Data Warehouse

(Udemy course)

Datawarehouse concepts

Why do we need Datawarehouse?

① Operational purposes
→ receive orders
→ react to complaints
→ fill up stocks (demand - supply)
basically to keep the company running
→ refer to OLTP (online transactional processing)

requirements
① one record at a time
② no longer history (only current data is maintained)
③ data input

② Analytical purpose.
→ # of sales this month vs last month
→ what is best category?
→ what can be improved?
to evaluate performance & make better decision
→ refer to OLAP (online analytical processing).

① Analyze 1000 to millions of records all at the same time.
② fast query performance
③ to analyze data over time. history data is maintained.

Datawarehouse (DWH) is there to address analytical data need.
. It is used for reporting and data need.

What is a Datawarehouse? A database used and optimized for analytical purpose.
✓ user friendly (same readability)
✓ fast query performance (to pull & process lot of data)
✓ enabling data analysis.

Operational data systems

Sales data
HR data
CRM data
Other data

→ Data warehouse

integrate all diff sources, so that they are in same format

ETL process.

extract data from operational systems

load data into centralized location for data

Goal of Datawarehouse:

① Data is available in centralized system in a consistent manner
② Data must be fast accessible (query performance)
③ User-friendly (easy to understand in term of modeling the data)
④ must load data consistently & repeatedly (ETL)
⑤ reporting & build data visualization on top of it.

Business Intelligence: Transform raw data to fetch meaningful insights & make better decisions.

Data lake	Data warehouse
① use centralized location for data storage as well.	① centralized location for data storage.
② store raw data	② store processed data (user friendly based on we need clean data)
③ Big data technology (CSV, JSON, image, video file - store unstructured data)	③ database (tables, schemas) clean tables (structured data)
④ usage is not defined yet	④ specific & ready to be used
⑤ use as Data scientists	⑤ Business user & IT
⑥ needs cleaning	⑥ highly user friendly

disadv: data quality is not ensured, do not know the purpose of data but by using cloud technologies which are used to process this data & are scalable the data quality issue can be reduced.

[To fetch insights → we datawarehouse → on some parts of datalake]

Hence, data lake and datawarehouse are not always exclusive to each other.

[we use ETL process to get data out of data lake & load into datawarehouse which is user friendly to fetch insights]

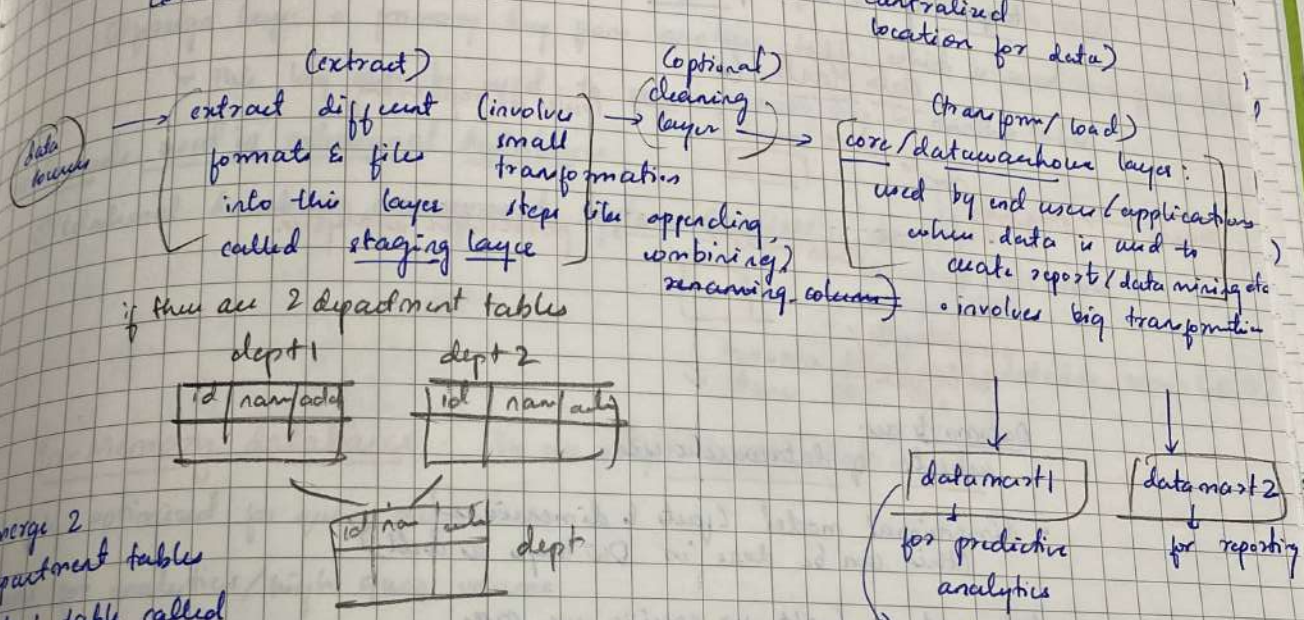
3 Layer

Data to use

eg: me dep in

Module — Data Warehouse Architecture

3 Layer → Data warehouse



Staging Area

Purpose: extract data from data sources and place in staging area.

Why do we need this layer?

① spend less time on source systems to avoid huge transactions

② quickly extract (quick read access → pull data → place it in tables)

③ move data into relational database in the form of tables

④ start transformation from here

- staging layer stores temporary data, i.e after every ETL the layer's data is truncated.
- to check what is new in data source, build delta logic in place which has delta column,

for ex: if delta column's max value was 5, then load all rows w/ values greater than 5.

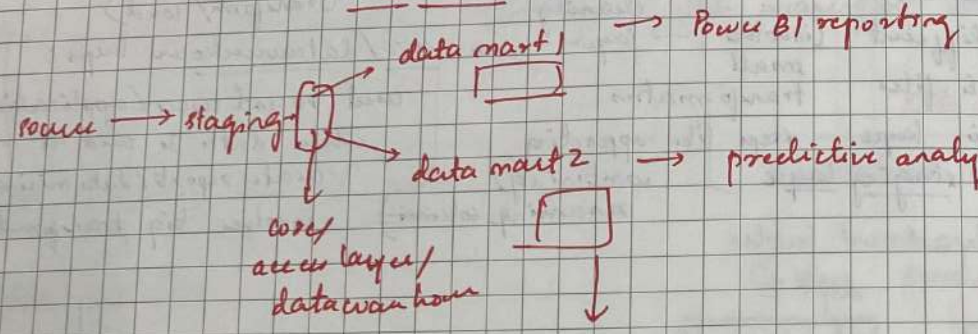
usually **date** is used as delta column.

- append data using transformation logic in DW.

Alternative: Persistent staging layer: used as a rollback option when we are never truncating data in staging layer.
(used to avoid going back to operational data source)

Type of staging layer
- temporary
- persistent

Data Marts (



Datamart are:
- subsets of datawarehouses
- dimensional model (facts & dimensions)
  (this can be done in DW layer as well)
- data mart are built for specific use case
- can be further aggregated based on requirement

Benefits of datamarts: ① usability + acceptance (data acceptance for use case)
↓
ability to work w/ only specific data

② used to improve performance : data is modeled in a dimensional way and we can use specific techs like in-memory DB / datacubes.

When to use datamarts / use cases: ① data marts can be created for diff tools
like dm for reporting w/ diff data
dm for predictive analytics

② dm's for diff department

③ diff dm's for diff regions.

④ diff dm's for diff use cases.

Relational Databases: A database where data is stored in the form of tables. (Tables are also called relations).

- data in a relation / table is stored in the form of rows/columns.
- SQL (Sequel query lang) used to query data in relational DBs.

Keys: ① Primary key: a column that uniquely identifies every row in a table.

② foreign keys: a primary key from another table, which is used in a given table.
↳ This key can be used to refer primary key in another table

* **Products used in relational databases**:

① Relational database management systems (RDBMS): Oracle,
Microsoft SQL server
Postgre SQL
My SQL
Amazon Relational database service (RDS)
Azure SQL DBs.

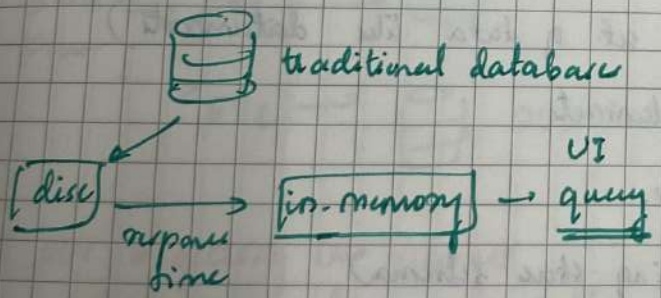
In-Memory databases: (to use only relevant data)

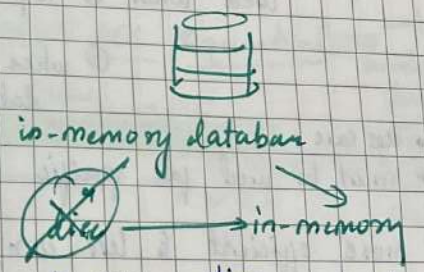
① highly optimized for query performance

② good for analytics / high query volume

③ usually used for data mart

④ relational & non-relational





the response time coming from disk is eliminated

ⓐ columnar storage (data is scanned through columns not by rows)

ⓑ parallel query plans: larger query can be broken down into multiple parts and then executed.

Eg: ① SAP HANA
② Oracle in-memory
③ MS SQL server in-memory tables
④ Amazon MemoryDB

disadvantage: ① durability: loses all information when device loses power or is reset.

resolution: ① snapshot / images of current state of data can be maintained if we need to roll back.
(costly)

② cost-factor: the amount of data is ↑ than to store them.

③ traditional DBs are also reducing usage of disc.

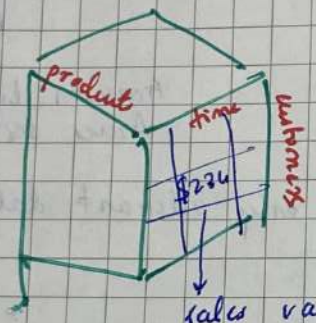
Cubes

OLAP cubes :

- In traditional DW, data is organized based on relational RDBMS (ROLAP)
- Data is organized non-relational in cube (MOLAP)

  cube = multi-dimensional dataset.
- Arrays are used instead of table
- Used for fast-query performance.
- works well with many BI solutions.

sales data into multiple dimension



- pre-calculated aggregated values.

- MDX language :
  Multi-dimensional expression for Microsoft can be used to query cubes.

Used when ① high performance is needed

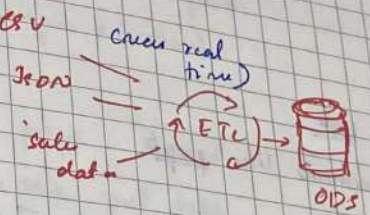
② when hierarchy plays a role, & when one needs to slice & dice the data or drill the data.

use case :

* must be used for specific use case (specific set of data like datamarts)
* more efficient & less complex w/ separate datamart
* good for interactive queries with hierarchies.
* can use data from relational DB (built using star schema)

Operational Data storage (ODS)

- similar to datawarehouse
- data is fetched from multiple data sources & using ETL, data is stored in operational data storage (ODS).
- ODS is used for operational decision making

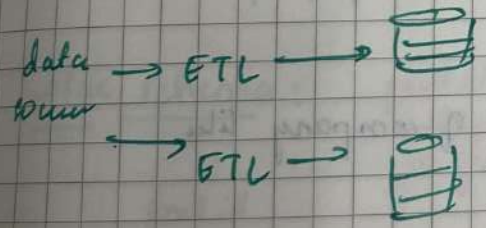


Use cases (used when: ① you dont need long history of data
② can be very current or real-time.

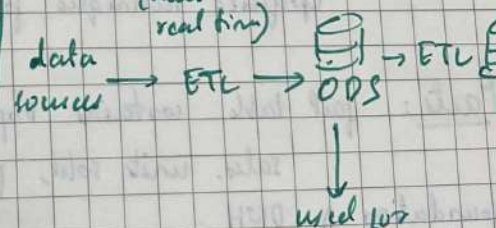
Eg: In a finance company,
customers can invest in ETF, stocks, crypto, etc.
- we need to combine overall money invested from different investment & show their current balance
  ↓ company
  this helps customers in deciding whether the company should give credits to customers or not.

① Data is not appended (history is kept. But it is removed (updated after usage

ODS parallel

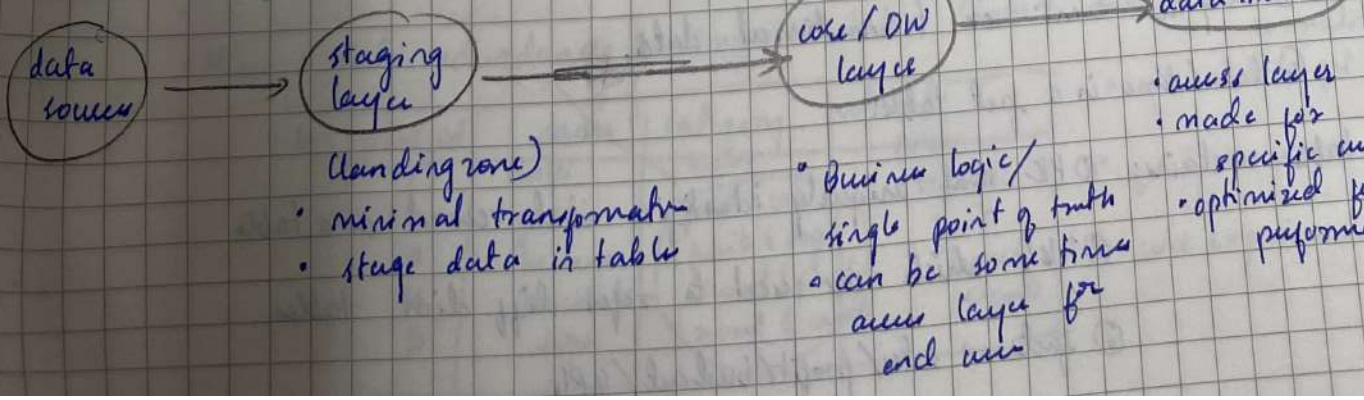


ODS sequential



ODS is getting less relevant because of advancement of HW & new tech for real time fast data.

Summary



Dimensional Modeling [unique way to structure the data]

[method of organizing data (in a data warehouse)]

fact
- measurement like profit.

dimensions
- context like category (period.

Eg: profit by year → give meaningful insight for data.

- fact is usually in middle connected by multiple dimension

Why dimensional Modeling?

Goal: fast data retrieval

- oriented around performance & usability.

→ wide table w/ many columns might have duplicate data.
This is not feasible for query performance.

Hence we make separate customer, product table &
retain only their IDs as foreign key in
fact table

⇓

this results in ↑ performance & ↑ usability.

(proposed technique for data warehouse!)

Fact: fact table contains important measurement of company like sales, unit sold, profit etc.

- foundation of DWH
- contains key measurement in a company.
- aggregated & analyzed.

fact are: ① aggregatable (numeric values, they are additive / can be added upto)
② measurable
③ they are event - or transactional data.
④ date / time in a fact table.

Fact table contains: ① PK that uniquely identifies each record in a table
② FK's which are used to refer diff dim tables
③ facts → sales / profit / budget / KPIs.

tructure

- fact table are defined by grain. → most atomic level are defined.

eg:

id	date_id	region_id	profit
1	2022/1/12	1	$23
2	2022/6/9	2	$100

fore specific region @ specific date, profit is defined.

∴ date is the most atomic value in this fact table

- different types of fact.

Dimensions: in star schema, dimensions are clustered around fact.

- their function is to categorize the facts.
- characters of dimensions: supportive & descriptive.
- used to filter, group & label our data.

→ dimensions are: ① not aggregatable (eg: date cannot be added 2019+2020 — no meaning)

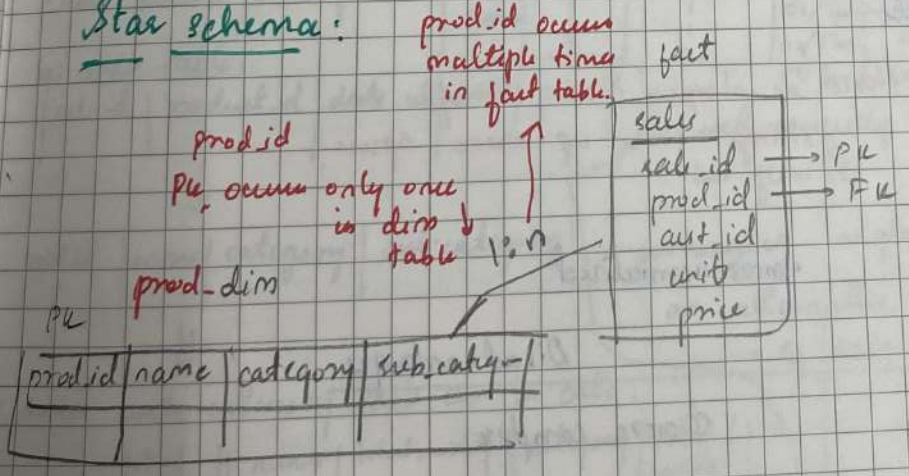
② descriptive in nature

③ more static (product name/categories) are not changing more than facts.

Dim table: contains → PK, FK & dimensions.

eg: people, product, place, time.

Star schema:



- there is data redundancy in dimension tables (data is denormalized & is not normalized)

eg:

prod_id	name	category	sub_cat
1	chilli	herbs	spices
2	garlic	fruit & veg	vegetable
3	banana	fruit & veg	fruit
4	chocolate	sweet & snacks	sweet
5	chips	sweet & snacks	snacks

- optimized to get data out
- query performance (read)
- user experience.

Normalized : ① technique to avoid redundancy
② minimizes storage.
③ write / read / update operations are easier
④ many tables
⑤ many joins & complex queries

Star schema is
① most common schema in data mart (usable & high performance)
② simplest form (vs. snowflake schema)
③ works best for specific needs (simple set of queries vs. complex queries)
④ usability + performance for specific (read) use-case.

Snowflake Schema

fact

sale_id	prod_id	cust_id	units	price

dimension: product

prod_id	name	category_id	sub_category
1	chilli	1	spices
2	garlic	2	vegetable
3	banana	2	fruit
4	chocolate	3	sweets

dimension: category

category_id	category
1	herbs
2	fruit & vegetables
3	sweets & snacks

stored as integers than strings as strings take up more space!

• more normalized

Advantage

① less data redundancy
② less disk space → less storage cost
③ easier to maintain / update data, less risk of corrupted data. due to no redundant data.
④ solve write / update slow downs

Disadvantages

① more complex
② more joins (more complex SQL queries)
③ less performance datamarts / cubes

Datamarts
star schema

Datamart

star schema

core (DW)

star schema (default)

may be snowflake schema (if freq update is thru / storage is concern)

fact :

| log_id | update_date (ID) | has_logg_id | project_ID | emp ID | div ID |

Dim ① : project ID
project_Name
project_priority

Dim ② : Employee ID | emp_name

Dim ③ : date_ID | year | month | quarter | day

Facts

Additivity in facts

Additive

- can be added across all dimensions
- most flexible & useful

eg: sale_table

| sale_id | product_id | date_id | unit | price |

↓

| prod_id | name | category | sub-category |

⇓

- across each category unit can be calc.

| date | total unit | name | total units | category | total units |

| sub-category | total_unit |

Semi-additive

- can be added across a few dimensions.

eg:	bal_id	portfolio_id	date	balance
today	1	1	01/Jan	50
	2	1	02/Jan	100
	3	2	03/Jan	100

- used carefully & less flexible
- averaging might be an alternative.
- eg: balance

Non-additive

- cannot be added across any dimension.

eg: price of a unit in sales table.

cannot get total price of category : it is unit price.

∴ we need to multiply unit * price = revenue.

eg: price
percentage
ratio (what is the inventory in warehouse).

- they have limited analytical value.

⇓

∴ store underlying values like numerator & denominator which can be calc from fact table.

Nulls in fact table

- Do not use null values as is. if present in foreign keys as it can be con error to join main table w/ referenced dim table

⇓

Hence, replace null w/ any default values like '999' / -1

Eg:

portfolioid
1
2
null
1
null

⇒

portfolio id
1
2
999
1
999

- replace null w/ 0 if calculating avg, sum, min

Year-to-date facts

- often requested by business users
- MTD, QTD, fiscal YTD etc.

⇓

these calculations are not in the defined grain of fact table.

⇓

fact table is @ a defined grain @ daily level.

⇓

Soln: store underlying values in defined grain

↓

like date value / month / year value.

↓

calculate to-date variations in BI tool

Types of facts

Transactional

- 1 row = measurement of event / transaction
- taken place at a specific time.
- 1 transaction defines the lowest grain.

sales — dim's

sale_id	prod_id	date_id	units

- typically additive
- have lot of dim (FK's) associated
- enormous in size
- most common & very flexible.

Periodic

- 1 row = summarize measure of many event / transactions.
- summarized of std. period (eg: 1 day, 1 week etc).
- lowest period defines the grain.

weekly_sales table

week_id	revenue	sales	cost

measures

- not large in size
- aggregated table (not very fine grained)
- growth is not rapid but controlled.
- typically additive data.
- has lot of facts & very few dim.
- If, no event = null or 0

Accumulating snapshot.

- 1 row = summarize measure of many event / transaction
- summarized of lifespan of 1 process (eg: order fulfillment)
- Has definite beginning & definite ending (& steps in between).

eg: start_dt, end_dt

- least common
- used for workflow / process analysis.
- Has many multiple date/time foreign keys (for each process step)
- date/time keys associated w/ role-playing dimension.

Type of fact table

Type	transactional	Periodic	Accumulating snapshot
① Grain	1 row = 1 transaction	1 row = 1 defined period (plus other dim)	1 row = lifetime of process/event
② Date dimension	1 transaction date	snapshot date (end of period)	Multiple snapshot date
③ No. of dimensions	high	lower	very high
④ facts	measure of transactions	cumulative measure of transaction in periods	measure of process is life
⑤ Size	largest (most detailed grain)	middle (little less detailed)	lowest (highest aggrega
⑥ Performance	can be improved w/ aggregation	Better (less detailed)	good performance.

Factless Fact table

A fact table can comprise of — fact (measures) → numeric
— dimensions (FK, FK)

Defn: Sometimes only dimensional aspects of an event are recorded → Factless fact table

Eg: reg_id | entry_date FK | dept_id | region_id | manager_id | Posid |

↓ has only event & no metrics.

① How many employees have been registered last month?

② How many employees have been registered in certain region?

Steps to create a fact table:

① Identify business process for analysis.

Eg: sales, order processing

② Declare the grain

Eg: transaction, order, order lines, daily, daily + location

↓ rather use more fine grain (higher atomic level)

↓ as it have options to analyze from all aspects of data whereas pre-aggregated data can be used only upto limited use-cases.

③ Identify dimensions that are relevant (what, when, where, how and why) → used for filtering and grouping.

Eg: time, location, product, customer.

④ Identify facts for measurements.

Surrogate keys : artificial keys (integer no. assigned to every single row).

→ **Natural keys** : come out of source system / source generated.

Eg: sale_id → PX0123 → alpha numeric values that take heavy space

⇒ **Surrogate keys** : are generated by database / ETL tool

Benefits → ① much smaller in size

② improve performance (less storage / better joins)

③ handle dummy values (nulls/missing values by using default values like 999 or -1.

④ integrate w/ multiple source systems.

⑤ easier to administrate/update

⑥ sometimes there are even no natural keys available.

① always use surrogate keys as PK or FK.
② Both for fact & dimension (except date dimension) } guideline
③ optionally keep natural keys

Case Study: E-commerce

Corporate IT

E-commerce company.

① 3 websites
② Each website is operated independently by multiple department

Step1: Identify the Business Process.

sales transaction — which product sold
what is sale profit
sales of each website
performance on different days.
sale over time

Step 2: Declare the Grain → (level of detail)
↓
what 1 row of fact table should represent.

• most analytical value comes w/ atomic grain.

Step 3: Define/Identify dimensions

Eg: make datetime_pk
customer
product

Step 4: Identify the facts : must comply w/ grain.

to check • if facts are additive (ie that can be agg across all dims)
* can use & absolute discount price
* do not store discount percentage
* calc profit

- Tables used to slice and dice data.
- Always has a primary key (PK)

ProductID	Name	category
P001	abc	A
P002	def	B
P003	ghi	C

- use surrogate key (eg: prod_PK → 1, 2, 3)
  ↓
  used to reduce space
  usually integer

Date Dimension

contain date related features: year, Month (name & number), day, quarter, week, weekday (name & number)...

- meaningful surrogate key: YYYYMMDD

  Eg: 2022-04-02 ⇒ 20220402

- Extra row for no date/null (source) ⇒ 1900-01-01 (dim)

- Time is usually a separate dimension.
- can be populated in advance (eg: for next 5 to 10 years)

Date features:

① Numbers & text (eg: January 1)

② long & Abreviated (Jan, January - Mon, Monday)

③ Combinations of attributes (Q1, 2022-Q1)

④ Fiscal dates (fiscal year etc)

⑤ Flags (weekend, company holidays etc).

Nulls in Dimensions

- Null must be avoided in FK's → as they break referential integrity
- They don't appear in Joins.

Eg: Promotion table

Promo_PK	promo_name
social media	1
website	2
no promo	-1

date table →

date_PK	date
20220101	1/1/2022
19000101	1/1/1900

- Replace nulls w/ descriptive values.

Hierarchies in Dimensions

- snowflake schema (normalized data)
  where a dimension is drilled down by another
  Eg: category → product

prod_id	product	cat_id

cat_id	category
1	groceries & electronics
2	

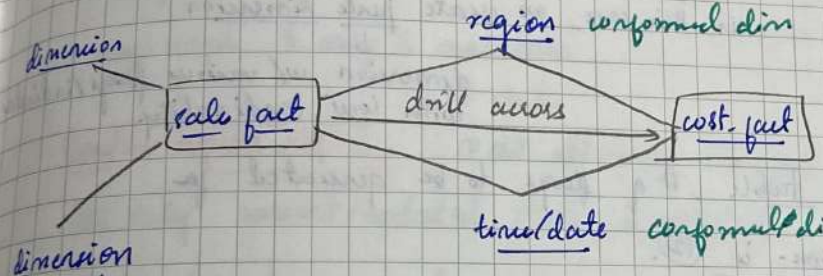
but this should be avoided → lead to lower performance & usability.

① one should always denormalize / flatten the data for DW.

② combine the hierarchies & collapse

Conformed Dimension: a dimension shared used by multiple fact tables / stars.

- used to compare facts across different tables



- same granularity is not necessary for conformed dim in 2 fact table

Eg: cost fact

m1	m2	date_fk
		20220101

sale table

sale_fact_fk
2022-01
2022-02

Degenerate Dimensions: A dimension that doesn't have separate dimension table, but still functions as dimension. (or associated dimension)

Eg: transaction sale fact

transaction_pk	amount	payment_fk
		263-023
		263-023

(here no separate table ∴ (descriptive value))

payment_fk	header
263-022	A
263-023	B
263-024	C

- All relevant information have already been extracted. (to other dimensions)
- Attribute can still be useful (eg of payment and)
- indicate that it is a not a FK & no dim table is associated w it.
  (_DD)
  Eg: payment_DD.
- Occurs mostly in transactional facts

Eg: Invoice no, billing no, order id no

Junk Dimension

Eg: transaction table

category (3)	Is_Bonus (2)	incoming/outbound (2)
A	Yes	incoming
B	No	outbound
C	Yes	incoming

- Eliminate them if not relevant
- If relevant, leave them in fact table.
- If the dim is long text value / long table size
  - ↓ create flag for each dimension
  - ↓ but this would be an issue if table has many columns / is wide
  - ↓ in this case, we create junk dimension
    - ↓ dimension w/ various flags/indicators with low cardinality. (few unique values)

Eg: for above transaction table, # of flags to be generated for junk dimension is (12).

But what if we have many dimensions?

9 indicators w/ 4 combinations (also dimensions)

↓

total ~~rows~~ value/unique flags for junk dim = $4^{9} = 262144$

↓

① In this case, we only extract the available combinations of fact table.

② or split up into 2 or more junk dimension.

Eg: $4^{5} = 1024$.

Role Playing Dimensions

A dimension that is referenced multiple times by diff fact table. Eg: date.

Eg: Prod table

id	order date FK (date FK)	# products	product_FK	product_start_FK (date FK)

role 1 → order date FK; role 2 → product_start_FK

left join

date_PK | date | month | short month | year-quarter | year | weekday | is_weekend

can analyze ① dt of product ② # of orders made on that dt.

Slowly changing Dimensions

① How to deal w/ changing dimensions in DWC (Dimensions are usually static):
② Be proactive: Ask about potential changes.
③ w/ Business users & IT
④ strategy for each changing attribute
⑤ SCD was introduced by Kimball in 1995 & distinguished b/w diff types (1,2,3..)

Type: Type 0 - original

- retain original data
- Applicable if there are no changes occuring in our dimensions.
- Usually applicable for date tables (except company holidays).
- 'Original'
- Very simple & easy to maintain → is static

Type 1 - overwrite type : ① old attributes are just overwritten w/ new values.
② Only current/updated state is reflected.

Eg:

prod_key	name	category
1	sunglasses	accessories
2	chocolate bar	cookies
3	oats	Biscuits

⇒ updated version

prod_key	Name	
1	sunglasses Re-7	accessories
2	chocolate bar - 20%	cookies
3	Delicious oats	Biscuits

only this updated version is maintained.

- Very simple to implement
  ↓
  as we only update values in the dimension tables.
- No fact table needs to be modified. (only dimension table is modified)

Cons: ① History is lost
② insignificant changes
③ might affect/break existing queries (eg: if case when queries are hardcoded).

Type 2 - Additional row (New row (powerful type)

- Purely partitions history
- Changes are reflected w/ history.

(default strategy).
to maintain history

- Instead of updating same row, create a new row w/ updated values!

Type 1:

Prod_key	Name	category
1	sunglass	accessories
2	choco bar	sweet
3	oat meal bis	sweet

↓ updated

prod_key	name	category
1	sunglass	accessories
2	choco bar	sweet
3	delicious oat meal bisc	Biscuit

Type 2:

prod_key	Name	Category
1	sunglass	accessories
2	choco bar	sweet
3	oat meal bisc	sweet

↓ add row

prod_key	Name	Category
1	sunglass	accessories
2	choco bar	sweet
3	oat meal bis	sweet
4	delicious oat meal biscuit	Biscuit

↓

to get # of products?

↳ do count prod_ID natural key (since generated does not change)

In queries only new foreign key for 'oat meal biscuit' - i.e 4 will be used in fact table

↓

hence we dont need to make any changes in fact table.

↳ hence respecting the history.

Eg:

Prod_FK	Prod_ID	name	category	ef_date	exp_date
1	SG-P07	sunglass	accessory	2022-01-01	2100-01-01 → default value (showing still valid)
2	CH-B20	choco bar	sweet	2022-01-01	2100-01-01
3	OT-BSC	oat meal	sweet	2022-01-01	2022-05-31
4	OT-BSC	del oat mel	Biscuit	2022-06-01	2100-01-01

used instead of null

↳ used to filter date ranges.

Is_current
Yes
Yes
No
Yes

↓

count of distinct prod_IDs → gives # of products.

Administering Type 2 SCD

how to get & identify current product name based on prod_ID

↓

hence include effective date & expdate to show period in which values are valid.

- correct FK must be used in fact table.
- requires surrogate key instead of natural key.

Correct FK in fact table: ① Add new row in dimension (ef_date, exp_date)
② lookup in dimension w/ natural_key + ef/ex_date.
③ Is_Current flag column can be used.

Mixing Type 1 and Type 2

- whether to use Type1, Type2 or both type1, type2 depends on dimensions.

Eg: prod_name can be type 1 (changing oat meal to 'delicious oat meal')

But for category, we need to maintain both old & new category.

↓

this is not a technical decision.

↓

need to ask business users & then define

Type 3: Additional Attribute

↓

is in-between: switching back and forth between — static previous state and static current state.

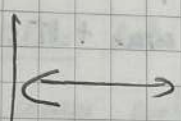
- Instead of adding row, we add a new column.

Eg:

prod_PK	prod_ID	Name	category	Prev_category
1	SG-T27	sunglass	accessories	accessories
2	CH-B70	choco bar	sweets	sweets
3	OT-BSC	oat meal	Biscuits	~~Biscuits~~ sweets

Current state

category	amount
accessories	$25
sweets	$6
biscuits	$8



Prev state

prev. category	amount
accessories	$25
sweets	$14

- typically used for significant changes at a time (eg: restructurings in organization)

region_PK	region	prev_region
1	North	North
2	West	West
3	South	West
4	East	not applicable

↓

can add ~~multiple~~ additional rows

- instead of adding a row, we add a column
- typically used for significant changes @ a time
- enables switching b/w historic & current view.

Limitations

① not suitable for frequent/unpredictable changes then → go for type 2

② minor change → type1

Understanding ETL Process

Extract

- Data is part of DWH
- Understanding data
- data is transformed from staging layer to core layer.
- Data is transient (most commonly)
  ↓
  i.e once all the data is copied from staging to core layer, & all transformations are complete, the data is then deleted / truncated from staging layer.

Extracting types

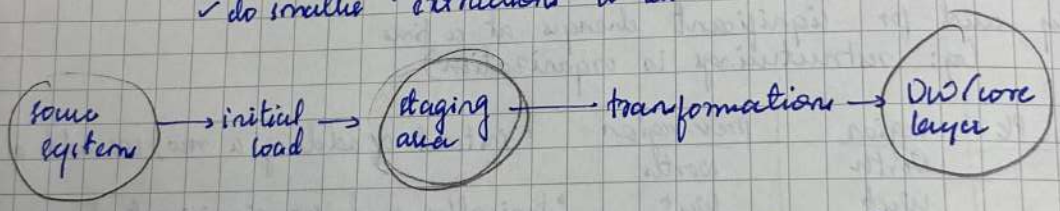
Initial load
- first (real) run
- all data

Delta load
- subsequent runs
- only additional data is loaded.

Initial load: ① extract all initial data from source systems

② After discussion w/ business user + IT:

- ✓ check what data is needed
- ✓ when is a good time to load data (night or weekends?)
- ✓ do smaller extractions to test



- After all transformation steps have been designed

Delta load: Incremental periodic extraction / load.

- there is a delta column for every table.
  → a column to identify new data.

sales_date	Name	amount

- transaction_date, create_dt etc can be used as delta columns.
- alternate columns such as primary_key can be used to identify new data

eg:

sales_key	name	Amount
1	sunglasses	$25
2	choco bar	$3
3	oat meal bisc	$4
4	choco bar	$3
5	oat meal bisc	$4

⇒ ✓ MAX (sales_key)
↓
✓ MAX (sales_key) → variable X

✓ Next run: sales_key > x

- Some tools can capture what data has been loaded & what has not been loaded

alternative case: for dim table (not fact table).

① Perform full load everytime

② compare data w/ data that has been already loaded

③ Depending on data volume → do performing of data load.

More data load = longer ETL process.

Load Workflow (insert/update)

- data load from staging to core/DWH layer

insert/update

- insert is done when the data is new (& was not loaded before)
- update: updating/replacing product name in product table,

if name exist or

if name does not exist, then add product name in product table

- Typically we do not delete data
- DELETE: if row in source system gets deleted

then do not remove row from DWH

add a new column w/ flag whether the row is active/valid in source system or not.

Transforming Data

Goal: make a consolidated view of all data for analysis purpose

① Consolidate (from multiple systems)

② Reshape (for analytical purpose)

① Consolidate:

table 1

Transaction_ID	amount	date
T1	$5030	10/1/2022
T2	$5053	11/1/2022
T3	$654	12/1/2022

table 2

transaction_ID	Amount (in thousands)	transaction_date
T14	$5.345	10-1-2022
T15	$7.953	11-1-2022
T16	$9.654	12-1-2022

- Making data compatible & consistent w/ each other.

② Reshape data according to business requirement

eg: ① date format: 10/1/2022 to 20220110

② pivot the data

Kinds of transformations

Basic : ① Deduplication
② Filtering (rows & columns)
③ cleaning & mapping (integration)
④ Value standardization (integration)
⑤ Key generation.

Advanced : ① Joining ② Aggregating
③ Splitting ④ Deriving new values.

Basic transformation: ① Deduplication: Merge 2 store table that have same product in both table
(remove duplicates) ↓
we distinct values

② Filtering rows: filter out irrelevant rows (eg: when amt < 0 or type = 'refund')

③ Filtering columns: If a column in a table contains same value in all row
(has 1 unique value)
↓
then delete the column

② cleaning & Mapping (integration):

map different values: Eg:

Name	gender
ABC	M
def	Fe
xyz	F

→ clean

Name	gender
mno	male
pqr	female
rst	male

replace M => Male
F => female

map/replace null values w/ 0's

Eg:

sale
300
100
null

⇒

sale
300
100
0

Value standardization:

month	sale in thousands
Jan 22	$1.5
Feb 22	$4.55
March 22	$3.321

→

month	sale
Jan 22	$1500
Feb 22	$4550
March 22	$3321

key generation: Add an auto-generated surrogate key

↓

(generated in ETL system or DBMS)

Advanced Transformation:

① Joining

Product Dim

prod_PK	prod_id	name	category	ef_date	exp_date
1	PS21	Almonds	nuts	2021-01-01	2121-01-01
2	PS22	garlic	fruit & veg	2021-01-01	2121-01-01
3	PS23	banana	fruit & veg	2021-01-01	2121-01-01
4	PS24	choco	sweet & snacks	2021-01-01	2121-01-01
5	PS25	chips	sweet & snacks	2021-01-01	2121-01-01

→ days and exp_date

Sale_fact

Sale_PK	product_id	date	prod_FK
3	PS33	2022-01-01	6
4	PS21	2022-01-01	1
5	PS22	2022-01-02	2
6	PS29	2022-01-02	9
7	SS30	2022-01-02	10

→ product surrogate key introduced as FK in sale fact table

② Splitting a column into 2 columns:

Store dim

store_id	location
1	New York, NY, 10011
2	Orland Park, IL, 06062
3	Houston, Tx, 27002

→

store_id	city	State	zip
1	New York	NY	10011
2	Orland Park	IL	06062
3	Houston	Tx	27002

- split by length/position

④ Aggregations

- sum of sales
- count number of sales
- distinct count of sales
- average
- subtract / multiply
- calc abs amount of tax using tax % & sale value.

sale_date	Name	Amount
2022-06-06	sunglass	$25
2022-06-06	choco bar	$8
2022-06-07	oat meal bisc	$4
2022-06-07	choco bar	$3
2022-06-08	oat meal bisc	$4

⇩

sale_date	Eq sales	Amount
2022-06-06	2	$28
2022-06-07	2	$7
2022-06-08	1	$4

Scheduling Jobs

source —extract→ staging —transform + load→ core

- require enterprise version in ETL tool (free version not possible)
  ↓
  use external tool

Scheduling : ① How frequent to update data? 1x, 3x/day, → ask business users
every 30 min

② How long does it take? 5 min? 1hr?

③ What is a good time to execute ETL? (∵ production source data might slow down if we use resources when there is heavy load)

need to be figured out w/ responsible people
- initial load vs delta load
- effect on production system.
- short read users
- night? early morning?

Demo: Plan of Attack

① Look at problem & plan

② Set up tables & schemas

③ Op staging table (& truncate)

④ Transform + Load: ⓐ read data from staging
ⓑ transform (clean + extract)
ⓒ update / insert

Problem

No new values ⇒ staging will be empty afterward & before the next run.

Empty staging ⇒ max(product_id) = null ⇒ full load.

ETL Tools

Enterprise
- commercial
- most mature
- ✓ graphical interface
- ✓ architectural needs met
- ✓ support

Eg: Alteryx, Informatica, Oracle data integrator, Microsoft SSIS.

Open-source
- source code
- ✓ often free
- ✓ Graphical interface
- ✓ support

Eg: Talend open studio, Pentaho data integration, Hadoop

cloud-native
- cloud-tech
- data already in cloud?
- ✓ efficiency.
- ✓ flexibility (can we to get data from other service providers)

Eg: Azure data flow, AWS Glue, Google cloud data flow.

Custom
- own development
- customized
- internal resources
- maintainence?
- training?

Choosing right ETL tool:

① Evaluate current situation (needs in company)
- a. What do you want to improve
- b. Data sources & other tools
- c. Define your requirements
- d. Define responsibilities.
- e. Who are the users?

Table to create while evaluating ETL tool

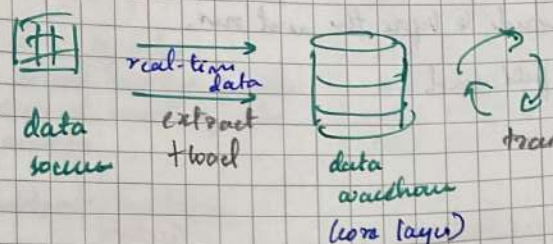
	context	must haves	weighting importance (1-5)	rating (1-5)
cost				
connectors				
capabilities			not very imp ✓	✓ super imp
ease of work/use				
reviews				
support/extras				

② Evaluate Tools

total weighted score:

③ Contact providers to get test/demo/free trial for POC.

ELT (Extract Load Transform)



Leverage compute power

[Since transformation is done in data warehouse this target data warehouse must have high compute power]

- use case for transformation
  Eg: Snowflake (can massive parallel processing)

- use when streaming data sources are used. (when real-time processing is used)

ETL

- more stable w/ defined transformations (based on defined transformation from stage → core)
- more generic use-cases
- security can be handled alot better (by agg send only anonymized / encrypted data to DW)

Use-cases: ① reporting (no real time req / defined transformation)

② generic use-cases

③ easy to use (default strategy)

ELT

- requires high performance target DB (but cloud platforms are easy)
- more flexible
- transformations can be changed quickly
- real-time requirements
- use cases: ① Req from data scientists, ML
  ② real-time req
  ③ big data & high volume

Data Warehouse Use cases:

DW: centralized data location for the purpose integrated data that is optimized for the purpose of data analysis

① used for reporting (to make data driven strategic decision) from diff data sources that are integrated.

② Performance for analysis (by quickly & easily connecting to DW by Business users

③ Data quality, accessible

④ continuous training of ML models for predictive analytics.

⑤ Use big data: Aggregate & filter

Optimizing A Data Warehouse

Issue: Business users facing bad query performance (after setting DW)
↓
wait long for result to be retrieved

Using index

① Data in DW disk space is not stored in a particular order
↓
it is stored wherever some free space is available
↓
this becomes a problem if we want to retrieve specific data.
Eg: select product_id from sale where cust_id = 5;
↓
table scan & read becomes inefficient.

fact table

Eg:

trans_id	prod_id	cust_id	payment	area
		4		
		5		
		5		
		8		
		5		

⇒ query: select prod_id from fact_table where cust_id = 5

result is large scan
∴ create index based on customer_id

location	value (cust_id)
1	4
2	5
5	8

↓
go to location using pointer assigned

disk_space ordering

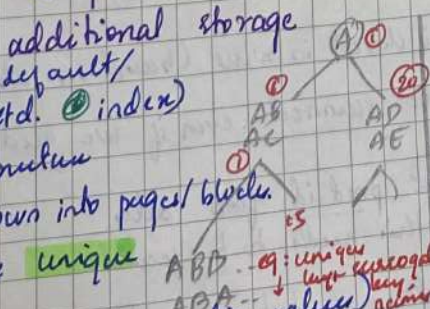
1, P0494, (4), usa → ①
6, P0058, (5), mauritania → ②
3, P0625, 5, usa
2, P0221, 5, usa
4, P0632, (8), mauritania → ⑤

search for cust_id in rows ⇒ 2, 3, 4 (2-4)

- makes data retrieval faster
↓
no need of full table scan

Cons: ① writing/updating data is slower
as pointer keys need to be maintained
② needs additional storage

① B-tree indexes: (std. default index)
- multi-level tree structure
- pages break data down into pages/blocks
- used when we have unique values in columns (high cardinality - high range of diff values)
- Never put an index on all columns (as it is costly in terms of storage, slower updates (writes).
- automatically set up on PKs (primary keys)



② Bitmap indexes (good for many replicating values)
(good for DW)
- used when data is large + low cardinality
- very storage efficient.
- data is stored in so called bit
- hence storage efficient
- not efficient for write/update
- very good (optimized for read operations & few DML operations

Bitmap - Index

trans_id	prod_id	cust_id	payment	price
1	P0494	4	visa	18.2
2	P0221	5	visa	1.5
3	P0625	5	visa	5.9
4	P0431	8	mastercard	11.6
5	P0058	5	mastercard	12.4

→ we use bit instead of list

row_id	value
1	visa
4	mastercard

Bit
11100
00011

value	1	2	3	4	5	6	7	8
mastercard				x	x			
visa	x	x	x					

① Should we put index on every column?
- ⓐ No, they come w/ a cost
- ⓑ storage + create/update time
- ⓒ should be used when necessary
  - ↓ (slow query performance due to full table scans)
- ⓓ small table do not require indexes

② On which columns should we place index?
- ⓐ To those columns & to those tables, that often used to have data filtered
  - ↓
  - Eg: if filtering is just done on 10% table then full table scan is avoided → if index is used.

Fact table
- ① B-tree on surrogate key (PK)
  - ↓ set up automatically
- ② Bitmap index on foreign keys.
  - ↓ often used to filter data, join columns → increase performance
    - ↓ for low cardinality (if large range of values / high cardinality then B-tree index can be used).

Dimension table
- ① size of the table (small table → no index) → if large table (millions of rows) then?
- ② Are they used in searches alot
- ③ choose based on cardinality

SYNTAX: (postgres syntax)

```
Create index index_name on table_name [USING method]
(
    column_name [ASC|DESC],
    ...
);
```

if there are 100 columns & only 5 columns are needed → then we don't have to process all 100 column data → 5% of data need to be processed.

Columnar Storage (used to improve query performance): ① traditionally data is stored in rows (based on above ex of index)
- ↓ all rows need to be scanned, even if we need to check 1 col of data

1, 2, 3, 4, 5
P0494, P0221, P0625, P0431, P0058
4, 5, 5, 8, 5
visa, visa, visa, mastercard, mastercard
18.3, 1.5, 5.9, 6, 12.4

Eg: select prod_id from sales → only this needs to be scanned.
- less data needs to be processed
- storage is less : data type stored is same → hence can be compressed efficiently
  - ↓ faster processing

The Modern Data Warehouse

On-premises DW

- Need to have own local hardware including: ① storage layer ② compute layer ③ software layer
- Need to have a physical data center

Benefits: ✓ full control (owns full infrastructure, hence own data governance rules)

Problems: ✓ full responsibility: (Buy if need for large workload, maintain)
✓ high cost for update & maintainance
✓ more internal resources
✓ less flexible
↓
If less workload, then there is wastage of resources which cannot be sold easily)

Cloud DW

- software-as-a-service: Not owning infrastructure but pay for using infrastructure
  ✓ pay for what you use
  ✓ managed services
  ✓ optimized for scalable analytics.

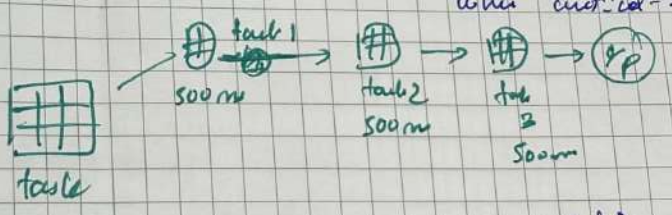
Benefits: ✓ fully managed
✓ scalable (can opt for additional subscriptions, resources)
✓ cost-efficient.
✓ managed security (better than on-prem)
✓ availability (w/ 99.9% uptime)
✓ implement quickly (time to market is quick)

Problems:
① regulations can be difficult to full fill in case of special needs.

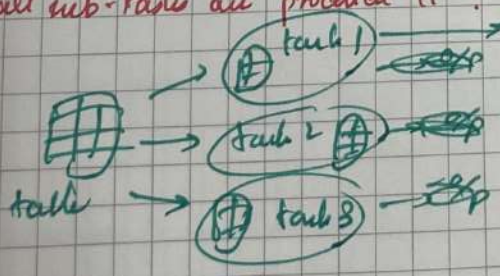
Most companies hence opt for cloud DW.
↓
they use technologies like MPP, columnar storage.

Massive Parallel Processing (MPP): Eg: if a query needs to be run: select * from sales where cust_id = 5;

thus in traditional systems
a big task is broken down to smaller tasks & are executed in sequential manner
↓
can be time consuming (need to wait until previous subtasks are completed)



In MPP, all sub-tasks are processed parallely. No need to wait for other sub-tasks to complete!

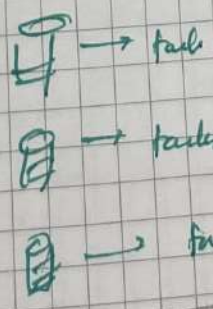


each nodes work independently (compute resources are not shared but disk is shared)

shared disk-architecture

- solves performance issues
- millions of rows → processed faster
- many people can run queries @ same time w/ good performance
- helpful w/ centralizing a massive amt of data.

- data storage is distributed to multiple data storage layers & compute nodes also work independently



shared-nothing architecture
(work load is split & processed individually)