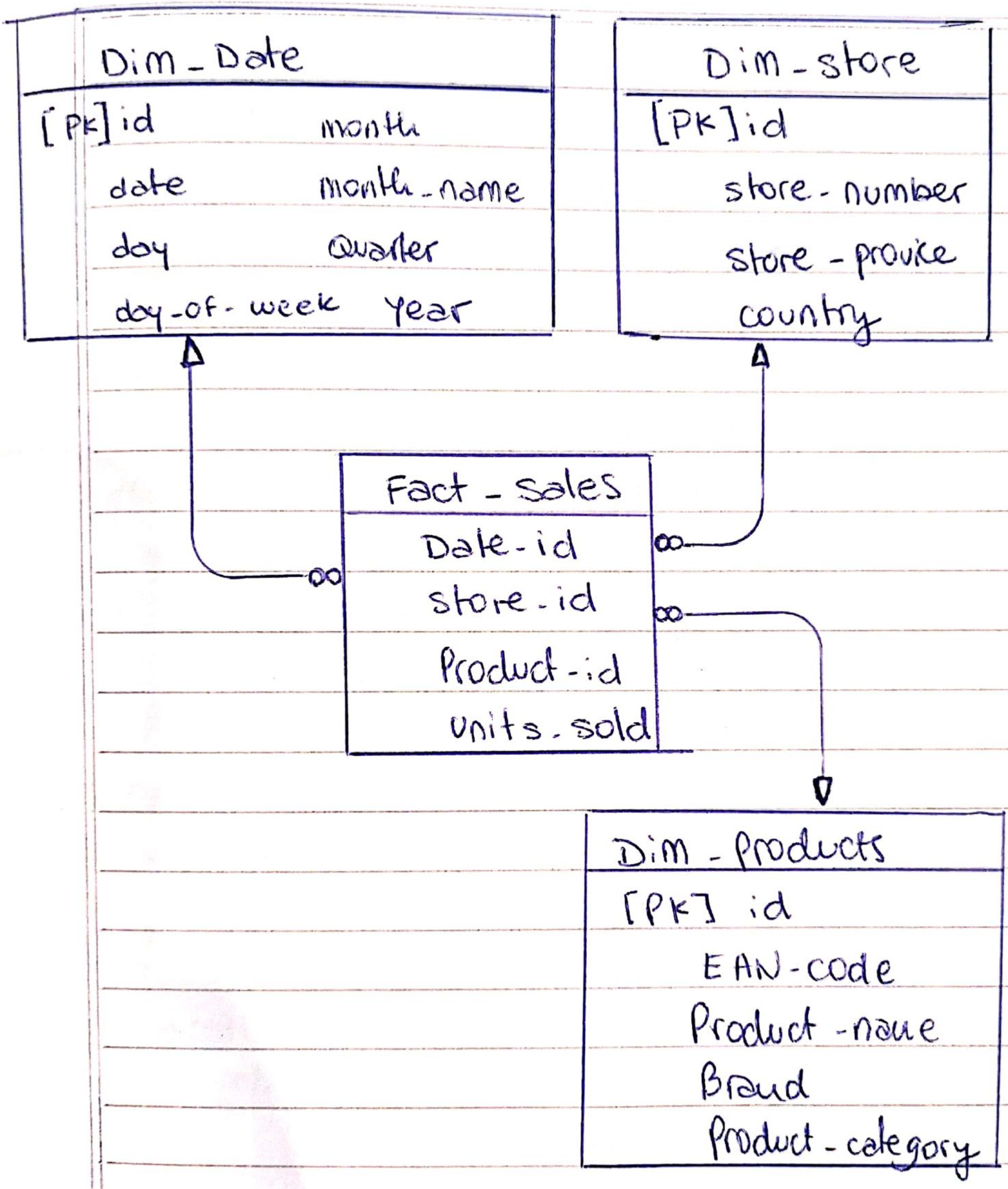
Fact & Dimensional Tables

Fact tables → consists of measurements, metrics, facts of a business process. Could contain aggregations of data, but are not meant to be modified in place.

Dimension tables → structures that categorizes facts in order to enable users to answer business questions. Dimensions are pleople, products, places, time...
They give context to facts.

PKs of dim table are included in fact tables.

2 main schemas.
→ star schema
→ snowflake schema


Dim_Date
[PK] id
date
day
day-of-week
month
month_name
Quarter
Year
Dim_store
[PK] id
store-number
store-provice
country
Fact_sales
Date-id
store-id
Product-id
units_sold
Dim_Products
[PK] id
EAN-code
Product-naue
Braud
Product-category

STAR SCHEMA

Simplest style of data mart schema.

→ Consists of 1 or more fact tables referencing any number of dimension tables.

BENEFITS	DRAWBACK
- Denormalized	→ Denormalization issues
- Simplified queries	- Data integrity
- Fast Aggregation	- Decrease query flexibility
	- Many↔many relationship

SNOW SCHEMA

Tables in a multidim DB represented by centralized fact tables which are connected to multiple dimensions.

- Allows for 1↔many relationships.
- more normalized than star.
  → Not 3NF though

NOSQL DATA MODEL

not-only sql

- Need H.A. in the data
- Have large amounts of data
- Need linear scalability
- Low latency
- Faster reads & writes

CASSANDRA

- Open Source NoSQL DB
- Masterless Architecture - NO SPOF
- High Availability
- Linearly Scalable → + Nodes ⇒ + perf.
- Used by Hulu, Facebook, Twitter, Netflix

DISTRIBUTED DB

→ To have H.A. we need more copies of your data.

Database runs over several nodes.

HA : no down time

Eventual consistency

Consistency model used in distrib. DB computing to achieve HA.

→ if no new updates are made to a given data item, eventually all access to the item will return the last updated value - Data may be inconsistent for only milliseconds

CAP Theorem

It is impossible for a distrib. db ~~to~~ store simullaneously provide more than two out of the following three guaranteed of consistency, availability and partition tolerance.

CONSISTENCY: Every read from the DB gets the latests (and correct) piece of data or an error.

AVAILABILITY: Every request is received and a response is given - w/o guarantee that the data is the latest update.

PARTITION TOLERANCE: System continues to work regardless of losing network connectivity between nodes

DENORMALIZATION in CASSANDRA

→ Absolutly critical. A MUST!

The biggest take away when data modeling in cassandra is to think your queries first.

- There is NO JOINS here.

- must be done for fast reads
- Cassandra has been optimized for fast writes.
- One Query per table → Great strategy.

(partition by search item)

```
import cassandra
cluster = Cluster({<ips>})
session = cluster.connect()

session.excecute ("""
    CREATE KEYSPACE IF NOT EXISTS
      udacity WITH REPLICATION =
        {'class' : 'SimpleStrategy',
         'replication_factor' : 1 } """)

session.set_keyspace ('udacity')
```

I want to run queries:

1. select * from music-library where year=1970
2. select * from artist-libray where artist = " "

FOR 1 Table : music_libary

columns : year, Artist name, album

PK : (year, artist name)

joint key

year is partition key

artist name is clustering key

FOR 2 : Table artist_library

PK : (artist_name, year)

PRIMARY KEY

→ how each row can be uniquely id. and how data is distributed across the nodes in our system.

• made up of only the
PARTITON KEY ← simple PK

or w/ the addition of
CLUSTERING ~~KEY~~ COLUMNS ← composite PK

Determines the distribution of the data.
⇒ The PK will be hashed and stored on the node that holds that range of values

• TO choose a PK:
→ we want to spread our data equaly.
→ Ensure that is unique for each row.

CLUSTERING COLUMNS

→ Determines the sort order within a partition. (DESC order)

More than 1 can be added, and they will sort in the order they were declared. → nested sorted order

... PRIMARY KEY ((year), artist-name, album-name)

→ If you want to use cc in select you must use them in order.

WHERE CLAUSE

DM in Cassandra is query focused

→ Partition key must be in your where

→ Clustering Cols. can appear in order.

Select * from table → can be done, but is highly discouraged.