

DATA SCIENCE e TECNOLOGIE

per le BASI di DATI

0) INTRODUZIONE

I **BIG DATA** sono dati la cui **dimensione** (tanti dati che scalano nel tempo), **diversità** (eterogenei) e **complessità** (spaziale e temporale) richiedono nuove architetture, tecniche e algoritmi che li gestiscano e che possano estrarre conoscenza da essi (usando viste personalizzate in base al target).

L'aggregazione di dati di natura diversa (es. sensori, immagini, video...) risulta complessa e viene svolta con tecniche di **data fusion**. I dati memorizzati e storici in un **data center** devono essere elaborati prima di essere mostrati ad utenti.

Le **5 V** dei big data sono:

- **Volume** = dimensione dei dati memorizzati (cresce esponenzialmente col tempo)
- **Velocità** = è necessario che l'elaborazione dei dati sia veloce
- **Varietà** = i dati possono essere di formato, tipo e struttura diverse
- **Veridicità** = i dati devono essere sicuri e affidabili
- **Valore** = i dati raccolti devono essere utili a creare valore/vantaggio

Il **PROCESSO DI DATA SCIENCE** è composto di 4 fasi:



1. Generazione

- a. Registrazione passiva (dati strutturati come transazioni, vendite...)
- b. Generazione attiva (dati non o semi-strutturati come testi, immagini...)
- c. Produzione automatica (dati dipendenti dal contesto come dati dei sensori IoT)

2. Acquisizione

- a. Raccolta (pull based [web crawler] e push based [video sorveglianza e click-stream])
- b. Trasmissione (trasferimento ad un data center con collegamenti veloci)
- c. Pre-elaborazione (integrazione, pulitura, eliminazione delle ridondanze)

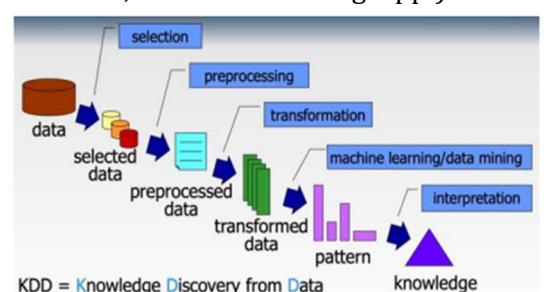
3. Memorizzazione

- a. Infrastruttura (memorie locali [HDD, SSD...] o di rete [DAS, NAS...])
- b. Gestione dei dati (file systems, struttura key-value, DB a colonne o documenti [MongoDB])
- c. Modelli di programmazione (elaborazione di stream e grafi)

4. Analisi

- a. Obiettivi:
 - i. Descrizione analitica (caratterizzazione del dataset)
 - ii. Analisi predittiva (predizione sulla base di un modello)
 - iii. Analisi prescrittiva (ottimizzazione sulla base del modello predittivo)
- b. Metodi: analisi statistica, machine learning (ML) e data mining, analisi associativa, classificazione e regressione, clustering (modellazione fatta sulla base dei dati stessi, divisione in sottogruppi)

Parliamo ora di **KNOWLEDGE DISCOVERY PROCESS**, ovvero l'estrazione dei dati con ML e tecniche di data science al fine di estrarre dai dati **informazioni implicite** e potenzialmente utili (estrazione automatica mediante **algoritmi**; le informazioni estratte sono rappresentate mediante **"pattern"** [modelli astratti]). È un processo iterativo: si fa prima su un campione e, se si ottengono buoni risultati, si estende a tutto il dataset. Questo ha alcune caratteristiche:



PREPROCESSING → questa fase si divide in:

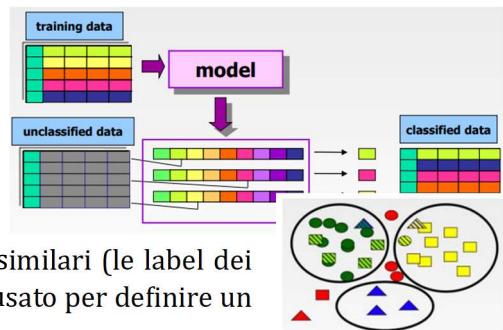
- o **Data cleaning** = riduzione dell'effetto del "rumore" sul campione, rimozione degli outliers e risoluzione delle inconsistenze;

- **Data integration** = unione dei dati estratti da diverse sorgenti (con risoluzione dei conflitti e gestione delle ridondanze) e integrazione dei metadati

TECNICHE DI DATA SCIENCE:

- **Regole di associazione** = usabile su tutti i dati, estraie correlazioni/pattern frequenti da un database transazionale (es. il suggerimento “spesso comprati insieme” di Amazon). È “**supervised**” in quanto abbiamo già una conoscenza pregressa dei dati. Una regola di associazione è composta da:
 - **Supporto** = % di record che contiene entrambi gli oggetti dell'associazione
 - **Confidenza** = quanto è forte l'associazione (tra i record che contengono l'informazione a sx dell'associazione, la % di quelli che contengono anche l'informazione a dx)

- **Classificazione** = “**supervised**” in quanto prevede di avere una lista di dati già classificati, usata per ottenere un modello per categorizzare i dati nuovi non ancora classificati.



- **Clustering** = “**unsupervised**”. Serve per trovare gruppi di dati simili (le label dei dati non sono note a priori), identificando eccezioni e outliers (usato per definire un campione rappresentativo)
- **Sequence mining** = viene considerato il criterio di ordinamento dei dati
- **Serie temporali e dati geospatiali** = considerate informazioni relative a spazio-tempo (es. sensori)
- **Regression** = predizione di un valore continuo (es. quotazioni in borsa)
- **Rilevazione degli outliers** (es. rilevare un'intrusione in un'analisi del traffico di rete)

TIPI DI DATA SCIENTIST:

- **Data expert** = strutturazione ed elaborazione dei dati
- **Data analyst** = data mining, statistiche e ML
- **Visualization expert** = visualizzazione grafica dei dati e story telling
- **Domain expert** = aiuta a comprendere il dominio dell'applicazione
- **Business expert**

Parlando invece dei **PROBLEMI DELLA DATA SCIENCE**: un problema è l'approccio “black box” degli algoritmi di AI (ML), ovvero le scelte che fanno non sono accuratamente giustificate (bias); per questo motivo si stanno sviluppando algoritmi di **explainable AI**, ovvero sono basati su alberi decisionali più “chiari”, c’è una spiegazione del modello e delle singole predizioni, e c’è una selezione di feauters interpretabili. Esistono **2 tipi di attributi correlati al bias in un algoritmo di ML**:

- Attributo **Bias** = direttamente discriminatorio (es. sesso, nazionalità, ...)
- Attributo **Proxy** = indirettamente discriminatorio (es. CAP, da cui si risale alla residenza)

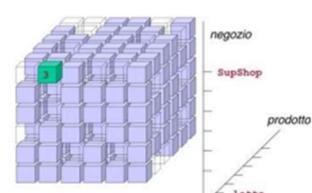
1) DATA WAREHOUSE

I dati operativi memorizzati da un’azienda possono contenere moltissime informazioni utili al supporto delle decisioni aziendali; questo è ambito della **Business Intelligence** (supporto alla decisione strategica aziendale; trasforma i dati aziendali in informazioni fruibili).

Una **DATA WAREHOUSE (DW)** è una **base di dati per il supporto alle decisioni**, separata dalle basi di dati operative dell’azienda, usata per migliori **prestazioni** e per migliore **gestione dei dati**. Vediamo le sue caratteristiche:

RAPPRESENTAZIONE:

- **Multidimensionale** = dati rappresentati come un “ipercubo” a 3 o più dimensioni, in cui gli assi sono le entità coinvolte e ogni intersezione contiene 1 o più misure (qualità, importo...) relative alle coordinate



- **Relazionale** ("a stella") = le misure presenti nella tabella dei fatti sono collegate alle entità mediante le relazioni

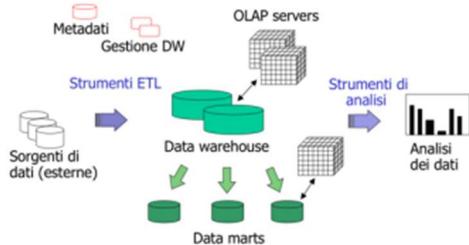


ANALISI DEI DATI:

- **Analisi OLAP** = calcolo di funzioni aggregate complesse; a differenza di OLTP (più orientato alla gestione delle transazioni operative [inserimento, aggiornamento e cancellazione dei dati]), si focalizza sull'analisi e sulla visualizzazione aggregata dei dati
- **Analisi dei dati mediante data mining**
- **Presentazione** = rappresentazione strutturata dei risultati di una ricerca
- **Ricerca di motivazioni** = esplorazione dei dati mediante approfondimenti

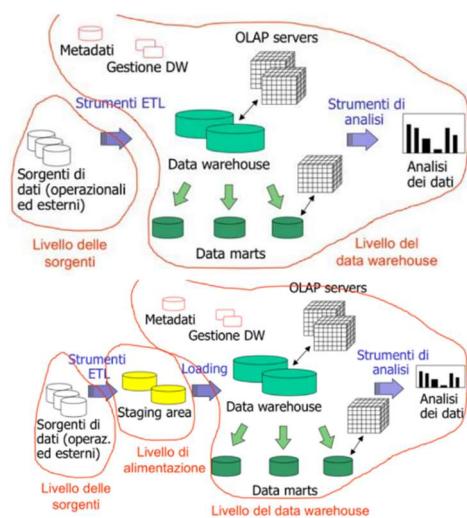
ELEMENTI COSTRUTTIVI:

- **Sorgenti di dati** = sorgenti eterogenee esterne al warehouse (es. excel, tweet...)
- **Warehouse aziendale** = contiene sia la modellazione del dato, sia strutture OLAP per interagire col dato
- **Data mart** = sottoinsieme dipartimentale focalizzato su un settore prefissato; può essere alimentato dal DW o direttamente dalle sorgenti
- **Server OLAP**:
 - **ROLAP** (Relational OLAP) = DBMS relazionale esteso con aggregazioni complesse; non ha vincolo sulla normalizzazione e può avere ridondanze
 - **MOLAP** (Multidimensional OLAP) = dati in forma matriciale (utile per dati densi)
 - **HOLAP** (Hybrid OLAP) = misto tra relazionale e multidimensionale
- **Strumenti ETL** = **ETL** è il processo di preparazione dei dati da mettere nel DW e viene eseguito sia al 1° popolamento sia all'aggiornamento periodico; 4 fasi:
 - **Estrazione** = acquisizione dati dalle sorgenti
 - **Pulitura** = miglioramento della qualità dei dati
 - **Trasformazione** = conversione dati dal formato operazionale a quello del DW (integrazione)
 - **Caricamento** = propagazione degli aggiornamenti al DW
- **Metadati** = informazioni relative ai dati ("dati sui dati") [sorgenti, attributi e relazioni tra i dati]; 3 tipi:
 - Trasformazione e caricamento = descrivono i dati sorgente e le trasformazioni necessarie
 - Gestione dei dati = descrivono la struttura dei dati presenti nel DW (derivati da liste)
 - Gestione delle query = dati sulla struttura delle query e monitoraggio della loro esecuzione



ARCHITETTURA:

- **A 2 livelli** (ovvero sorgenti + DW) → ha diversi vantaggi:
 - Disaccoppiamento dalle sorgenti (gestire dati esterni al sistema OLTP)
 - Facilità di gestione delle differenti granularità temporali dei dati
 - Separazione del carico transazionale da quello analitico
 - Preparazione dei dati "on the fly" (se l'operazione va male bisogna ricominciare dall'inizio)
- **A 3 livelli** (ovvero sorgenti + alimentazione + DW) → viene introdotta una "staging area" (ODS) che fa da transito e consente la separazione dell'elaborazione (ET) dal caricamento (L), permettendo operazioni di trasformazione e pulitura dei dati; introduce però ridondanza.



2) PROGETTAZIONE di DATA WAREHOUSE

Quando si progetta una DW bisogna tenere conto di alcuni rischi: aspettative elevate degli utenti, ridotta qualità dei dati e dei processi OLTP di partenza e gestione politica dei progetti. Ci sono 2 **APPROCCI** di progettazione:

- **TOP-DOWN** = DW costruito nel suo insieme fin dall'inizio, quindi progettazione complessa e costi/tempi significativi (non conviene con DW grandi)
- **BOTTOM-UP** = DW realizzato in maniera incrementale, partendo dalla costruzione di **data mart** su settori aziendali strategici per poi estendere a tutta l'azienda, quindi progettazione semplice e costi/tempi della prima consegna bassi

Il **BUSINESS DIMENSIONAL LIFECYCLE** rappresenta il flusso di progettazione di un DW:

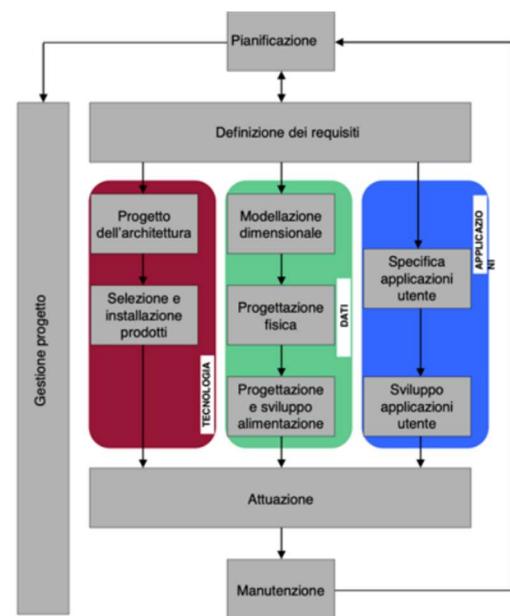
Definizione dei requisiti → 2 tipi di requisiti:

- Requisiti **utente** (appresi mediante "intervista" con gli attori del business)
- Requisiti relativi ai **dati** e alle **sorgenti**

Progettazione:

- **Tecnologia** (HW + SW)
- **Dati:**
 - Progettazione concettuale = modellazione dimensionale con cui devo definire le dimensioni di analisi, le metriche e le gerarchie delle dimensioni di analisi
 - Progettazione fisica = descrizione di come devono essere sviluppate le tabelle
 - Progettazione e sviluppo dei processi di ETL (caricamento iniziale e periodico)
- **Applicazioni** = sviluppo di applicazioni per la generazione di report, esplorazione interattiva dei dati e elaborazione dei dati con tecniche di data science

Attuazione → deploy del sistema; durante l'utilizzo, potrebbero uscire nuovi requisiti che ci riportano in fase di progettazione



La **PROGETTAZIONE DI DATA MART** è composta:

Caratterizzazione delle sorgenti (come testo, modello logico o modello ER)

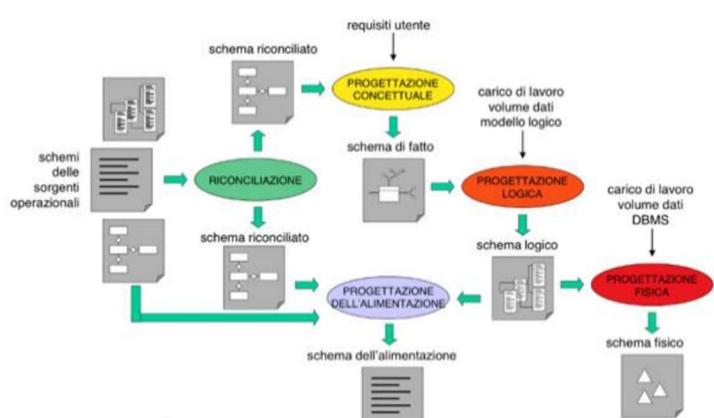
Riconciliazione = visione globale delle fonti

Progettazione concettuale = creazione del DW sulla base dei requisiti utente soddisfando la sopra citata progettazione concettuale (dimensioni, misure e gerarchie di dimensioni)

Progettazione logica = elaborazione di uno schema logico denormalizzato (contenente le tabelle e le relazioni tra esse), che tenga conto del volume di dati relativo alle query eseguite sul DW, in modo da prevedere un sistema ottimizzato

Progettazione del processo di ETL = produce uno schema di alimentazione composto da script

Progettazione fisica = strutture fisiche che possano ridurre il tempo di esecuzione delle operazioni



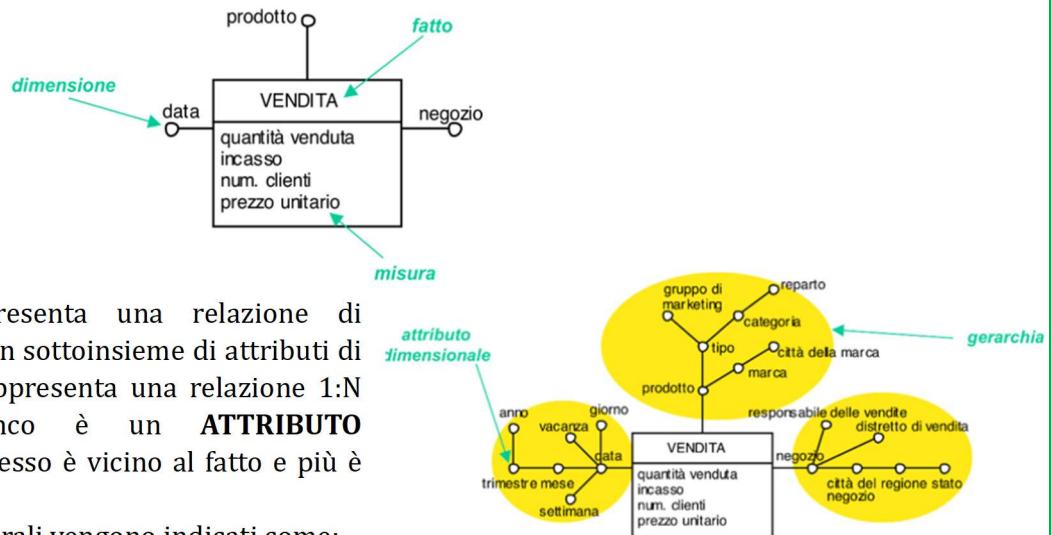
L'ANALISI DEI REQUISITI è fare interviste agli utilizzatori del sistema ("business users") e ai detentori delle informazioni delle sorgenti (amministratori del sistema informativo [IS]), e scegliere 1 o più data mart che faranno parte del progetto pilota (iniziale) [i data mart devono essere strategici e alimentati da poche sorgenti affidabili]. 2 tipi di requisiti:

- **APPLICATIVI** → descrivono gli eventi di interesse (i "fatti"), l'intervallo di storicizzazione (la "granularità temporale") e il carico di lavoro (la "dimensione") dei dati relativi al settore dell'azienda del data mart
- **STRUTTURALI** → **periodicità dell'alimentazione** (ogni quanto i dati vengono trasferiti dalle sorgenti al DW), **spazio disponibile**, **tipo di architettura** (la scelta del numero di livelli è legata alla complessità del processo ETL e al volume di dati periodici) e **pianificazione del deployment** (avviamento + formazione agli utilizzatori)

Vediamo nel dettaglio le progettazioni:

PROGETTAZIONE CONCETTUALE = rappresentazione di alto livello di come i dati vengono organizzati; viene superato il modello ER (troppo rigido e troppo legato alla normalizzazione): noi usiamo il **DIMENSIONAL FACT MODEL** (questo offre anche una documentazione utile per la revisione dei requisiti, al fine di comprendere a posteriori i motivi delle scelte fatte). È composto da:

- **FATTO** = elemento che modella un insieme di eventi di interesse [es. vendite, spedizioni...] in relazione al tempo
- **DIMENSIONE** = descrive una coordinata di analisi di un fatto [es data, negozio, prodotto...] e rappresenta una relazione N:N
- **MISURA** = descrive una proprietà numerica di un fatto [es. incasso...] (spesso oggetto di operazioni di aggregazione)



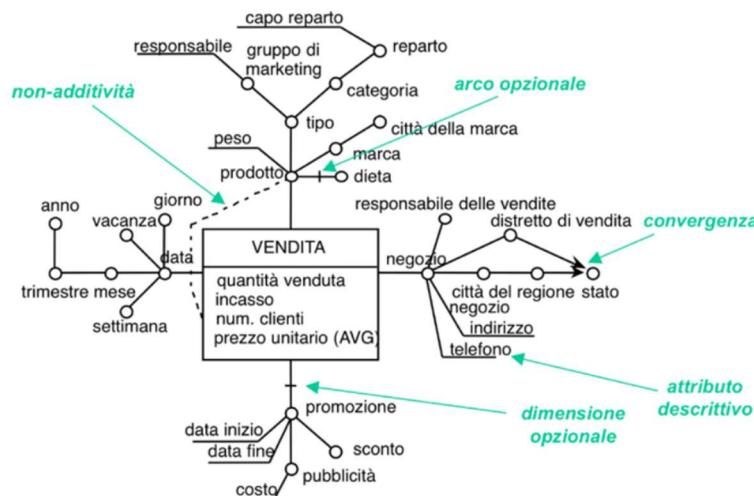
- **GERARCHIA** = rappresenta una relazione di generalizzazione tra un sottoinsieme di attributi di una dimensione e rappresenta una relazione 1:N (ogni pallino bianco è un **ATTRIBUTO DIMENSIONALE**, più esso è vicino al fatto e più è ritenuto specifico)

Gli attributi temporali vengono indicati come:

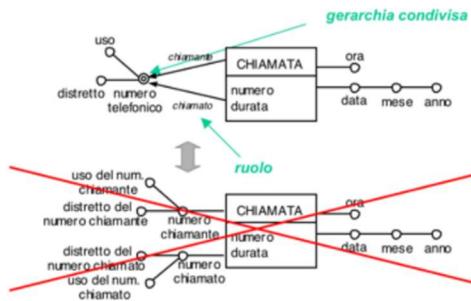
- Anno (2022)
- Mese (ottobre 2022, contiene sempre l'anno)
- Trimestre ([1-4] 2022, contiene sempre l'anno)
- Settimana ([1-52] 2022, contiene sempre l'anno)
- Giorno (5 ottobre 2022, contiene mese e anno)

COSTRUTTI AVANZATI:

- **Non-additività** (arco tratteggiato) = identifica una metrica e una dimensione non aggregabili (es. se prodotto1 ha 100 vendite e prodotto2 ne ha 200, la somma dovrebbe essere 300, ma ci sono dei clienti intersecati che verrebbero contati 2 volte)
- **Arco opzionale** (trattino) = attributo non disponibile per tutti i record (sostituisce la cardinalità di ER)
- **Convergenza** = 2 rami che convergono verso lo stesso attributo
- **Attributo descrittivo** (senza pallino bianco) = non viene interessato da operazioni come la group by, ma rappresentano del contenuto informativo dell'attributo a cui essi sono collegati
- **Dimensione opzionale** (trattino) = dimensione non disponibile per tutti i record del fatto (come l'arco)



- **Gerarchia condivisa** (freccia) = usare 1 gerarchia per 2 dimensioni di analisi

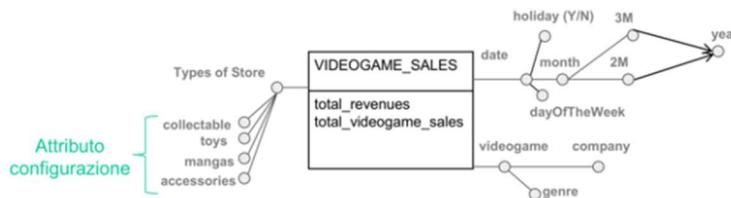


- **Arco multiplo** = rappresenta una relazione N:N tra 2 attributi dimensionali; 2 rappresentazioni:

- Arco multiplo tra la dimensione e il fatto: la tabella dei fatti presenterà N record per lo stesso fatto dovuti al numero delle configurazioni della dimensione (se il numero di attributi multipli è limitato si preferisce usare un'attributo "configurazione")
- La dimensione diventa un attributo di una nuova dimensione che rappresenta un **gruppo** di configurazioni della dimensione iniziale (usato per snellire la tabella dei fatti)



- **Attributo configurazionale** = rappresenta un attributo multivalore che è solitamente caratterizzato da pochi valori distinti (<=10), quindi rappresentabile come enumerazione dei suoi valori (rappresentati in tabella come colonne che possono assumere un valore booleano)



AGGREGAZIONE = processo di calcolo del valore di misure a granularità "meno fine" di quella presente nello schema di fatto originale; **classificazione delle misure in base a:**

➤ **Aggregabilità:**

- Additive (aggregabili lungo una gerarchia mediante l'operatore di SUM)
- Non additive
- Non aggregabili

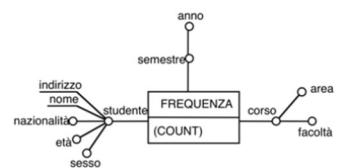
➤ **Tipo:**

- Misure di flusso = valutate cumulativamente alla fine di un periodo di tempo e sono aggregabili mediante tutti gli operatori standard (es. numero prodotti venduti)
- Misure di livello = valutate in specifici istanti di tempo e non sono additive lungo la dimensione del tempo (es. livello di inventario)
- Misure unitarie = valutate in specifici istanti di tempo e non sono additive lungo nessuna dimensione (es. prezzo unitario di un prodotto)

OPERATORI DI AGGREGAZIONE:

- Distributivi → sempre possibile il calcolo di aggregati da dati di dettaglio maggiore (SUM, MIN, MAX)
- Algebrici → calcolo degli aggregati da dati di livello di dettaglio maggiore è possibile in presenza di misure aggiuntive di supporto (es. AVG, richiede informazione sul COUNT)
- Olisitici → non è possibile calcolo di aggregati da dati di livello di dettaglio maggiore (es. moda, mediana)

Altro importante elemento della progettazione concettuale è lo **SCHEMA DI FATTO VUOTO**: il fatto rappresenta il verificarsi di un evento e non contiene misure al suo interno (utile per conteggiare gli eventi accaduti e rappresentare gli eventi non accaduti; la misura può essere solo 1 COUNT).



⚠ Vediamo alcuni elementi a cavallo tra progettazione concettuale e logica:

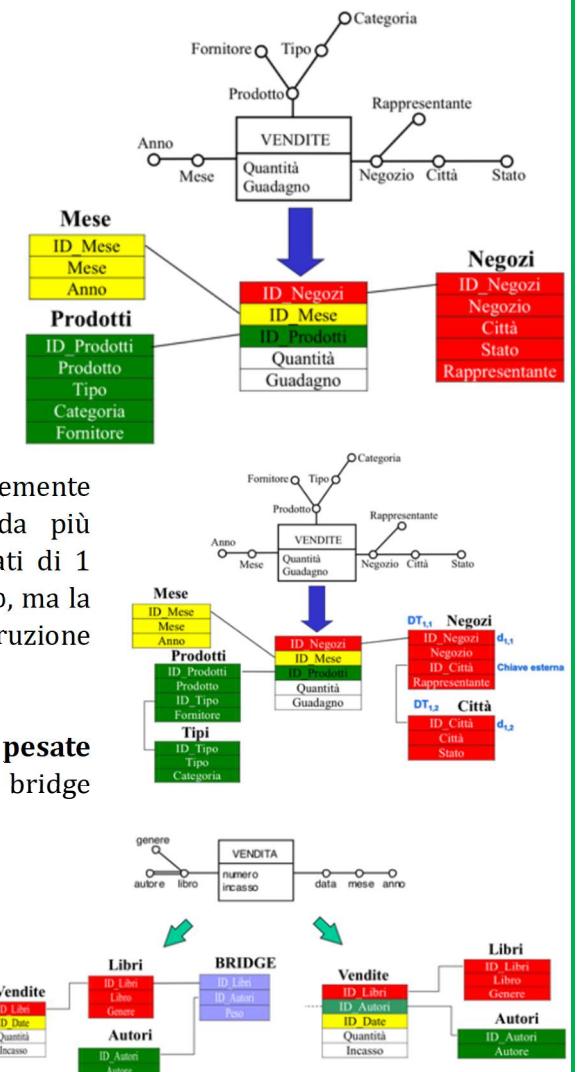
- ❖ **Variazione dei dati nel tempo** → gestita in 3 modi:
 - **Sovrascrittura** (rappresentazione del tempo di tipo 1) = il nuovo dato sovrascrive il valore attuale (non rimane traccia del valore precedente, né del cambiamento)
 - **Nuova istanza incorrelata** (rappresentazione del tempo di tipo 2) = per ogni variazione della dimensione viene creata una nuova istanza nella dimensione, correlandone i nuovi eventi
 - **Nuova istanza correlata alle vecchie** (rappresentazione del tempo di tipo 3) = vengono introdotti una coppia di timestamp (inizio e fine; per la gestione della validità del dato) e un attributo che consente l'identificazione della sequenza di variazione di una specifica istanza
- ❖ **Carico di lavoro** → il possibile carico reale previsto viene calcolato mediante report e stime discusse con gli utenti (stime che devono prevedere anche il successo del sistema e la variazione della tipologia delle interrogazioni nel tempo) [tuning = monitorare e migliorare il sistema post deploy]
- ❖ **Volume dei dati** → è importante la stima dello spazio che il data mart necessita per la memorizzazione dei dati e delle strutture fisiche accessorie; questa previsione viene fatta aggregando diversi fattori:
 - Numero di eventi per fatto
 - Numero di valori distinti negli attributi delle gerarchie
 - Lunghezza degli attributi
 - Intervallo di memorizzazione dei dati
 - **Sparsità** (numero degli eventi accaduti non corrisponde a tutte le possibili combinazioni delle dimensioni)

PROGETTAZIONE LOGICA = rappresentazione dello schema delle tabelle mediante il modello relazionale ROLAP; consente la ridondanza dei dati e la denormalizzazione delle tabelle. Schemi:

- **SCHEMA A STELLA:**
 - **Dimensioni:** viene creata una tabella per ogni dimensione, con 1 chiave primaria generata artificialmente (surrogata) contenente tutti gli attributi senza esplicitare gerarchie tra loro
 - **Fatti:** viene creata una tabella per ogni schema di fatto, con una chiave primaria costituita dalla combinazione delle chiavi esterne delle dimensioni; le misure sono rappresentate come attributi della tabella. Una dimensione opzionale può essere inclusa o meno nella chiave primaria (dipende dalle specifiche del progetto).
- **SCHEMA A SNOWFLAKE (SCONSIGLIATO;** usato più frequentemente solo quando una parte della gerarchia è condivisa da più dimensioni): separa dipendenze funzionali separando i dati di 1 dimensione in più tabelle: lo spazio necessario viene ridotto, ma la diminuzione della ridondanza rallenta la fase di ricostruzione dell'informazione della dimensione.

Le query che caratterizzano un arco multiplo possono essere **pesate** (considerano il peso dell'arco multiplo [peso = attributo della bridge table]) o **di impatto** (il valore aggregato viene calcolato senza guardare i pesi). Gli **ARCHI MULTIPLI** possono essere realizzati in 2 modi:

- **Bridge table** = viene aggiunta una tabella che modella la relazione N:N contenente un nuovo attributo che consente di pesare la partecipazione delle tuple nella relazione (adatto a query pesate) [il calcolo del peso viene fatto in fase di ETL, difficile cambiarlo dopo]



- **Push down** (**SCONSIGLIATO** perché separa il concetto del fatto su più righe) = l'arco multiplo viene integrato nella tabella dei fatti, così da aggiungere una nuova dimensione (aumenta quindi la ridondanza nella tabella dei fatti). Alcuni problemi sono che è difficile eseguire interrogazioni di impatto, il calcolo del peso viene fatto durante l'alimentazione e le modifiche successive sono difficoltose.

Una **DIMENSIONE DEGENERE** è una dimensione costituita da 1 solo attributo ed è rappresentabile in 3 modi:

- Trattata come una dimensione **normale** (ciò consente l'aggiunta di future gerarchie)
- **Integrata** nella tabella dei fatti (per attributi di dimensione contenuta) e nella rispettiva chiave primaria
- **Junk dimension** = in presenza di più dimensioni degeneri, queste vengono raggruppate figurando come un'unica dimensione; questa deve poter rappresentare tutte le possibili combinazioni delle dimensioni al suo interno (quindi usata solo per cardinalità di attributi limitata)

3) ANALISI dei DATI

L'analisi dei dati viene svolta usando 3 strumenti:

- Calcolo di **funzioni aggregate**
- Operazioni di **confronto**
- Analisi con **data mining**

La DW può essere interrogata usando ambiente controllato di query, strumenti specifici di query oppure strumenti di data mining:

- **AMBIENTE CONTROLLATO DI QUERY** = consente di fare ricerche, analisi e rapporti con struttura prefissata (i risultati vengono rappresentati mediante grafici e report personalizzati). Si possono introdurre elementi specifici del settore considerato ed è necessario codice ad hoc
- **AMBIENTE DI QUERY AD HOC** = è possibile definire interrogazione OLAP, progettate al momento dall'utente (utile se i report predefiniti non sono abbastanza)

L'ANALISI OLAP permette l'analisi dei dati con diverse operazioni che possono essere combinate tra loro nella stessa query e eseguite in sequenza in una stessa sessione OLAP:

- **Roll up** → **riduzione del dettaglio** dei dati:
 - aumentando il livello di gerarchia (cambio delle clausole della group by)
 - eliminando 1 delle dimensioni dalla group by

Metrics Customer Region	Dollar Sales					
Month	North-East	Mid-Atlantic	South-East	Central	South	North-West
Jan 97	\$ 620	\$ 753	\$ 30	\$ 660	\$ 2.405	\$ 1.312
Feb 97	\$ 258	\$ 252	\$ 800	\$ 975	\$ 160	\$ 582
Mar 97	\$ 646	\$ 244	\$ 148	\$ 250	\$ 1.085	\$ 2.961
Apr 97	\$ 787	\$ 588	\$ 447	\$ 486	\$ 226	\$ 506
May 97	\$ 1.350	\$ 245	\$ 936	\$ 159	\$ 664	\$ 626
Jun 97	\$ 842	\$ 582	\$ 1.281	\$ 937	\$ 240	\$ 774
Jul 97	\$ 652	\$ 690	\$ 466	\$ 1.293	\$ 605	\$ 303
Aug 97	\$ 1.783	\$ 304	\$ 1.032	\$ 170	\$ 398	\$ 356
Sep 97	\$ 581	\$ 778	\$ 3.558	\$ 587	\$ 440	\$ 1.652
Oct 97	\$ 2.291	\$ 1.840	\$ 600	\$ 656	\$ 1.300	\$ 718
Nov 97	\$ 39	\$ 1.602	\$ 1.082	\$ 1.187	\$ 842	\$ 759
Dec 97	\$ 381	\$ 1.588	\$ 343	\$ 118	\$ 1.459	\$ 635

Metrics Customer Region	Dollar Sales					
Category	Year	North-East	Mid-Atlantic	South-East	Central	South
Electronics	1997	\$ 138	\$ 1.774	\$ 384	\$ 138	\$ 2.346
	1998	\$ 1.184	\$ 4.529	\$ 1.892	\$ 7.232	\$ 651
Food	1997	\$ 759	\$ 682	\$ 729	\$ 262	\$ 588
	1998	\$ 538	\$ 925	\$ 959	\$ 677	\$ 213
Gifts	1997	\$ 2.532	\$ 1.355	\$ 1.854	\$ 1.413	\$ 2.535
	1998	\$ 1.955	\$ 2.785	\$ 2.800	\$ 2.695	\$ 1.813
Health & Beauty	1997	\$ 624	\$ 640	\$ 1.317	\$ 647	\$ 588
	1998	\$ 611	\$ 887	\$ 566	\$ 382	\$ 499

Metrics Customer Region	Dollar Sales					
Quarter	North-East	Mid-Atlantic	South-East	Central	South	North-West
Q1 1997	\$ 1.526	\$ 1.249	\$ 978	\$ 1.885	\$ 3.650	\$ 4.855
Q2 1997	\$ 2.979	\$ 1.415	\$ 2.664	\$ 1.582	\$ 1.130	\$ 1.906
Q3 1997	\$ 3.016	\$ 1.772	\$ 5.076	\$ 2.050	\$ 1.443	\$ 2.311
Q4 1997	\$ 2.711	\$ 5.030	\$ 2.025	\$ 1.961	\$ 3.601	\$ 2.112
Q1 1998	\$ 2.989	\$ 1.772	\$ 5.076	\$ 2.050	\$ 1.443	\$ 2.311
Q2 1998	\$ 1.773	\$ 3.659	\$ 1.862	\$ 1.213	\$ 1.115	\$ 4.635
Q3 1998	\$ 1.818	\$ 5.402	\$ 1.707	\$ 2.485	\$ 1.704	\$ 3.632
Q4 1998	\$ 2.091	\$ 2.968	\$ 4.664	\$ 5.917	\$ 1.775	\$ 3.162



Metrics	Dollar Sales
Category	Year
Electronics	1997
	1998
Food	1997
	1998
Gifts	1997
	1998
Health & Beauty	1997
	1998



Metrics Customer Region	Dollar Sales					
Quarter	North-East	Mid-Atlantic	South-East	Central	South	North-West
Q1 1997	\$ 1.526	\$ 1.249	\$ 978	\$ 1.885	\$ 3.650	\$ 4.855
Q2 1997	\$ 2.979	\$ 1.415	\$ 2.664	\$ 1.582	\$ 1.130	\$ 1.906
Q3 1997	\$ 3.016	\$ 1.772	\$ 5.076	\$ 2.050	\$ 1.443	\$ 2.311
Q4 1997	\$ 2.711	\$ 5.030	\$ 2.025	\$ 1.961	\$ 3.601	\$ 2.112
Q1 1998	\$ 2.989	\$ 1.772	\$ 5.076	\$ 2.050	\$ 1.443	\$ 2.311
Q2 1998	\$ 1.773	\$ 3.659	\$ 1.862	\$ 1.213	\$ 1.115	\$ 4.635
Q3 1998	\$ 1.818	\$ 5.402	\$ 1.707	\$ 2.485	\$ 1.704	\$ 3.632
Q4 1998	\$ 2.091	\$ 2.968	\$ 4.664	\$ 5.917	\$ 1.775	\$ 3.162



Metrics Customer Region	Dollar Sales					
Quarter	Arlin	San pedro	Springfield	Chappel Hill	Scrnburg	Pebble Beach
Q1 1997	\$ 675					\$ 50
Q2 1997				\$ 203		
Q3 1997					\$ 113	\$ 45
Q4 1997					\$ 192	\$ 348
Q1 1998					\$ 85	\$ 79
Q2 1998					\$ 12	\$ 37
Q3 1998					\$ 237	\$ 30
Q4 1998					\$ 119	\$ 119

- **Drill down** → **aumento di dettaglio** dei dati [grafici opposti alla roll up]
 - riducendo il livello di gerarchia (cambio delle clausole della group by)
 - aggiungendo 1 nuova dimensione alla group by

- Slice and dice → riduzione del volume dei dati da analizzare:
 - Slice = predicato di uguaglianza che seleziona una fetta lungo una dimensione di riferimento
 - Dice = combinazione di predici che seleziona una porzione di dimensioni

Category	Year	Metrics Customer Region	Dollar Sales									
			North-East	Mid-Atlantic	South-East	Central	South	North-West	South-West	England	France	Germany
Electronics	1997		\$ 138	\$ 1.774	\$ 384	\$ 139	\$ 2.346	\$ 2.554	\$ 2.184	\$ 566	\$ 199	\$ 1.98
	1998		\$ 4.529	\$ 1.892	\$ 7.232	\$ 1.251	\$ 1.767	\$ 5.289	\$ 1.265	\$ 2.187	\$ 2.22	\$ 7
Food	1997		\$ 759	\$ 682	\$ 729	\$ 262	\$ 588	\$ 469	\$ 807	\$ 156	\$ 615	\$ 1
	1998		\$ 538	\$ 925	\$ 959	\$ 677	\$ 213	\$ 1.503	\$ 261	\$ 165	\$ 175	\$ 1
Gifts	1997		\$ 2.532	\$ 1.355	\$ 1.054	\$ 1.413	\$ 2.535	\$ 2.132	\$ 1.904	\$ 908	\$ 375	\$ 1.0
	1998		\$ 1.955	\$ 2.785	\$ 2.800	\$ 2.695	\$ 1.813	\$ 2.844	\$ 1.778	\$ 1.158	\$ 717	\$ 6
Health & Beauty	1997		\$ 624	\$ 640	\$ 1.317	\$ 647	\$ 588	\$ 754	\$ 654	\$ 143	\$ 292	\$ 3
	1998		\$ 611	\$ 887	\$ 566	\$ 382	\$ 499	\$ 1.162	\$ 1.044	\$ 273	\$ 72	
Household	1997		\$ 5.354	\$ 4.112	\$ 5.410	\$ 4.446	\$ 3.058	\$ 3.974	\$ 2.654	\$ 3.454	\$ 2.875	\$ 2.7
	1998		\$ 5.787	\$ 5.320	\$ 5.416	\$ 6.812	\$ 4.334	\$ 5.008	\$ 7.588	\$ 2.139	\$ 3.649	\$ 2.7
Kid's Korner	1997		\$ 199	\$ 1.096	\$ 398	\$ 409	\$ 323	\$ 397	\$ 409	\$ 19	\$ 1	
	1998		\$ 247	\$ 422	\$ 441	\$ 380	\$ 221	\$ 295	\$ 200	\$ 198	\$ 19	
Travel	1997		\$ 624	\$ 505	\$ 564	\$ 306	\$ 300	\$ 978	\$ 416	\$ 48	\$ 30	
	1998		\$ 608	\$ 559	\$ 1.096	\$ 611	\$ 464	\$ 316	\$ 573	\$ 257	\$ 198	

Metrics	Dollar Sales	Customer City	Afton	Akron	Albon	Alameda	Alka	Allagash	Alta	Altoola	Amestra	Amsterdam	Andersonville	Annap
Subcategory														
Audio														
Automotive														
Chocolate	\$ 42	\$ 42												
Christmas	\$ 30													
Classic Toys														
Gadgets														
Comfort														
Furniture														
Games & Puzzles														
Gift Baskets														
Hobbies														
Hearth														
Jewelry														
Lawn & Garden														
Learning														
Meat & Cheese														
Natural Remedies														
Pet Supplies														
Plants & Flowers														
Safety & Security														
Skin Care														
Sleeping														
Toys & Accessories														

Filter Details:
Category = Electronics
AND
Year = 1998

Category	Year	Metrics Customer Region	Dollar Sales									
			North-East	Mid-Atlantic	South-East	Central	South	North-West	South-West	England	France	Germany
Electronics	1997		\$ 1.104	\$ 4.529	\$ 1.892	\$ 7.232	\$ 651	\$ 9.488	\$ 476	\$ 2.683	\$ 702	
	1998		\$ 1.184	\$ 4.529	\$ 1.892	\$ 7.232	\$ 651	\$ 9.488	\$ 476	\$ 2.683	\$ 462	\$ 7
Food	1997		\$ 538	\$ 925	\$ 959	\$ 677	\$ 1.503	\$ 261	\$ 165	\$ 175	\$ 615	\$ 1
	1998		\$ 538	\$ 5.638								
Gifts	1997		\$ 1.955	\$ 2.785	\$ 2.800	\$ 2.695	\$ 1.813	\$ 2.844	\$ 1.779	\$ 1.158	\$ 717	\$ 686
	1998		\$ 611	\$ 887	\$ 566	\$ 382	\$ 499	\$ 1.162	\$ 1.044	\$ 273	\$ 72	
Health & Beauty	1997		\$ 6.042	\$ 6.665								
	1998		\$ 6.042	\$ 38.383								
Household	1997		\$ 5.787	\$ 5.320	\$ 5.416	\$ 6.812	\$ 4.334	\$ 5.008	\$ 7.588	\$ 2.139	\$ 3.649	\$ 2.791
	1998		\$ 247	\$ 422	\$ 441	\$ 380	\$ 221	\$ 592	\$ 290	\$ 198	\$ 19	
Kid's Korner	1997		\$ 2.559	\$ 2.943								
	1998		\$ 4.497	\$ 4.792								

Filter Details:
Category = Electronics
AND
Year > 80
AND
Customer Region = North-West
AND
Year = 1997

Metrics	Dollar Sales	Customer City	Alta	Armstrong	Avery	Heights	Lane	Mt. Everest	San Francisco
Subcategory									
Audio									
Gadgets	\$ 98								
Comfort									
Furniture									
Games & Puzzles									
Gift Baskets									
Hobbies									
Hearth									
Jewelry									
Lawn & Garden									
Learning									
Meat & Cheese									
Natural Remedies									
Pet Supplies									
Plants & Flowers									
Safety & Security									
Skin Care									
Sleeping									
Toys & Accessories									

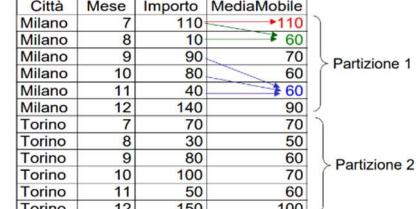
- Pivot → riorganizzazione dell'orientamento della struttura, senza variare il livello di dettaglio:

Category	Year	Metrics	Dollar Sales	Customer Region	North-East	Mid-Atlantic	South-East	Central	South	North-West	South-West	England	France	Germany
Electronics	1997		\$ 10.616		\$ 10.616									
	1998		\$ 29.299											
Food	1997		\$ 5.300											
	1998		\$ 5.638											
Gifts	1997		\$ 16.315											
	1998		\$ 20.047											
Health & Beauty	1997		\$ 6.042											
	1998		\$ 5.665											
Household	1997		\$ 38.383											
	1998		\$ 50.391											
Kid's Korner	1997		\$ 2.559											
	1998		\$ 2.943											
Travel	1997		\$ 4.497											
	1998		\$ 4.792											

Category	Year	Metrics	Dollar Sales	Customer Region	North-East	Mid-Atlantic	South-East	Central	South	North-West	South-West	England	France	Germany
					1997	1998	1997	1998	1997	1998	1997	1998	1997	1998
Electronics			\$ 10.616		\$ 10.616	\$ 29.299								
Food			\$ 5.300		\$ 5.300	\$ 5.638								
Gifts			\$ 16.315		\$ 16.315	\$ 20.047								
Health & Beauty			\$ 6.042		\$ 6.042	\$ 5.665								
Household			\$ 38.383		\$ 38.383	\$ 50.391								
Kid's Korner			\$ 2.559		\$ 2.559	\$ 2.943								
Travel			\$ 4.497		\$ 4.497	\$ 4.792								

Esempio (visualizzare per ogni città e mese l'importo delle vendite e la media rispetto al mese corrente e ai 2 precedenti [separatamente per ogni città]):

```
SELECT Città, Mese, Importo,
       AVG(Importo) OVER (PARTITION BY Città
                           ORDER BY Mese
                           ROWS 2 PRECEDING)
AS MediaMobile
FROM Vendite
```



⚠ Nell'esempio, è necessario specificare l'ordinamento, in quanto l'aggregazione richiesta usa le righe in modo ordinato; si nota inoltre che, quando la finestra è incompleta, il calcolo viene fatto sulla parte della finestra presente (si può specificare in tal caso che il risultato deve essere NULL).

La **finestra di aggregazione** può essere definita:

- **A LIVELLO FISICO** = formando il gruppo mediante il conteggio delle righe, è adatto per dati **densi**:
 - **ROWS 2 PRECEDING** (o **FOLLOWING**)
 - **ROWS BETWEEN 3 PRECEDING AND 1 PRECEDING**
 - **ROWS UNBOUNDED PRECEDING**
- **A LIVELLO LOGICO** = formando il gruppo in base alla **definizione di un intervallo intorno alla chiave di ordinamento**, è adatto per dati **sparsi**:
 - **RANGE 2 MONTH PRECEDING**

Esempio 1 [totali cumulativi] (visualizzare per ogni città e mese l'importo delle vendite e l'importo cumulativo delle vendite al trascorrere dei mesi, separatamente per ogni città):

```
SELECT Città, Mese, Importo,
       SUM(Importo) OVER (PARTITION BY Città
                           ORDER BY Mese
                           ROWS UNBOUNDED PRECEDING)
  AS SommaCumul
FROM Vendite
```

Città	Mese	Importo	SommaCumul
Milano	7	110	110
Milano	8	10	120
Milano	9	90	210
Milano	10	80	290
Milano	11	40	330
Milano	12	140	470
Torino	7	70	70
Torino	8	30	100
Torino	9	80	180
Torino	10	100	280
Torino	11	50	330
Torino	12	150	480

Esempio 2 [confronto tra dati dettagliati e dati complessivi] (vedere per ogni città e mese l'importo delle vendite e l'importo totale delle vendite sul periodo completo per la città corrente)

```
SELECT Città, Mese, Importo,
       SUM(Importo) OVER (PARTITION BY Città)
  AS ImpTotale
FROM Vendite
```

Città	Mese	Importo	ImpTotale
Milano	7	110	470
Milano	8	10	470
Milano	9	90	470
Milano	10	80	470
Milano	11	40	470
Milano	12	140	470
Torino	7	70	480
Torino	8	30	480
Torino	9	80	480
Torino	10	100	480
Torino	11	50	480
Torino	12	150	480

Esempio 3 [uguale al 2] (visualizzare per ogni città e mese l'importo delle vendite, il rapporto tra importo della riga corrente per le vendite e il totale complessivo, tra corrente e complessivo per città, e tra corrente e complessivo per mese)

⚠ Nella prima finestra di calcolo è necessario **OVER()** perché non è possibile fare operazioni tra un valore e un aggregato in SQL non esteso!

```
SELECT Città, Mese, Importo
       Importo/SUM(Importo) OVER ()
  AS PercTotale
       Importo/SUM(Importo) OVER (PARTITION BY Città)
  AS PercCittà
       Importo/SUM(Importo) OVER (PARTITION BY Mese)
  AS PercMese
FROM Vendite
```

Città	Mese	Importo	PercTotale	PercCittà	PercMese
Milano	7	110	110/950	110/470	110/180
Milano	8	10	10/950	10/470	10/40
Milano	9	90	90/950	90/470	90/170
Milano	10	80	80/950	80/470	80/180
Milano	11	40	40/950	40/470	40/90
Milano	12	140	140/950	140/470	140/290
Torino	7	70	70/950	70/480	70/180
Torino	8	30	30/950	30/480	30/40
Torino	9	80	80/950	80/480	80/170
Torino	10	100	100/950	100/480	100/180
Torino	11	50	50/950	50/480	50/90
Torino	12	150	150/950	150/480	150/290

⚠ Si può abbinare l'uso di finestre di calcolo con il raggruppamento eseguito dalla **GROUP BY**; la tabella generata diventa l'operando a cui applicare le operazioni definite per la finestra di calcolo.

⚠ **IMPORTANTE:** in una finestra di calcolo si possono usare **solo le colonne disponibili a livello di SELECT**. Esempio:

```
SELECT Città, Mese, SUM(Importo) AS TotMese,
       AVG(SUM(Importo)) OVER (PARTITION BY Città)
  AS MediaMobile
FROM Vendite, ...
WHERE <cond. join> GROUP BY Città, Mese
```

Non si può usare, nella finestra di calcolo, l'attributo importo, ma solo la SUM(Importo); non si può neanche calcolare AVG(Importo) come operazione relativa alla finestra di calcolo, in quanto Importo non è disponibile a livello di SELECT.

FUNZIONI DI RANKING → calcolare la posizione di un valore all'interno di una partizione (si può computare i top "N" record del ranking facendo **WHERE Rank <= N**):

- **RANK()** = calcola la posizione, lascia intervalli vuoti in presenza di record a pari merito [1,1,3]
- **DENSERANK()** = calcola la posizione senza lasciare intervalli vuoti se pari merito [1,1,2]

Esempio [ranking] (vedere per ogni città nel mese di dicembre l'importo delle vendite e la posizione nella graduatoria ordinando per importi superiori):

```
SELECT Città, Importo,
       RANK() OVER (ORDER BY Importo DESC)
  AS Graduatoria
FROM Vendite
WHERE Mese = 12
```

Città	Importo	Graduatoria
Torino	150	1
Milano	140	2

⚠ Si può usare una rank all'interno di un'altra interrogazione al fine di trovare i "top N" elementi, usando poi una WHERE <= N nella select esterna.

```
SELECT * FROM
  (SELECT COD_A, SUM(Q),
  RANK() OVER (ORDER BY SUM(Q))
  AS RankVendite
  FROM FAP
  GROUP BY COD_A)
  WHERE RankVendite<=2;
```

ESTENSIONE DELLA GROUP BY:

- **ROLLUP**: calcolare le aggregazioni su tutti i gruppi ottenuti togliendo 1 colonna per volta [GROUP BY ROLLUP a,b,c → GROUP BY a,b,c; GROUP BY a,b; GROUP BY a; NO GROUP BY]
- **CUBE**: calcolare le aggregazioni su tutte le possibili combinazioni delle colonne specificate [se ho su a,b,c farà: a,b,c | a,b | a,c | b,c | a | b | c | NO]
- **GROUPING SETS**: specificare un elenco di raggruppamenti richiesti [GROUP BY GROUPING SET (a,(a,b,c)) → GROUP BY a; GROUP BY a,b,c]

⚠ Per ciascuna di queste clausole, verranno fuori dei risultati simili alla tabella a dx: quando troviamo un **NULL**, significa che quel record non è aggregato rispetto a quella colonna, ma solo per le altre]

Città	Mese	Pkey	TotVendite
Milano	7	145	110
Milano	7	150	10
Milano
Milano	7	NULL	8500
Milano	8
Milano	NULL	NULL	150000
Torino	150
Torino	...	NULL	2500
Torino	NULL	NULL	135000
...
NULL	NULL	NULL	25005000

ALTRE FUNZIONI UTILI:

- **ROW_NUMBER** = assegna un numero progressivo ad ogni riga all'interno di una partizione. Esempio:
SELECT Tipo, Peso, ROW_NUMBER OVER (PARTITION BY Tipo ORDER BY Peso) AS RowNumberPeso FROM ART;
- **CUME_DIST** = assegna un peso tra 0 e 1 ad ogni riga, in funzione del n° di valori distinti che precedono il valor assunto dal campo usato per fare l'ordinamento nelle partizioni. Esempio:
SELECT Tipo, Peso, CUME_DIST() OVER (PARTITION BY Tipo ORDER BY Peso) AS CumePeso FROM ART;
- **NTILE(n)** = divide ogni partizione in "n" sottogruppi, ognuno con lo stesso numero di dati/record (se possibile); ad ogni sottogruppo viene associato un ID numerico. Esempio:
SELECT Tipo, Peso, NTILE(3) OVER (PARTITION BY Tipo ORDER BY Peso) AS Ntile3Peso FROM ART;

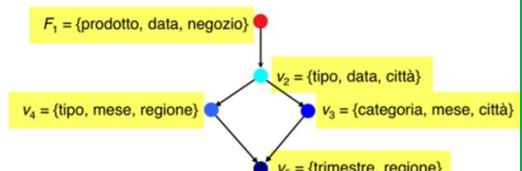
Tipo	Peso	RowNumberPeso	
Barra	12	1	Partizione 1
Ingranaggio	19	1	Partizione 2
Vite	12	1	Partizione 3
Vite	14	2	
Vite	16	3	
Vite	16	4	
Vite	16	5	
Vite	16	6	
Vite	17	7	
Vite	17	8	
Vite	18	9	
Vite	20	10	

Tipo	Peso	CumePeso	
Barra	12	1 (= 1/1)	Partizione 1
Ingranaggio	19	1 (= 1/1)	Partizione 2
Vite	12	.1 (= 1/10)	Partizione 3
Vite	14	.2 (= 2/10)	
Vite	16	.6 (= 6/10)	
Vite	16	.6 (= 6/10)	
Vite	16	.6 (= 6/10)	
Vite	17	.8 (= 8/10)	
Vite	17	.8 (= 8/10)	
Vite	18	.9 (= 9/10)	
Vite	20	1 (= 10/10)	

Tipo	Peso	Ntile3Peso	
Barra	12	1	Partizione 1
Ingranaggio	19	1	Partizione 2
Vite	12	1	Partizione 3
Vite	14	1	
Vite	16	1	Sottogruppo 1
Vite	16	1	
Vite	16	2	Sottogruppo 2
Vite	17	2	
Vite	17	3	
Vite	18	3	Sottogruppo 3
Vite	20	3	

4) VISTE MATERIALIZZATE

Le **VISTE MATERIALIZZATE** sono riassunti precalcolati della tabella dei **fatti**, memorizzati direttamente nella DW, che permettono di velocizzare le interrogazioni che richiedono aggregazioni. Il **reticolo delle viste** rappresenta l'insieme delle viste di cui si ha bisogno, rappresentando le viste con un contenuto più aggregato andando verso il basso.



Vengono **definite da query SQL** a partire dalle tabella sul DW (o da viste con contenuto meno aggregato di quello che si vuole generare); bisogna identificare l'insieme "ottimo" di viste da realizzare rispetto alle query che poi le useranno (cioè le più "utili") partendo da delle **"viste candidate"**. L'**insieme "ottimo"** deve minimizzare spazio & tempo (spazio su disco, tempo di aggiornamento e esecuzione, tempo di risposta e freschezza dei dati) e costi. Possono essere usate in ogni interrogazione come fossero tabelle.

Si può **CREARE UNA VISTA** con:

```
CREATE MATERIALIZED VIEW Name
[BUILD {IMMEDIATE | DEFERRED}]
[REFRESH {COMPLETE | FAST | FORCE | NEVER} {ON COMMIT | ON DEMAND}]
[ENABLE QUERY REWRITE] AS Query;
```

- *Name* = nome della vista
- *Query* = interrogazione associata alla vista

- **BUILD:**
 - **IMMEDIATE** = crea la vista materializzata e carica subito i risultati dell'interrogazione al suo interno
 - **DEFERRED** = crea la vista materializzata senza caricare i dati associati all'interrogazione in essa
- **REFRESH:**
 - **COMPLETE** = ricalcola il risultato dell'interrogazione su tutti i dati
 - **FAST** = aggiorna il contenuto basandosi sulle modifiche avvenute dall'ultimo refresh (usa dei file di log da cui computa quali dati sono da aggiornare)
 - **FORCE** = se possibile (se abbiamo i file di log) viene fatto REFRESH FAST, altrimenti COMPLETE
 - **NEVER** = non si aggiorna automaticamente con Oracle il contenuto della vista
 - **ON COMMIT** = il refresh avviene automaticamente quando le operazioni SQL eseguite modificano il contenuto della vista
 - **ON DEMAND** = il refresh viene fatto solo su richiesta esplicita dell'utente (con il comando DBMS_MVIEW.REFRESH('Name', {'C'|'F'}), dove *Name* è il nome della vista da aggiornare, C = COMPLETE e F = FORCE)
- **ENABLE QUERY REWRITE** = abilita il DBMS ad usare la vista come blocco base per eseguire più velocemente altre interrogazioni

Esempio (Voglio “materializzare” la query SELECT Cod_F, Cod_A, SUM(Q) FROM FAP GROUP BY Cod_F, Cod_A con caricamento dei dati immediato, refresh completo operato solo su richiesta dell’utente e abilitazione alla riscrittura delle query):

```
CREATE MATERIALIZED VIEW Frn_Art_sumQ
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE
AS SELECT Cod_F, Cod_A, SUM(Q) FROM FAP GROUP BY Cod_F, Cod_A;
```

⚠ I file di log (“**MATERIALIZED VIEW LOG**”, richiesti per il **REFRESH FAST**) sono **1 per tabella** e contengono le sue variazioni (o dei suoi attributi). Per CREARE UN MATERIALIZED VIEW LOG associato alla tabella uso:

CREATE MATERIALIZED VIEW LOG ON Tabella

WITH SEQUENCE, ROWID # **SEQUENCE** = istante in cui è avvenuta la modifica; **ROWID** = tupla modificata (*Attributo1, Attributo2, Attributo3*) # Attributi della tabella di cui il log deve tenere traccia

INCLUDING NEW VALUES; # indica la gestione di nuovi valori sugli attributi

⚠ Si può **eliminare** una vista con **DROP MATERIALIZED VIEW Name**; si può **modificare** con **ALTER MATERIALIZED VIEW Name Options**.

Ritornando al discorso di prima, rimane la **PROGETTAZIONE FISICA**, composta da diverse fasi:

- **Analisi del carico di lavoro** = analisi delle query, della loro frequenza di esecuzione e della frequenza di aggiornamento dei dati

- **Definizione delle strutture fisiche “accessorie”** → indici di:

- **Bitmap** = permettono di eseguire operazioni legate ai predicati di selezione più velocemente
- **Join** = permettono di eseguire operazioni di join più velocemente (memorizzando delle tabelle che contengono già i row id da andare a mettere nel join)

- Considerare le **caratteristiche dell’ottimizzatore** = sceglie se e quando usare le viste fatte dall’operatore

- **Procedimento effettivo della progettazione fisica:**

- Scelta delle strutture adatte a supportare le query **più frequenti**
- Scelta delle strutture in grado di contribuire al miglioramento di più interrogazioni **contemporanee**
- Rispetto dei **vincoli** di spazio e frequenza di aggiornamento dei dati

- **Tuning** = verificare se la progettazione fisica soddisfa le necessità applicative (altrimenti riprogetto)

Per indicizzare le **DIMENSIONI**, devo creare un indice per gli attributi coinvolti in predicati di selezione:

- se il dominio ha cardinalità elevata, si usa un indice **B-tree**
- se il dominio ha cardinalità ridotta, si usa un indice **bitmap**

⚠ Per l’indicizzazione delle operazioni di **join**, si usa un **bitmapped join index** (se disponibile); per l’indicizzazione delle operazioni di **group by**, si usano le **viste materializzate**.

Parliamo dell’**ALIMENTAZIONE** del DW: il **PROCESSO ETL** (**Extraction, Transformation & Loading**) è la preparazione dei dati da mettere nel DW, ed è composto da estrazione dei dati dalle sorgenti, pulitura,

trasformazione e caricamento. Questo processo viene eseguito sia durante il **1° popolamento del DW** sia durante l'**aggiornamento periodico dei dati**. Vediamo le fasi:

- **ESTRAZIONE** = acquisizione dei dati dalle sorgenti (devono essere dati di qualità); 2 modalità:
 - **Statica** (fotografia dei dati dalle sorgenti [1° popolamento])
 - **Incrementale** (si considerano solo i dati nuovi o aggiornati [aggiornamento periodico]) = può avvenire mediante:
 - **Applicazione** che cattura le modifiche
 - **Log** che storicizzano le modifiche
 - **Trigger** che catturano le modifiche
 - **Timestamp** che marcano l'ultima modifica di un record (richiede però modifica dello schema della base di dati e può perdere stati intermedi)

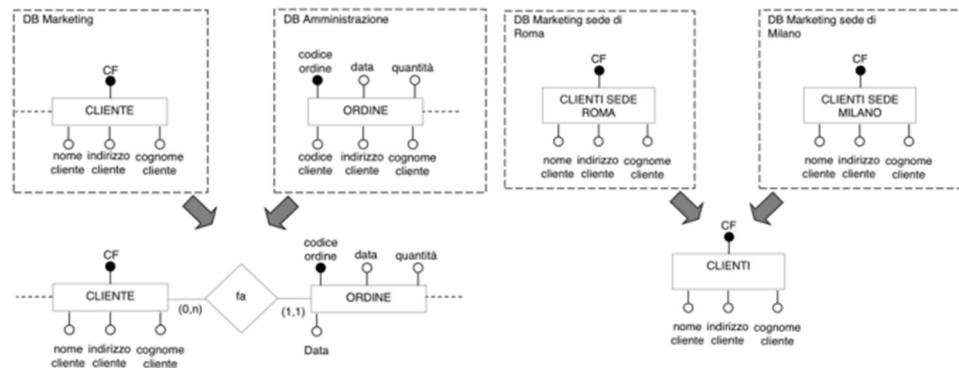
L'estrazione può avvenire su dati operazioni di natura diversa:

- ✓ **Storicizzati** → si salvano tutte le modifiche del dato per un periodo definito di tempo
- ✓ **Semi-storicizzati** → si salvano solo un n° limitato di stati (estrazione incrementale più difficile)
- ✓ **Transitori** → si salva solo lo stato corrente (estrazione molto difficile)

Cod	Prodotto	Cliente	Qta	Azione
3	Barbera	Lumin	75	D
4	Sangiovese	Cappelli	145	U
5	Vermontino	Maltoni	25	I
6	Trebbiano	Maltoni	150	

- **PULITURA** = miglioramento della qualità dei dati, si opera su dati **duplicati, mancanti, uso non previsto di un campo**, valori **impossibili/errati** e **inconsistenza** fra valori logicamente associati. La soluzione migliore è la prevenzione. A seconda del problema si usano tecniche diverse:

- **Tecniche basate su dizionari**: per attributi con un dominio ristretto che presentano errori di battitura o formato
- **Tecniche di fusione approssimata**: per riconoscimento di duplicati/correlazioni tra dati simili:
 - **Join approssimato** = join usando gli attributi comuni non-chiave
 - **Purge/Merge** = dati in 2 tabelle diverse vengono uniti in 1 tabella (necessita l'identificazione e l'eliminazione dei dati duplicati, e la gestione di 2 attributi simili ma in formato diverso)



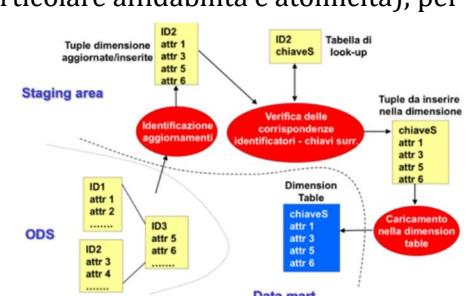
- Identificazione di outliers

- **TRASFORMAZIONE** = dati convertiti dal formato operazionale a quello del DW; richiede **rappresentazione uniforme** del dato che avviene in 2 fasi:

- Sorgenti operazionali → Dati riconciliati:
 - Conversioni e normalizzazione
 - Matching
 - Filtraggio dei dati significativi
- Dati riconciliati → DW: generazione di chiavi surrogate e di valori aggregati

- **CARICAMENTO** = aggiornamenti propagati al DW (richiede ACID, in particolare affidabilità e atomicità); per mantenere l'integrità, le **MODIFICHE** vengono fatte nell'**ordine**:

- **Alimentazione delle TABELLE delle DIMENSIONI:**
 - Identificazione degli aggiornamenti
 - Individuazione della chiave surrogata usando tabella di lookup
 - Aggiornamento dei dati nella tabella delle dimensioni

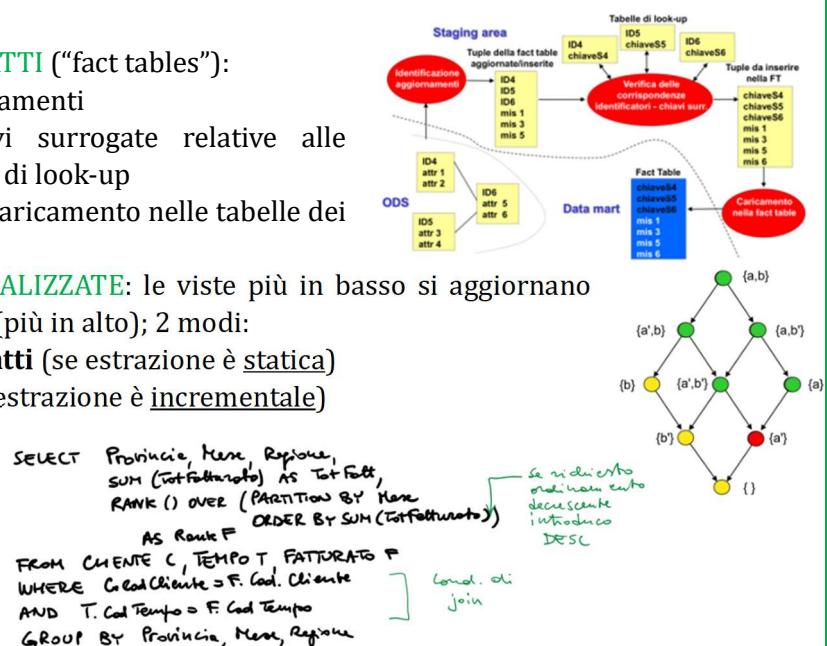


- **Alimentazione delle TABELLE dei FATTI** ("fact tables"):
 - Identificazione degli aggiornamenti
 - Individuazione delle chiavi surrogate relative alle dimensioni mediante tabelle di look-up
 - Inserimento delle misure e caricamento nelle tabelle dei fatti
- **Alimentazione delle VISTE MATERIALIZZATE**: le viste più in basso si aggiornano partendo da quelle meno aggregate (più in alto); 2 modi:
 - Partendo dalla **tabella dei fatti** (se estrazione è statica)
 - Partendo dalle **sorgenti** (se estrazione è incrementale)

Esercizio:

Visualizzare per ogni provincia e mese

- la provincia
- la regione della provincia
- il mese
- il fatturato totale associato alla provincia nel mese in esame
- il rank della provincia in funzione del fatturato totale, separato per mese



5) DATA LAKES

I **DATA LAKES** sono repository di dati di tipo **raw** (grezzi) storicizzati senza alcuna elaborazione, da cui è possibile estrarre valore se appositamente lavorati. Contengono dati **strutturati** (relazionali), **semi-strutturati** (csv, json, xml...), **non-strutturati** (testo) e **binari** (immagini, audio...). In questi sistemi le operazioni di ricerca di un dato sono più complesse e più simili alle ricerche su Google, in cui però bisogna fare **data wrangling** (convertire i dati in un formato fruibile) [non si a priori quale valore e quale utilizzo avranno i dati memorizzati, permettendo di memorizzarli tutti in un unico luogo]. Quindi un data lake ha come **PRO**:

- memorizza tutti i dati (da più sorgenti, in meno tempo)
 - gestisce dati di qualsiasi tipo (si adatta facilmente ai cambiamenti), permettendone l'esplorazione
 - utile ad altri tipi di utenti (e non solo gli attori di business)
 - fornisce informazioni utili più velocemente
 - integrazione fuori dallo storage
- Come **CONTRO**:
- dati memorizzati senza essere sicuri del contenuto
 - consistenza e qualità dei dati non sono certe
 - dati non usabili da business users (ma solo da chi sa gestire questi dati)
 - query ostiche ("rogue") possono rallentare grandi cluster

⚠ Confronto con DW:

Data warehouse	Data lake
Contiene dati relazionali	Può anche contenere dati non-relazionali
Lo schema del DW è definito a priori	Lo schema viene scritto durante l'analisi
Alti costi	Bassi costi
Dati di qualità	Nessuna certezza sulla qualità
Utilizzato dai business analysts	Utilizzato dai data scientist e dai data developers
Analytics: BI e visualizzazione, batch reporting	Analytics: full-text search, machine learning, predictive analytics, data discovery, profiling e utilizzo di algoritmi unsupervised

⚠ Oggi non si memorizzano sempre tutti i dati, ma si usa un approccio più conservativo (non salvare dati inutili); questo per non finire nei "**data swamp**" (paludi di tanti dati inusati).

6) DATA PREPROCESSING

I **dati** sono una collezione di **oggetti** con i loro **attributi**; gli **attributi** sono una proprietà/caratteristica dell'oggetto → un insieme di attributi descrive un **oggetto**. Gli attributi possono essere **Nominali** (ID, colore occhi, CAP), **Ordinali** (numeri interi come in un ranking), **Intervalli** e **Rapporti** (valori floating-point).

Il **tipo** di un attributo dipende da quali delle seguenti proprietà possiede:

- **Distinguibilità** (==, !=) → Tutti
- **Ordine** (<, >) → Ordinale, Intervallo, Rapporto
- **Addizione** (+, -) → Intervallo, Rapporto
- **Moltiplicazione** (*, /) → Rapporto

Un attributo può essere:

- **Discreto** = ha un n° finito oppure infinito-numerabile di valori possibili (di solito variabili intere) [es. binari]
- **Continuo** = valore reale che ha quindi un n° infinito-non-numerabile di valori possibili (di solito floating [FP])

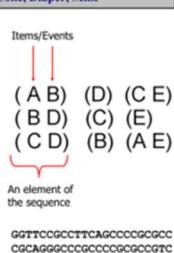
⚠ Identificare il tipo di dato serve a mappare il dato al problema e identificare se serve fare preprocessing.

Un **DATA SET** può essere di diversi tipi:

- **RECORD**:
 - Dati di tipo **tabellare** (numero fisso di attributi, nome colonna definito 1 volta sola [alla creazione della tabella])
 - Dati di tipo **documenti** (dati testuali semi-strutturati/non-strutturati; potremo avere 1 record per ogni parola/frase/paragrafo; anche pagine web con i tag)
 - Dati **transazionali** (ogni record ha ID + insieme di oggetti; ordine non conta)
- **GRAFO**: insieme di nodi e archi con un determinato peso
- **ORDINATO**:
 - **Sequenza di transazioni**: ogni transazione è formata da una sequenza di oggetti, in cui ogni oggetto è separato da parentesi tonde; non ci interessa l'ordine nel singolo oggetto, bensì l'ordine degli oggetti nella sequenza
 - **Sequenza di geni** (qui la sequenza non è legata ad un ordine temporale)
 - **Sequenza spazio-temporale** (es. mappa di temperature mensili in diverse regioni)

TID	Refund	Marital Status	Taxable Income	Cheat
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No

TID	Items
1	Bread, Coke, Milk
2	Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

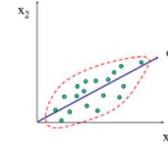


Una **scarsa qualità dei dati** crea più sforzi nella fase di preprocessing; i problemi sui dati possono essere:

- **Rumore** = dati che hanno subito una distorsione
- **Outliers** = oggetti nel dataset con caratteristiche molto diverse dagli altri (da scartare o da ricercare)
- **Valori mancanti** = a causa di informazioni non raccolte o attributi non usabili in tutti i casi; gestione:
 - eliminare gli oggetti con dati mancanti
 - stimare i valori mancanti in base agli altri dati
 - ignorare i valori mancanti
- **Dati duplicati** = possono essere duplicati a causa di una data integration (unione di 2 DB con lo stesso record)
- **Dati errati**

Il **DATA PREPROCESSING** ha diverse **tecniche**:

- **AGGREGAZIONE** → combinazione di 2 o più attributi/oggetti in 1 attributo/oggetto; può avere diversi scopi:
 - **riduzione dei dati** (riduzione del n° di attributi/oggetti)
 - **cambio di scala** (es. città aggregate in regioni, stati...)
 - **dati più "stabili"** (i dati aggregati tendono ad avere meno variabilità)
- **SAMPLING (CAMPIONAMENTO)** → si considerano solo alcuni oggetti del dataset che rappresentino un campione del set (stesse caratteristiche del set) [usato perché lavorare su tutto il dataset è costoso]; 2 tipi:
 - **Casuale** = stessa possibilità di selezionare qualsiasi item:
 - senza rimpiazzo (elemento selezionato viene rimosso dalla popolazione)
 - con rimpiazzo (elemento selezionato rimane nella popolazione e si può rielezionare)
 - **Stratificato** = dati vengono divisi in partizioni, viene selezionata una % di elementi da ogni partizione
- ⚠ “Curse of dimensionality” = all'aumentare delle dimensioni, i dati diventano più sparsi e serve un n° di samples >; perciò serve la riduzione dimensionale ↓
- **RIDUZIONE DIMENSIONALE** → facilita la visualizzazione del dato, non considerando le features meno significative (evita la curse of dimensionality, riduce i tempi e la memoria richiesta per il mining). La tecnica principale è la **PCA** (Principal Component Analysis) il cui obiettivo è trovare



una proiezione degli attributi che prenda la maggiore variazione tra i dati (gli attributi ottenuti non sono uguali e sono una combinazione lineare di quelli originali)

- **SELEZIONE DI UN SUBSET DI FEATURE** → ridurre la dimensionalità del dato, selezionando solo gli attributi già presenti (senza crearne di nuovi), eliminando gli attributi ridondanti e non rilevanti; tecniche:
 - **Brute-force** = vengono provate tutte le possibilità come input dell'algoritmo di data science
 - **Embedded** = selezione avviene nell'algoritmo di data science (utile solo con pochi attributi)
 - **Filter** = selezione avviene prima che l'algoritmo di data science venga eseguito (es. usare approccio di Pearson per calcolare la correlazione fra variabili [se indice > 0.8 = variabili correlate])
 - **Wrapper** = l'algoritmo di data science viene usato come black-box per trovare il subset migliore (opposto del brute-force), confrontando gli output da diverse combinazioni di attributi [a differenza del brute-force, si usa solo un gruppo di possibili combinazioni e non tutte]
- **CREAZIONE DI FEATURES** → creare un nuovo attributo che catturi informazioni utili in maniera più efficiente a partire dagli attributi originali [permette anche di mappare il dato in uno spazio diverso (es. $t \rightarrow f$)]
- **DISCRETIZZAZIONE** → conversione di un attributo continuo in uno ordinale; 2 tipi:
 - **Supervised** = uso etichette di classe per individuare gli insiemi
 - **Unsupervised:**
 - N intervalli della **stessa dimensione** $W = \frac{v_{max} - v_{min}}{N}$ (tecnica facile incrementale [se ho nuovi valori basta aggiungere un bucket], ma limitato se ho outliers e dati sparsi)
 - N intervalli con la **stessa cardinalità** (non incrementale, gestisce meglio outliers e dati sparsi)
 - **Clustering** = gruppi identificati sulla base della distanza (bene con outliers e dati sparsi)
 - **Analisi della distribuzione** = divisione sulla base di quartili/percentili
- **BINARIZZAZIONE** → attributo mappato in variabili binarie:
 - Attributi **continui**: prima mappati in attributo categorico (enum) [altezza → {basso, medio, alto}]
 - Attributi **categorici**: mapping ad un insieme di attributi binari; se nella codifica ho solo 1 bit ad 1 per volta ho “one-hot encoding” [{basso,medio,alto} → {100,010,001}]
- **TRASFORMAZIONE DI UN ATTRIBUTO** → funzione che mappa il set di valori di un attributo in un nuovo set:
 - **NORMALIZZAZIONE** → corregge la distanza tra gli attributi in termini di frequenza, media, varianza...
 - Normalizzazione **min-max**:
$$v' = \frac{v - min_A}{max_A - min_A} (new_{max_A} - new_{min_A}) + new_{min_A} \quad \text{con } v' \in [new_{min_A}, new_{max_A}]$$
 - Normalizzazione **z-score**:
$$v' = \frac{v - \mu_A}{\sigma_A} \quad \text{con } v' \in (-\infty, +\infty)$$
 - Scaling **decimale**:
$$v' = \frac{v}{10^j} \quad \text{con } j = \min t.c \max(|v'|) < 1, \quad v' \in (-1, +1)$$
 - **Standardizzazione** → sottrarre il valore medio e dividere per la deviazione standard (come z-score)

Parliamo ora di **SIMILARITÀ** e **DISSIMILARITÀ** tra 2 oggetti; calcolare in vari modi a seconda dell'attributo [dx].

Attribute Type	Dissimilarity	Similarity
Nominal	$d = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$	$s = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$
Ordinal	$d = x - y / (n - 1)$ (values mapped to integers 0 to $n - 1$, where n is the number of values)	$s = 1 - d$
Interval or Ratio	$d = x - y $	$s = -d, s = \frac{1}{1+d}, s = e^{-d}, s = 1 - \frac{d - min_d}{max_d - min_d}$

In particolare:

- **Distanza Euclidea** → $d(x, y) = \sqrt{\sum_{k=1}^n (x_k - y_k)^2}$ con $n = n^\circ$ attributi, e x_k, y_k i k-esimi attributi degli oggetti x e y [la standardizzazione è necessaria se la scala è diversa]
- **Distanza di Minkowski** → $d(x, y) = (\sum_{k=1}^n |x_k - y_k|^r)^{\frac{1}{r}}$ con $r = \text{parametro}$ che può assumere valori tra 1 (se somma delle differenze degli attributi) e ∞ (distanza suprema, indica la massima differenza fra gli attributi)

Le proprietà comuni delle distanze sono:

- **Positiva:** $d(x, y) \geq 0$ (con = 0 solo se $x = y$)
- **Simmetria:** $d(x, y) = d(y, x)$
- **Disugualanza triangolare:** $d(x, z) \leq d(x, y) + d(y, z)$
- ⚠ Una formula della distanza che soddisfa queste proprietà è detta **metrica**.

Le **similarità** sono strettamente legate alle distanze e le proprietà comuni delle similarità sono **Similarità massima** [$s(x, y) = 1$ solo se $x = y$] e **Simmetria** [$s(x, y) = s(y, x)$].

Vediamo ora altre tecniche:

- **Distanza di Mahalanobis** $\rightarrow d(x, y) = (x - y)^T (\Sigma(x - y))^{-1}$ con Σ = matrice di covarianza; consente di calcolare la distribuzione tra 2 punti rapportandola alla distribuzione di tutti gli altri punti

- **Similarità tra vettori binari:** dati 2 vettori binari p e q (chiamando $M_{ij} = n^o$ di attributi che in p assumono il valore i (0 o 1) e in q assumono il valore j (0 o 1)) definiamo:

- **Simple Matching** $\rightarrow SMC = \frac{n^o \text{ similarità}}{n^o \text{ attributi}}$
- **Jaccard Coefficients** $\rightarrow J = \frac{n^o \text{ di matc } 11}{n^o \text{ di match che non hanno } 00}$

$$p = \begin{matrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{matrix}$$

$$\begin{aligned} M_{01} &= 2 \quad (\text{the number of attributes where } p \text{ was 0 and } q \text{ was 1}) \\ M_{10} &= 1 \quad (\text{the number of attributes where } p \text{ was 1 and } q \text{ was 0}) \\ M_{00} &= 7 \quad (\text{the number of attributes where } p \text{ was 0 and } q \text{ was 0}) \\ M_{11} &= 0 \quad (\text{the number of attributes where } p \text{ was 1 and } q \text{ was 1}) \end{aligned}$$

$$SMC = (M_{11} + M_{00}) / (M_{01} + M_{10} + M_{11} + M_{00}) = (0+7) / (2+1+0+7) = 0.7$$

$$J = (M_{11}) / (M_{01} + M_{10} + M_{11}) = 0 / (2 + 1 + 0) = 0$$

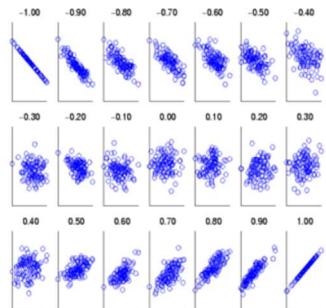
- **Cosine similarity** \rightarrow usata solo nei documenti di testo, quando sono rappresentati come vettori di parole; è il prodotti dei vettori per la loro norma, ovvero $\cos(d_1 d_2) = \frac{d_1 d_2}{\|d_1\| \|d_2\|}$

⚠ Le tecniche di similarità si possono combinare perchè gli attributi di un oggetto sono di tipo diverso; 2 modi:

- Definiamo una metrica di distanza che è combinazione delle metriche usando l'indicatore δ (= 0 se 1 dei 2 oggetti ha un valore mancante all'attributo considerato, altrimenti 1)
- Definiamo un peso ω riguardante l'importanza che vogliamo dare ad un singolo attributo

$$\text{similarity}(\mathbf{x}, \mathbf{y}) = \frac{\sum_{k=1}^n \delta_k s_k(\mathbf{x}, \mathbf{y})}{\sum_{k=1}^n \delta_k}$$

$$\text{similarity}(\mathbf{x}, \mathbf{y}) = \frac{\sum_{k=1}^n \omega_k \delta_k s_k(\mathbf{x}, \mathbf{y})}{\sum_{k=1}^n \omega_k \delta_k}$$



⚠ Ultimo argomento di questa parte è la **Correlazione di Pearson**, che permette di identificare se 2 variabili sono legate da una relazione lineare, quadratica, ... (usa la matrice di correlazione); se c'è un'alta correlazione fra 2 attributi (di solito ≥ 0.8) se ne considera solo 1, dato che l'altro è già rappresentato dal primo (bisogna analizzare bene il segno della correlazione).

7) REGOLE DI ASSOCIAZIONE

L'obiettivo delle **REGOLE DI ASSOCIAZIONE** è estrarre correlazioni frequenti (pattern) da un DB transazionale. Dato un insieme di transazioni fatto da diversi oggetti non ordinati, consideriamo una regola di associazione:

$$A, B \Rightarrow C \quad \text{con} \quad \begin{cases} A, B = \text{corpo della regola} \\ C = \text{testa della regola} \\ \Rightarrow = \text{co - occorenza} \end{cases}$$

È una tecnica **esplorativa** che si può applicare a **qualsiasi tipo di dato** tra cui:

- **Dati testuali** = documento considerato come una transazione, le parole sono gli item (oggetti)
- **Dati strutturati** = riga considerata come una transazione, coppie attributo-valore sono gli item (oggetti)

Si definisce **ITEMSET** un insieme di oggetti/elementi; questo si dice "**frequente**" se ha un supporto $>$ di una soglia. Data una regola di associazione $A \Rightarrow B$ (A e B sono itemset), definiamo 2 metriche di qualità:

- **Supporto** = % di transazioni contenenti sia A sia B , ovvero $Supp(A, B)$
- **Confidenza** = % di transazioni che presentano anche B rispetto alle transazioni che contengono solo A , cioè $\frac{Supp(A, B)}{Supp(A)}$ (rappresenta quindi la "forza" del " \Rightarrow ")

Dato un set di transazioni, l'association rule mining consiste quindi nell'estrarre regole di associazione che rispettano $Supp \geq minSupp$ e $Confidenza \geq minConf$. L'**ESTRAZIONE DELLE REGOLE DI ASSOCIAZIONE** avviene in 2 fasi:

- Estrazione/Generazione degli itemset frequenti** [che spieghiamo nel dettaglio sotto]:
 - A livelli (Apriori)** = vengono estratti prima gli itemset formati da 1 solo elemento, poi 2 etc...
 - Senza generazione di candidati (FP-growth)** = algoritmo ricorsivo che estraе gli itemset cercati
- Estrazione delle regole di associazione** = generazione di tutte le possibili combinazioni binarie di ogni itemset frequente usando una soglia per la confidenza [in questo corso non vediamo questa parte]

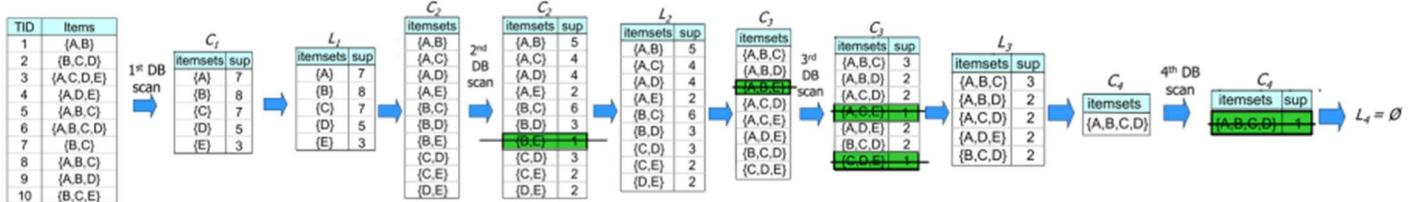
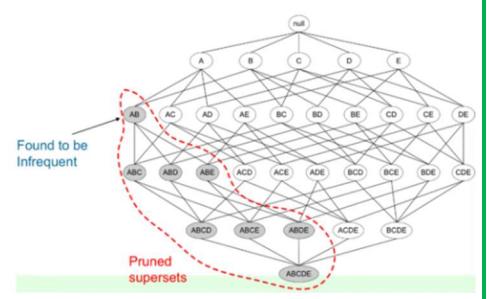
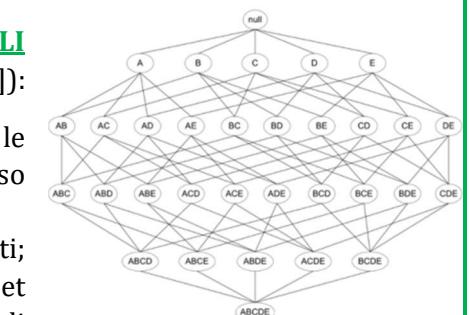
Vediamo nel dettaglio il punto 1) **ESTRAZIONE/GENERAZIONE DEGLI ITEMSET FREQUENTI** (dati N oggetti, 2^N possibili itemset candidati [reticolo]):

- **BRUTE-FORCE** → esame tutto il DB calcolando il Supporto di tutte le possibili combinazioni di itemset. Non è usabile in quanto troppo oneroso (devo quindi ridurre i candidati, le transazioni e/o i confronti)
- **APRIORI** → se un itemset è frequente, tutti i suoi subset sono frequenti; analogamente se è infrequente (in questo modo posso togliere tutti i subset infrequenti). Ha un approccio a livelli: ad ogni iterazione vengono estratti gli itemset di lunghezza k e per ogni livello vengono fatte 2 fasi:

- **Generazione dei candidati:**

- **Join** = vengono generati candidati di lunghezza $k + 1$ unendo gli itemset frequenti di lunghezza k che hanno la parte iniziale lunga $k - 1$ in comune
- **Prune** = eliminiamo i candidati di lunghezza $k + 1$ che hanno almeno 1 k-itemset infrequente

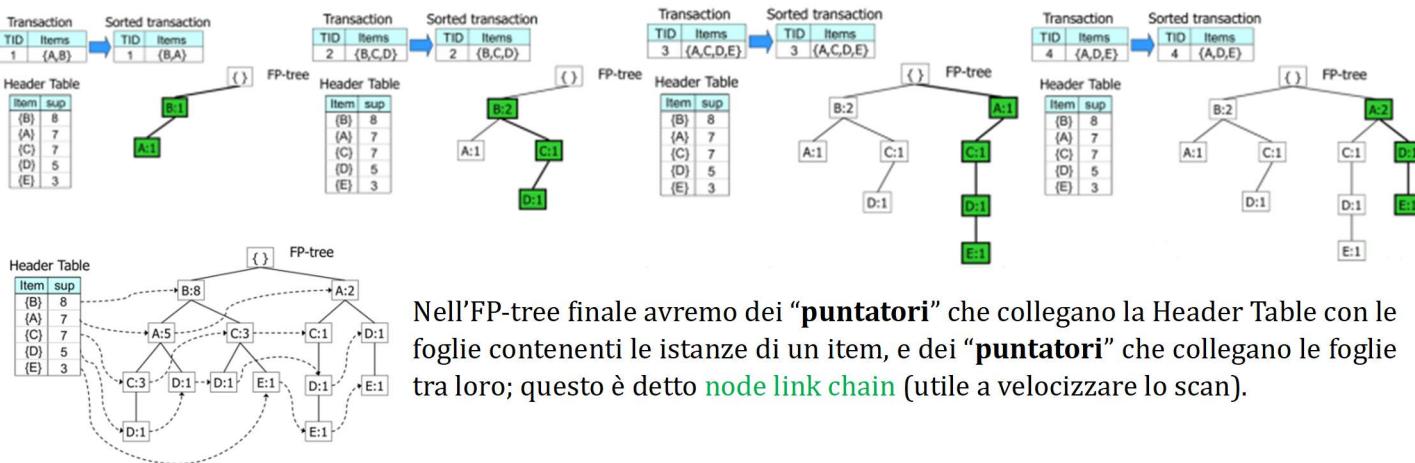
- **Generazione degli itemset frequenti** = viene calcolato il Supporto dei candidati $k + 1$ e eliminiamo quelli sotto $minSupp$



È molto efficiente se voglio individuare solo gli itemset frequenti di una certa lunghezza k . Ciò che limita le performance è il valore **minSupp** (se >, troveremo solo correlazioni ovvie; se <, troveremo troppe correlazioni e quindi alto costo computazionale), il n° di **items**, il n° di **transazioni** nel DB e la **lunghezza media** di ogni transazione. Per limitare il costo computazionale, si usa un “**hash-tree**” che limita i costi di lettura del DB.

- **FP-GROWTH** → sfrutta una rappresentazione compatta del DB salvata su memoria (in modo da non dover fare scansioni successive). L'accesso al DB è fatto solo alla creazione della struttura dati e per il conteggio dei **Supporti**. La **creazione del FP-tree** avviene in 3 fasi:

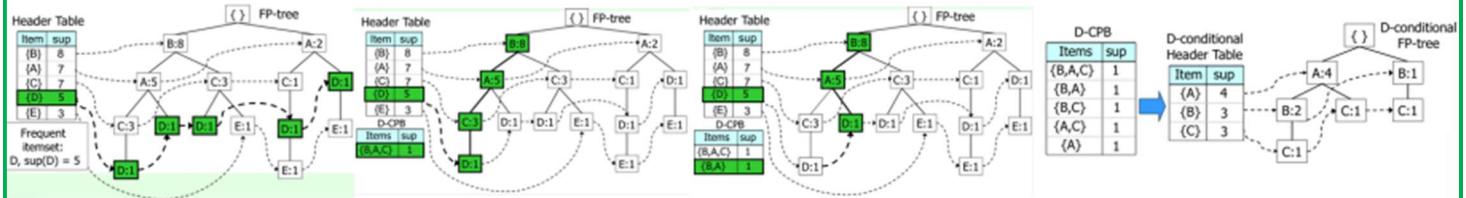
- Conteggio dei **Supporti** ed eliminazione degli item < di $minSupp$
- Costruzione della **Header Table**, ordinando gli oggetti in ordine di Supporto decrescente
- Creazione del **FP-tree**, ovvero per ogni transazione ordiniamo gli oggetti in ordine di Supporto decrescente e li inseriamo all'interno dell'albero, creando un nuovo branch (ramificazione) quando la sequenza di oggetti cambia



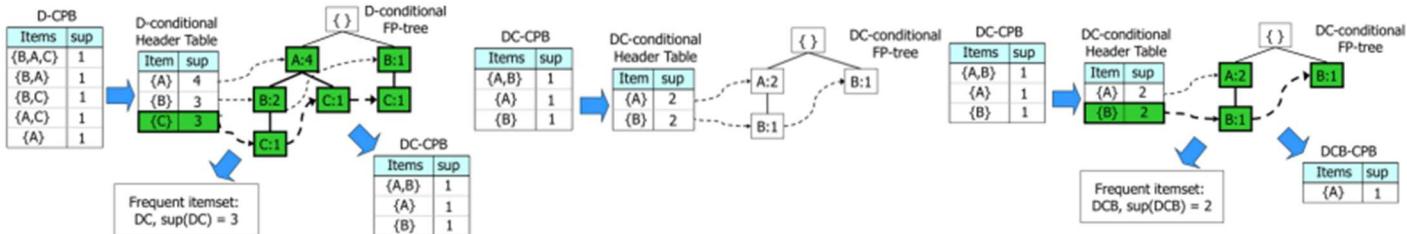
Nell'FP-tree finale avremo dei “**puntatori**” che collegano la Header Table con le foglie contenenti le istanze di un item, e dei “**puntatori**” che collegano le foglie tra loro; questo è detto **node link chain** (utile a velocizzare lo scan).

L'algoritmo esplora la Header Table in ordine di Supporto (dal più basso) per ogni elemento i e i precedenti:

1. Costruiamo il **Conditional Pattern Base** per l'elemento i (ovvero la i -CPB), selezionando i percorsi relativi ai prefissi di i dal FP-tree
2. Invochiamo ricorsivamente FP-growth su i -CPB (sotto un esempio dove costruiamo la D -CPB)



Analogamente, consideriamo i prefissi relativi all'item D, costruendo un FP-tree specifico; poi invochiamo ricorsivamente FP-growth su D-CPB, ottenendo gli alberi DC-CPB e dopo di esso DCB-CPB (riducendo sempre di più la sua dimensione):



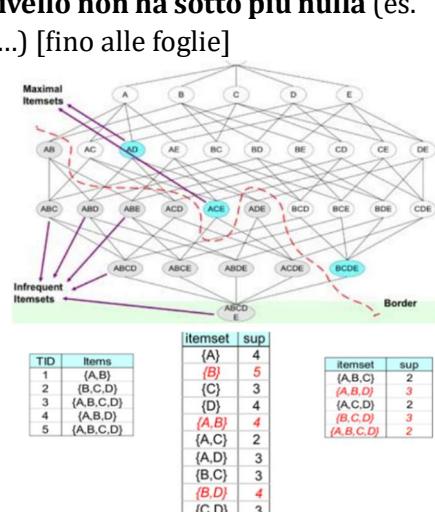
Nell'esempio visto sopra vediamo che A non è frequente in DCB-CPB (ha Supp = 1 < 2, quindi non è frequente perché non ha Supp > 1); quindi torniamo indietro e generiamo l'albero DCA-CPB etc...

Alla fine, dopo varie eliminazioni e dopo averle fatte per ogni item nella Header Table, otteniamo l'insieme di tutti gli itemset frequenti. Il nostro esempio su D-CPB si conclude con la scelta di tutti gli elementi con Supp > 1 contenenti D [img dx]

⚠ Si va avanti ricorsivamente ai "sottoalberi" ($D \rightarrow DB \rightarrow DBA\dots$) fino a che il livello non ha sotto più nulla (es. se DBA contiene solo sé stesso, ci fermiamo torniamo indietro e facciamo DBC ...) [fino alle foglie]

⚠ Come si possono trovare gli itemset frequenti, si possono trovare anche:

- **ITEMSET FREQUENTE MASSIMALE** = se nessuno dei suoi superset direttamente collegati è frequente: **guardando la figura**, non esiste nessun sovrainsieme di AD (ABD, ACD, ADE) frequente [rappresentano gli itemset frequenti di lunghezza massima]
- **ITEMSET FREQUENTE CLOSED** = se nessuno dei suoi superset adiacenti ha lo stesso Supp dell'itemset (massimale \in closed \in frequenti)



La misura della **CONFIDENZA** non è sempre affidabile (es. nei casi in cui la testa della relazione dovrebbe essere negativa per avere significato); per far fronte a ciò, usiamo il concetto di **CORRELAZIONE** (o **LIFT**) associato ad ogni regola estratta [correlazione/lift \neq correlazione di Pearson]; data $r: A \Rightarrow B$, si definisce:

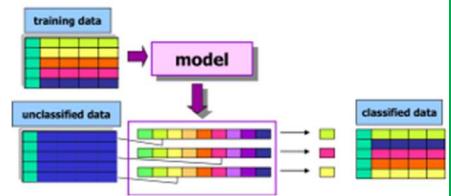
$$Lift = \frac{P(A, B)}{P(A)P(B)} = \frac{Conf(r)}{Supp(B)} \rightarrow \begin{cases} \cong 1 \rightarrow A \text{ e } B \text{ completamente incompatibili} \\ \ll 1 \rightarrow \text{correlazione negativa}, \text{ usata con regola inversa } A \Rightarrow !B \\ > 1 \rightarrow \text{correlazione positiva} \text{ e la regola non deve essere modificata} \end{cases}$$

⚠ In alcuni casi, ogni transizione/oggetto può avere diversa importanza, ovvero diverso **peso** ad esso associato.

⚠ In alcuni casi potrebbe essere utile considerare nelle regole una loro **generalizzazione** (detta **tassonomia**) usando lo stesso concetto delle gerarchie. Possiamo dunque avere regole "cross" che contengono in A un oggetto e in B una generalizzazione (l'estrazione degli itemset frequenti in questo caso ci dà regole di diverse granularità).

8) CLASSIFICAZIONE

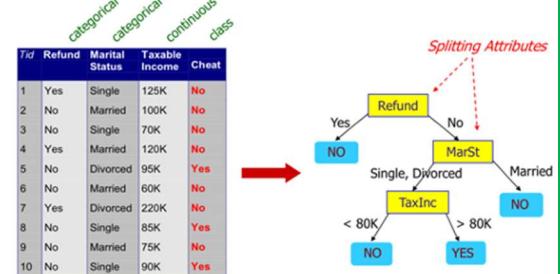
L'obiettivo della classificazione è predire e **assegnare un'etichetta/label di classe** (presa da un insieme predefinito di etichette di classe) usando un modello interpretabile generato sulla base di un **dataset di training** e validato mediante un **dataset di test** (per questo è definito come metodo supervised).



Le **tecniche di classificazione** vengono valutate secondo diversi fattori:

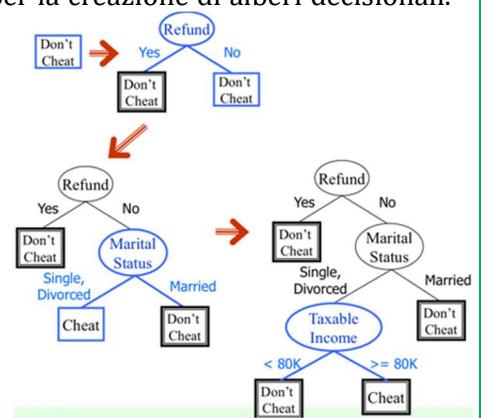
- **Accuratezza** = qualità delle predizioni
- **Interpretabilità** e compattezza del modello
- **Incrementabilità** = il modello si aggiorna dinamicamente in presenza di nuovi dati classificati aggiunti al dataset di training
- **Efficienza** = tempo di costruzione del modello e di classificazione
- **Scalabilità** = performance dell'algoritmo rispetto a dimensione del training set e numero di attributi
- **Robustezza** = capacità dell'algoritmo di operare correttamente anche in presenza di rumore e dati mancanti

Partiamo dalla struttura più facile, ovvero l'**ALBERO DECISIONALE**, il quale permette di determinare l'etichetta di classe di un elemento sfruttando di volta in volta delle **condizionalità** (nei nodi) sui valori degli attributi, scorrendo l'albero fino alle foglie (che rappresentano le **classi**). Dato un dataset, si possono avere diversi alberi decisionali. Nella fase di training si crea l'albero, nella fase di test su applica il modello sui dati di test per capire quanto sia accurato.



L'**algoritmo di Hunt** è un algoritmo (intepretabile ma non incrementale) per la creazione di alberi decisionali. Definendo D_t = insieme di record di training che raggiungono il nodo t :

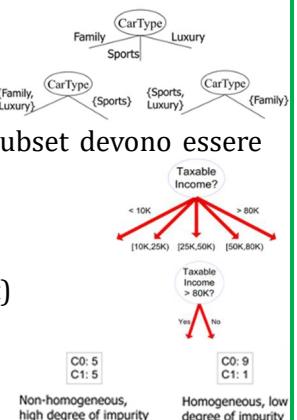
- Se D_t contiene record appartenenti a **più di 1 classe**, selezionamo l'attributo A migliore per lo split e nominiamo il nodo t con A ; poi dividiamo D_t in subset più piccoli e usiamo ricorsivamente questa procedura
- Se D_t contiene record che appartengono ad **1 sola classe** y_t , allora t diventa un nodo foglia etichettato con y_t
- Se D_t è **vuoto** allora t è un nodo foglia etichettato con la classe di maggioranza y_d



Quando ho finito gli attributi da analizzare, ma i record sono ancora eterogenei, assegno la label che ha la maggioranza all'interno dell'insieme.

Questo algoritmo ha certe **tematiche da analizzare**:

- Scelta della struttura della condizione di test → dipende dal tipo di attributo:
 - **Nominale**:
 - Split multiplo = creo un ramo per ogni possibile valore
 - Split binario = divido valori in 2 subset e trovo partizione ottimale
 - **Ordinale** = analogo ai nominali ma nello split binario i valori nello stesso subset devono essere continui
 - **Continuo**:
 - Discretizzazione = genera un attributo categorico ordinale
 - Decisione binaria = $A < v$ oppure $A \geq v$ (bisogna prendere bene lo split)
- Selezione dell'attributo migliore per lo split → sono preferibili attributi con **distribuzione delle classi omogenea**; per far ciò occore misurare la "purezza" dell'attributo e possiamo farlo con diverse metriche:
 - **GINI index** (valori da 0 a 1)
 - **Entropy** (valori da 0 a $\log_2 n_c$)
 - **Information gain** (misura la riduzione dell'entropia data dallo split; predilige split multipli e piccoli)
 - **GainRatio** (aggiustamento dell'information gain per penalizzare split multipli e piccoli)
 - **Misclassification error**



Come si usano queste metriche? Dopo aver calcolato l'impurità prima dello splitting e quella dopo lo splitting relativa ad entrambi gli attributi, confrontiamo la differenza di impurità (**gain**) e scegliamo lo split con gain >. Il GINI index serve per calcolare l'impurità di un nodo figlio e si calcola con:

$$GINI(t) = 1 - \sum_j [p(j|t)]^2 \quad \text{con } p(j|t) = \frac{\text{elementi}_{\text{classe } j}}{\text{elementi}_{\text{totali}}} = \text{frequenza della classe } j \text{ rispetto al nodo } t$$

⚠ Se **GINI index <**, impurità < →

C1	0
C2	6
Gini=0.000	

C1	1
C2	5
Gini=0.278	

C1	2
C2	4
Gini=0.444	

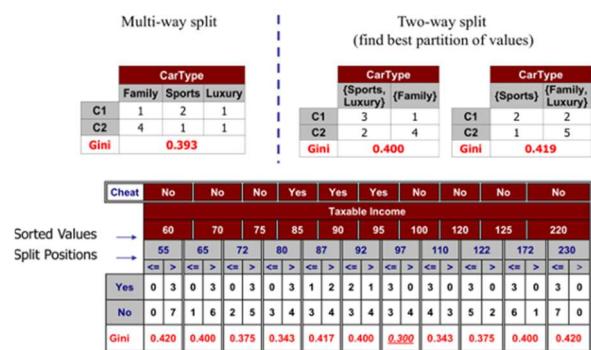
C1	3
C2	3
Gini=0.500	

Se in uno split abbiamo n nodi possibili, usiamo un **GINI pesato** rispetto alla numerosità dei record dei figli:

$$GINI_{\text{split}} = \sum_{i=1}^k \frac{n_i}{n} GINI(i) \quad \text{con } \begin{cases} n_i = \text{n}^{\circ} \text{ record al figlio } i \\ n = \text{n}^{\circ} \text{ record al nodo } p \end{cases}$$

Vediamo questo caso per **attributi**:

- **booleani** = calcolo avviene normalmente per i 2 nodi figli
- **categorici** = per ogni valore distinto contiamo i valori di ciascuna classe e decidiamo se fare uno split multiplo o binario in base alla matrice
- **continui** = se la discretizzazione non è ancora avvenuta, l'algoritmo valuta qual è la condizione migliore confrontando i GINI index relativi a tutti i possibili valori per tutte le possibili condizioni, scegliendo lo split con GINI index <



Altra cosa sono i **CRITERI DI STOP** nella creazione di foglie all'interno dell'albero:

- Fermiamo l'espansione di un nodo quando tutti i suoi record appartengono alla **stessa classe**
- Fermiamo l'espansione di un nodo quando tutti i suoi record hanno valori di **attributi simili**
- Fermiamo **anticipatamente**:
 - **Pre-pruning** = nella creazione dell'albero noto c'è un n° abbastanza alto di nodi e mi fermo
 - **Post-pruning** = una volta creato l'albero completo decido quali sezioni/nodi togliere

⚠ All'aumentare dei nodi, il modello diventa sempre più accurato, ma fino ad una certa soglia; superata questa soglia si parla di **OVERFITTING**. Se invece i nodi sono troppo pochi parliamo di **UNDERFITTING**. In alcuni casi, l'overfitting dipende dalla presenza di rumore: in questi casi conviene avere un modello meno preciso che vada ad identificare semplicemente la diagonale.

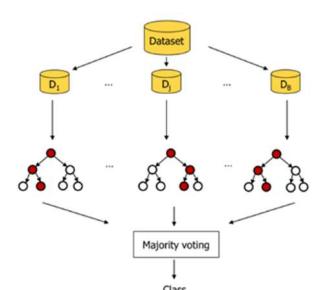
La presenza di **valori mancanti** può influenzare la costruzione dell'albero decisionale in diversi modi:

- influenza il calcolo dell'impurità
- non si sa come indirizzare il record su un certo ramo se lo specifico attributo non è indicato
- non si sa come classificare un'istanza di test con valori mancanti

Valutazione:

- ✓ **Accuratezza** = per dataset semplici, comparabile con altre tecniche di classificazione
- ✓ **Interpretabilità** = è interpretabile quando l'albero non è troppo grande e le predizioni sono interpretabili
- ✓ **Incrementabilità** = NO
- ✓ **Efficienza** = costruzione del modello e classificazione sono veloci
- ✓ **Scalabilità** = SI
- ✓ **Robustezza** = robusto, ma la gestione dei dati mancanti è critica

Un'altra struttura è la **RANDOM FOREST**: prevede la **generazione di diversi alberi decisionali** che lavorano insieme, migliorando l'accuratezza e evitando l'overfitting. Nel training, vengono generati diversi alberi che classificheranno autonomamente i record; la classe effettiva viene assegnata per **majority voting**. Il dataset viene diviso in **random subset** in cui record sono estratti con rimpiazzamento, per cui viene selezionato un



subset di attributi random [se $p = n^o$ di attributi, seleziono \sqrt{p} attributi] e viene generato per ognuno dei subset un albero; gli alberi ottenuti sono decorrelati.

Valutazione:

- ✓ **Accuratezza** = maggiore rispetto agli alberi decisionali
- ✓ **Interpretabilità** = diminuisce all'aumentare del n^o di alberi, ma permette di capire l'importanza delle features
- ✓ **Incrementabilità** = NO
- ✓ **Efficienza** = più veloce degli alberi decisionali perché, una volta scelti i subset, la costruzione e la classificazione degli alberi vengono eseguite contemporaneamente
- ✓ **Scalabilità** = SI (perché gli alberi vengono creati su una porzione di dati)
- ✓ **Robustezza** = robusto contro rumore e outliers e lavora meglio nel caso di missing values

Altro modello è la **RULE-BASED CLASSIFICATION** (basata sulle regole di classificazione). Il modello di predizione dell'etichetta di classe è composta da un insieme di regole nella forma (**Condition**) $\rightarrow y$ dove la *Condition* è un insieme di predicati, mentre y è l'etichetta di classe assegnata. Una regola “**cobre**” un record se gli attributi del record soddisfano le condizioni della regola.

Nella **CONDIZIONE IDEALE** abbiamo regole:

- **Mutualmente esclusive** = 2 regole non possono essere vere contemporaneamente e ogni record deve essere coperto al max da 1 regola
- **Esaustive** = le regole devono considerare tutte le possibili combinazioni di valori di attributi e ogni record deve essere coperto almeno da 1 regola

Ci sono degli **algoritmi** che **estraggono regole da un albero decisionale**: ogni path dalla radice ad una foglia rappresenta una regola e la condizione ideale viene rispettata (a meno che l'albero non abbia subito pruning). Ci sono 2 metodi per estrarre le regole di classificazione:

- **Diretto** = regole estratte dai dati (dal dataset)
- **Indiretto** = regole estratte dai modelli di classificazione pre-esistenti

⚠ Le regole si possono **semplificare** in 3 modi: semplificando le regole estratte dall'albero completo, estraendo le regole da un albero “prunato”, estraendo le regole direttamente dal training set.

Questo però porta la perdita della condizione ideale (mutua esclusività + esaustività):

- se un record soddisfa più di una regola (NO mutua esclusività), bisogna scegliere
- se un record non soddisfa nessuna regola (NO esaustività), gli viene data una classe di default

Valutazione:

- ✓ **Accuratezza** = maggiore ai precedenti se le regole vengono estratte in modo diretto
- ✓ **Interpretabilità** = SI
- ✓ **Incrementabilità** = NO
- ✓ **Efficienza** = SI (veloce sia costruzione sia classificazione)
- ✓ **Scalabilità** = SI (sia cardinalità del training set sia il n^o di attributi)
- ✓ **Robustezza** = robusto rispetto agli outliers

Un'estensione del rule-based è la **CLASSIFICAZIONE ASSOCiativa**, che però considera **solo regole associative** del tipo (**Condition**) $\rightarrow y$. Quindi il formato è lo stesso del rule-based, ma cambia come le regole vengono estratte (estrazione di regole con itemset frequenti): durante la generazione del modello, le regole devono essere **selezionate e ordinate** rispetto a *Supp*, *Conf*, correlazione e thresholds; poi, per evitare overfitting, si tengono solo le regole più in alto nell'ordinamento (le altre “prunate”).

Valutazione:

- ✓ **Accuratezza** = maggiore ai precedenti perché viene considerata la correlazione tra attributi
- ✓ **Interpretabilità** = SI
- ✓ **Incrementabilità** = NO
- ✓ **Efficienza** = creazione lenta (estrazione degli itemset frequenti è onerosa), classificazione veloce
- ✓ **Scalabilità** = SI rispetto alla cardinalità del training set, meno rispetto al n^o di attributi
- ✓ **Robustezza** = robusto rispetto agli outliers, non è affatto da problemi sui dati mancanti

Altro modello è il **K-NEAREST NEIGHBOR**: usa il dataset di training per predire l'etichetta di classe senza la necessità di un modello predittivo. Quando un nuovo oggetto deve essere classificato, si identificano i k oggetti del training set considerati "vicini" secondo una metrica di **distanza** e assegnamo la classe che identifica il maggior n° di vicini [si può anche usare una metrica pesata]. La **scelta del n° di vicini k** (che si può fare con tentativi successivi) è importante in quanto:

- se è troppo piccolo → alta sensibilità al rumore
- se è troppo grande → probabile che il "vicinato" includa anche punti appartenenti ad altre classi

Valutazione:

- ✓ **Accuratezza** = come le altre tecniche
- ✓ **Interpretabilità** = NO, ma le singole predizioni possono essere descritte dai vicini
- ✓ **Incrementabilità** = SI
- ✓ **Efficienza** = la fase di training non c'è in quanto non viene creato un modello, mentre la classificazione è lenta perché bisogna calcolare le distanze per ogni punto/oggetto
- ✓ **Scalabilità** = NO (bisogna calcolare le distanze per ogni record + problema di curse of dimensionality)
- ✓ **Robustezza** = dipende dal metodo usato per calcolare la distanza

Altra tecnica è la **CLASSIFICAZIONE BAYESIANA** (basata sul **TEOREMA DI BAYES** per individuare l'etichetta): consideriamo tutti gli attributi come variabili casuali $x_1, \dots x_k$ con il record rappresentato da $X = \langle x_1, \dots x_k \rangle$, mentre C è l'etichetta di classe. La classificazione Bayesiana calcola $P(C|X) = \frac{P(X|C)*P(C)}{P(X)}$, ovvero la probabilità di assegnare al record X la classe C calcolata sfruttando Bayes: la classe assegnata a X è quella per cui $P(C|X)$ è max. $P(X) = \frac{1}{N} = \text{costante}$ (perché non abbiamo record duplicati), mentre $P(C) = \frac{N_C}{N}$ con N_C = n° di record della classe C , e N = n° di record totali.

Il calcolo di $P(X|C)$ viene fatto con l'ipotesi di Naive (indipendenza statistica degli attributi), ovvero $P(X|C) = P(x_1|C) * P(x_2|C) \dots * P(x_k|C)$; il singolo $P(x_n|C)$ viene calcolato:

- per funzioni **discrete** → $P(x_n|C) = \frac{|x_{k_C}|}{N_C}$
- per funzioni **continue** → distribuzione di probabilità

Outlook	Temperature	Humidity	Windy	Class
sunny	hot	high	false	N
sunny	hot	high	true	N
overcast	hot	high	false	P
rain	mild	high	false	P
rain	cool	normal	false	P
rain	cool	normal	true	N
overcast	cool	normal	true	P
sunny	mild	high	false	N
sunny	cool	normal	false	P
rain	mild	normal	false	P
sunny	mild	normal	true	P
overcast	mild	high	true	P
overcast	hot	normal	false	P
rain	mild	high	true	N

outlook	
$P(\text{sunny} p) = 2/9$	$P(\text{sunny} n) = 3/5$
$P(\text{overcast} p) = 4/9$	$P(\text{overcast} n) = 0$
$P(\text{rain} p) = 3/9$	$P(\text{rain} n) = 2/5$
temperature	
$P(\text{hot} p) = 2/9$	$P(\text{hot} n) = 2/5$
$P(\text{mild} p) = 4/9$	$P(\text{mild} n) = 2/5$
$P(\text{cool} p) = 3/9$	$P(\text{cool} n) = 1/5$
humidity	
$P(\text{high} p) = 3/9$	$P(\text{high} n) = 4/5$
$P(\text{normal} p) = 6/9$	$P(\text{normal} n) = 2/5$
windy	
$P(\text{true} p) = 3/9$	$P(\text{true} n) = 3/5$
$P(\text{false} p) = 6/9$	$P(\text{false} n) = 2/5$

$$\begin{aligned} P(p) &= 9/14 \\ P(n) &= 5/14 \end{aligned}$$

Data to be labeled

$X = \langle \text{rain}, \text{hot}, \text{high}, \text{false} \rangle$

For class p

$$\begin{aligned} P(X|p) \cdot P(p) &= \\ &= P(\text{rain}|p) \cdot P(\text{hot}|p) \cdot P(\text{high}|p) \cdot P(\text{false}|p) \cdot P(p) \\ &= 3/9 \cdot 2/9 \cdot 3/9 \cdot 6/9 \cdot 9/14 = 0.010582 \end{aligned}$$

For class n

$$\begin{aligned} P(X|n) \cdot P(n) &= \\ &= P(\text{rain}|n) \cdot P(\text{hot}|n) \cdot P(\text{high}|n) \cdot P(\text{false}|n) \cdot P(n) \\ &= 2/5 \cdot 2/5 \cdot 4/5 \cdot 2/5 \cdot 5/14 = 0.018286 \end{aligned}$$

Valutazione:

- ✓ **Accuratezza** = simile o più bassa (l'ipotesi di Naive semplifica il modello)
- ✓ **Interpretabilità** = NO, ma i pesi dei contributi in una singola predizione possono essere usati per spiegare
- ✓ **Incrementabilità** = SI (e non richiede l'accesso al training set)
- ✓ **Efficienza** = VELOCE (sia costruzione sia classificazione)
- ✓ **Scalabilità** = SI (sia dimensione training set sia n° attributi)
- ✓ **Robustezza** = dipende dalla correlazione fra gli attributi (da quanto l'ipotesi di Naive sia rispettata)

Un altro modello è il **SVM (Support Vector Machines)**: identifica un superpiano che separa le classi di interesse per creare un modello che distingua i record appartenenti a classi diverse; ci sono diversi possibili iperpiani, ma la soluzione migliore viene scelta in quello che ha la dimensione dei margini maggiore.

Valutazione:

- ✓ **Accuratezza** = ALTA
- ✓ **Interpretabilità** = NO
- ✓ **Incrementabilità** = NO

- ✓ **Efficienza** = la costruzione del modello richiede tuning dei parametri, la classificazione è invece molto veloce
- ✓ **Scalabilità** = mediamente scalabile
- ✓ **Robustezza** = SI (rispetto a rumore e outliers)

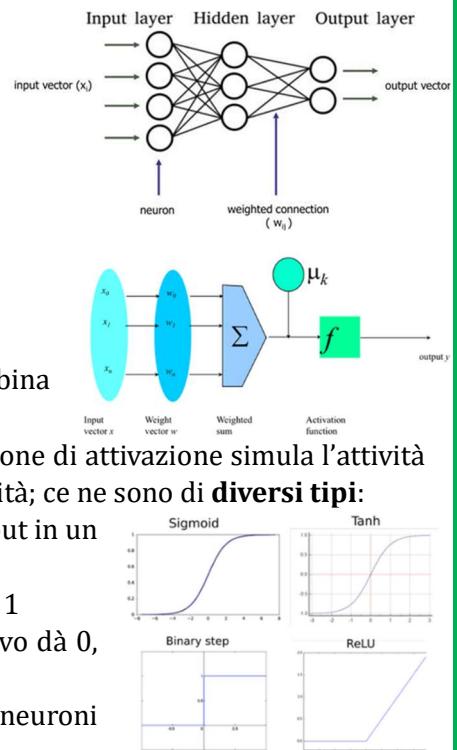
Ultimo modello che vediamo (che è uno dei più importanti) è **ARTIFICIAL NEURAL NETWORK** (come il cervello umano, dove i neuroni sono le unità di elaborazione e le sinapsi sono le connessioni). Diversi **tipi** di reti neurali:

- **Feed Forward Neural Network (FFNN)** → **diversi layer (livelli)** di neuroni, in cui l'output di un livello è l'input del livello successivo (elaborazioni successive). Il dataset di training viene messo nella rete neurale con il livello di input, mentre il livello di output dà l'etichetta di classe. È un modello fully connected (l'output di ogni neurone del livello precedente viene dato in input a tutti i neuroni del livello successivo).

Alle connessioni dei diversi nodi è associato un **vettore peso**.

L'elaborazione effettuata da ogni neurone è:

- riceve in input il **vettore valori** e il **vettore pesi**
- fa la sintesi degli input e dei pesi con una **somma pesata** e la combina col suo offset
- applica una **funzione di attivazione** che genera l'output. La funzione di attivazione simula l'attività biologica dei neuroni ad uno stimolo in input e fornisce non linearità; ce ne sono di **diversi tipi**:
 - **Sigmoid, tanh**: fanno lo scaling dell'input fornendo un output in un output in un range [0,1] o [-1,1]
 - **Binary step**: se il valore in input è negativo dà 0, altrimenti 1
 - **ReLU (Rectified Linear Unit)**: se il valore in input è negativo dà 0, altrimenti dà un valore proporzionale [non fa scaling]
 - **Softmax**: usata solo per il livello di output e considera tutti i neuroni del livello



L'algoritmo per **costruire una FFNN** è:

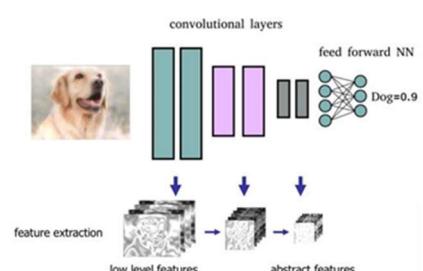
1. vengono inizialmente assegnati valori random per i pesi e gli offset
2. le istanze nel training set vengono processate 1 alla volta
 - a. ogni neurone fa la sua elaborazione
 - b. il risultato viene propagato fino a che non viene calcolato l'output
 - c. viene calcolato l'errore comparando output calcolato e atteso
 - d. l'errore viene propagato all'indietro aggiornando i pesi e l'offset per ogni neurone
3. il processo finisce in 3 casi: l'accuratezza supera un threshold, l'errore è minore di un threshold o viene raggiunto il n° max prestabilito di epoch

Valutazione:

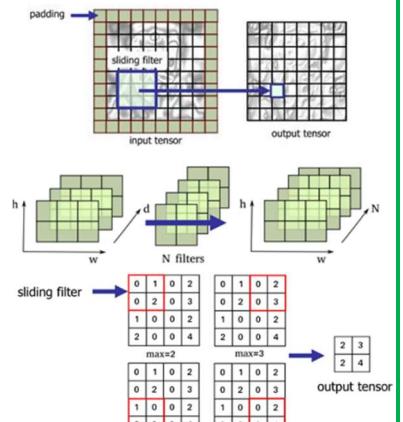
- ✓ **Accuratezza** = MOLTO ALTA
- ✓ **Interpretabilità** = NO
- ✓ **Incrementabilità** = NO
- ✓ **Efficienza** = costruzione del modello richiede tuning dei parametri, classificazione è invece molto veloce
- ✓ **Scalabilità** = mediamente scalabile
- ✓ **Robustezza** = SI (rispetto a rumore e outliers), ma richiede un training set molto grande

- **Convolutional Neural Network (CNN)** → introduce dei **convolutional layer** (livelli) per modellare i dati in input usando **feature extraction**; l'output viene dato in pasto alla FFNN. I vari livelli estraggono features con caratteristiche diverse (i primi estraggono feature specifiche del dato, i successivi estraggono feature per sintetizzare il dato). I dati che passano in una CNN si basano sul concetto di **tensore** (vettore multidimensionale con rank = n° dimensioni, e shape = n° di elementi per ogni dimensione) [un'immagine può essere rappresentata come un tensore con 3 matrici per i colori RGB e 1 per la scala di grigi].

L'elaborazione dei singoli livelli convoluzionali segue **3 fasi**:

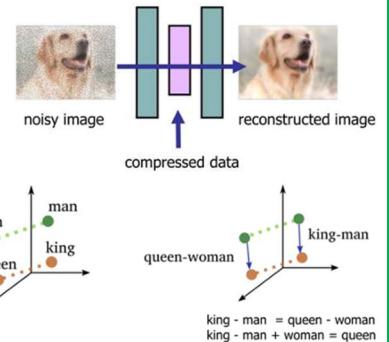


- **Convoluzione** → input = input iniziale oppure tensore prodotto dal livello precedente; output = tensore con features estratte. Viene fatta con n filtri e produce 1 pixel (per le immagini) ottenuto elaborando il pixel prima in quella posizione, correlandolo ai pixel adiacenti
- **Attivazione** → viene applicata una funzione di attivazione al tensore in input (come nella FFNN). Si usa una **ReLU** (training più veloce)
- **Pooling** → fa il downsampling del tensore mediante un filtro scorrevole che rimpiazza i valori del tensore con una sintesi statistica degli output vicini (di solito usato maxpool che considera il valore massimo)
- ⚠ Nel training, ogni filtro riconosce specifici pattern nel tensore in input



- **Recurrent Neural Network (RNN)** → consente l'elaborazione di dati che considerano il concetto di **sequenza** (es. tempo) [usato per traduzioni, predizione di serie temporali, trascrizione vocale, analisi grammaticale]. L'input è un vettore $x(t)$ nello stato dello step precedente $s(t - 1)$; il training viene fatto con **Backpropagation Through Time** (dato $x(t)$ e $y(t)$ [output atteso], l'errore viene propagato all'indietro). Ha però il problema del vanishing gradient (decresce velocemente e i pesi non vengono aggiornati bene, rendendo difficile usare memorie a lungo termine), risolto con LSTM (Long Short Term Memories).

- **Autoencoder** → consente la compressione dei dati in input e la successiva ricostruzione di questi; ha vari usi:
 - **Feature extraction** = la rappresentazione compressa può essere usata come un set di features utili per la rappresentazione dei dati in input
 - **Denoising**
- **Word Embeddings (Word2Vec)** → le parole vengono rappresentate da vettori numerici che catturano l'informazione **semantica** e fanno in modo che parole con significati simili abbiano caratteristiche simili (la distanza semantica tra parole viene calcolata con la distanza tra i vettori)

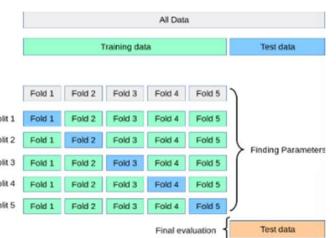


La **valutazione delle performance** di un modello può dipendere da diversi fattori (distribuzione dei record nelle classi, impatto sul modello di una classificazione errata, dimensione del training e del test set) oltre che dall'algoritmo.

La **curva di apprendimento** mostra invece come l'accuratezza cambia al variare della dimensione del training sample e mostra come un sample piccolo comporti una bassa accuratezza e un'alta deviazione dell'errore.

Ci sono 3 fasi per un algoritmo di neural network:

- **TRAINING** → riguardo alla **CREAZIONE DEL TRAINING SET e DEL TEST SET**, esistono diversi modi per il partizionamento dei dati in training e test set:
 - **Sampling** con o senza rimpiazzamento
 - **Holdout** = partizionamento fisso (**utile con DB grossi**)
 - **Cross validation (k-fold)** = dati partizionati in k subset disgiunti; il training viene fatto su $k-1$ subset e il test sul subset rimanente; ciò viene ripetuto per tutte le k combinazioni possibili (quindi **molto accurato, ma non va bene per DB grossi**)
- ⚠ Caso particolare è il **leave-one-out** ($k=n$), in cui il test set è composto da 1 solo elemento (appropriato solo per dataset molto piccoli)



- **VALIDAZIONE** → parlando di **VALIDAZIONE DEI MODELLI** (usata per stimare l'accuratezza di un algoritmo), a questa fase viene riservata una parte del training set. Viene fatta un'analisi di **sensitività** ai parametri propri dell'algoritmo e una **comparativa** tra i risultati dei diversi algoritmi. Le metriche per la valutazione delle performance sfruttano la **matrice di confusione** (rappresenta il n° di predizioni effettuate rispetto alle classi effettive). La metrica più usata è l'**accuratezza**:

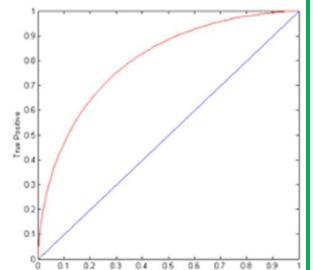
ACTUAL CLASS	PREDICTED CLASS		a: TP (true positive) b: FN (false negative) c: FP (false positive) d: TN (true negative)
	Class=Yes	Class>No	
Class=Yes	a (TP)	b (FN)	
Class>No	c (FP)	d (TN)	

$$accuracy = \frac{n^{\circ} \text{ oggetti classificati bene}}{n^{\circ} \text{ oggetti classificati}}$$

⚠ L'accuratezza risulta **inaffidabile** quando abbiamo dati con classi sbilanciate (es. se ho un dataset con 99% dei dati di classe A e 1% di classe B, avremo che un modello che classifica tutti i record come appartenenti alla classe A ha un'accuratezza del 99%)

Possibili **metriche** per una valutazione all'interno della classe sono:

- $Recall(r) = \frac{n^o \text{ oggetti correttamente assegnati alla classe } C}{n^o \text{ oggetti appartenenti alla classe } C} = \frac{TP}{TP+FN}$
 - $Precision(p) = \frac{n^o \text{ oggetti correttamente assegnati alla classe } C}{n^o \text{ oggetti assegnati alla classe } C} = \frac{TP}{TP+FP}$
 - $F1_{score} = 2 * \frac{r*p}{r+p}$ (per aumentare entrambi r e p , bisogna massimizzare $F1$)



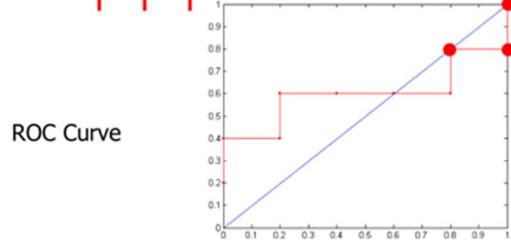
Una tecnica per la comparazione dei modelli è la **Curva ROC** che plotta in un grafico il **True Positive Rate ($TPR = \frac{TP}{TP+F}$)** = *Recall*) e il **False Positive Rate ($FPR = \frac{FP}{FP+TN}$)**.

Il punto $(0,1)$ rappresenta la situazione ideale. Nella figura la diagonale blu rappresenta un modello che predice l'etichetta di classe randomicamente; la curva con area > rappresenta il modello che per forma meglio.

La curva viene costruita considerando la **probabilità** (generata a posteriori dal modello) $P(+|A)$ [ovvero che il record appartenga alla classe “+” positiva] per ogni record e ordinandoli decrescentemente rispetto a ciò; per ogni record viene plottato un **punto (FPR, TPR)** e l’insieme dei punti genera la curva.

⚠ Non esiste un modello che è “generalmente” meglio degli altri; dipende dai miei obiettivi (es. voglio evitare falsi positivi? FPR <; voglio evitare falsi negativi? FPR >)

Class	+	-	+	-	-	-	+	-	+	+	
P(+ A)	0.25	0.43	0.53	0.76	0.85	0.85	0.85	0.87	0.93	0.95	1.00
TP	5	4	4	3	3	3	3	2	2	1	0
FP	5	5	4	4	3	2	1	1	0	0	0
TN	0	0	1	1	2	3	4	4	5	5	5
FN	0	1	1	2	2	2	2	3	3	4	5
TPR	1	0.8	0.8	0.6	0.6	0.6	0.6	0.4	0.4	0.2	0
FPR	1	1	0.8	0.8	0.6	0.4	0.2	0.2	0	0	0



9) CLUSTERING

Il **CLUSTERING** identifica gruppi di oggetti in modo che gli oggetti siano simili tra loro e diversi dagli oggetti negli altri gruppi (permette di capire i dati e sintetizzare i dataset grandi). **CLUSTERING** = insieme di cluster; 2 tipi:

- **Partizionale** = oggetti vengono divisi in subset non sovrapposti in modo che ogni oggetto sia in 1 solo cluster
 - **Gerarchico** = oggetti vengono divisi in cluster nidificati organizzati in un albero gerarchico

Altre caratteristiche di un cluster sono:

- **Esclusivo** (o non esclusivo) = 1 punto non può appartenere a diversi cluster
 - **Fuzzy** (o non fuzzy) = ogni punto appartiene a tutti i cluster con un peso compreso tra 0 e 1 per ogni cluster
 - **Parziale** (o completo) = algoritmo parziale se clusterizza solo una parte di dati (considerando il resto rumore)
 - **Eterogeneo** (o omogeneo) = cluster creati hanno diverse o uguali dimensioni, forme e densità

I cluster possono essere:

- **Well separated** → un cluster è un insieme di punti tale che ognuno dei punti è più vicino a tutti gli altri punti del cluster rispetto a tutti i punti fuori dal cluster
 - **Center-based** → un cluster è un insieme di oggetti tali che un oggetto in un cluster è più vicino al centro del proprio cluster, piuttosto che al centro di un qualsiasi altro cluster (il centro è un **centroide** [media aritmetica degli oggetti nel cluster] oppure un **medoide** [oggetto del dataset più vicino al centroide])
 - **Contiguity-based** → un cluster è un insieme di punti tali che un punto in un cluster sia più vicino ad uno o più punti nel cluster rispetto a qualsiasi punto fuori dal cluster
 - **Density-based** → un cluster è una regione densa di punti, separati da zone a bassa densità (usato con cluster irregolari e in presenza di rumore e outliers)
 - **Concettuali** (o proprietà condivise) → cluster con proprietà comuni (o che rappresentano un concetto)

Parliamo di **ALGORITMI DI CLUSTERING**:

- **K-means** → algoritmo **partizionale** che associa ognuno dei k cluster ad un centroide (o medoide), **assegnando ogni punto al cluster con centroide più vicino**.

Dopo la selezione iniziale (randomica) dei k centroidi, vengono creati i k cluster (assegnando i punti al centroide più vicino) e viene ricalcolata la posizione di tutti i centroidi rispetto ai punti dei vari cluster; ciò viene ripetuto fino a che i centroidi smettano di spostarsi.

Un **posizionamento dei centroidi iniziali diverso dà risultato diverso**. La distanza dei punti rispetto ai centroidi si calcola come distanza Euclidea, cosine similarity, correlazione etc... La convergenza maggiore avviene nelle prime iterazioni. La complessità è $O(n_{points} * n_{clusters} * n_{iterations} * n_{attributes})$.

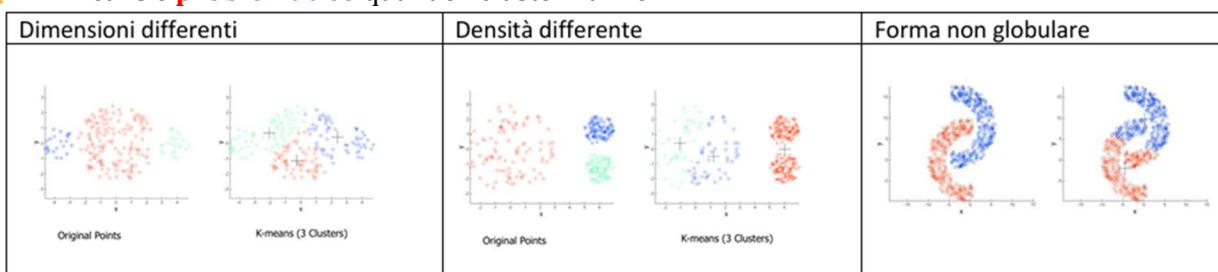
Per vedere quanto è efficace un certo clustering si usa $SSE = \sum_i \sum_{x \in C_i} dist^2(m_i, x)$ dove x = punto nel cluster C_i e m_i = centroide di C_i (**se SSE >, clustering peggiore**).

La **scelta dei centroidi iniziali** quindi è cruciale perché influenza la qualità del clustering ottenuto e si fa con:

- Run multipli = clustering eseguito diverse volte e viene scelto quello con SSE <
- Soluzione mista = viene fatto un sample e applicato un algoritmo gerarchico
- Selezione dei centroidi distanti = vengono generati più di k centroidi e vengono scelti i più distanti
- Postprocessing = viene analizzato il risultato ottenuto al termine di un run per migliorarlo al next run
- Bisezionamento (vediamo dopo) ↓

La selezione del k iniziale viene fatta analizzando l'**elbow graph** e identificando il gomito della curva (si vede SSE rispetto a k e viene scelto k per cui il guadagno ottenuto dall'aggiunta di un centroide è trascurabile). C'è il problema dei **cluster vuoti** (diminuiscono il numero di cluster realmente analizzati, aumentando quindi la distanza e l'errore [perché i punti vengono distanziati quanto lo erano con un $k <$]) [risolto assegnando i centroidi scegliendo un punto dal cluster con SSE > o scegliendo il punto che influisce di più sull'SSE].

⚠ Il K-means è **problematico** quando i cluster hanno:



Oppure in presenza di **outliers**. Una **soluzione** è usare diversi cluster che verranno poi uniti nel postprocessing

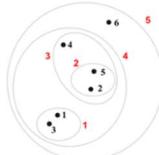
- **Bisectioning K-means (Bisezionamento)** → dividere in 2 di volta in volta il cluster con SSE più alto (quindi consiste nell'eseguire ricorsivamente un K-means con $k=2$ e può portare a soluzioni migliori rispetto al normale K-means)

- **Clustering gerarchico** → produce un insieme di cluster nidificati organizzati in un albero gerarchico e rappresentati con il **dendogramma**; scansionando il dendogramma dall'alto verso il basso, si individuano tutte le possibili soluzioni e aumenta il n° di cluster individuati. "Tagliare" il dendogramma significa scegliere 1 dei clustering (ogni intersezione rappresenta 1 cluster) [non ha bisogno di parametri in input perchè si può ottenere un qualsiasi n° di cluster]. Ci sono **2 tipi di clustering gerarchico** (entrambi usano una **matrice delle distanze o delle similarità**):

- **Agglomerativo** = partendo dai punti (considerati come cluster individuali) ad ogni step viene unita la coppia di cluster più vicina, fino a che non rimane 1 solo cluster
- **Divisivo** = partendo da 1 cluster, viene diviso di volta in volta fino a che ogni punto si trova in un cluster

Gli algoritmi più popolari sono gli **AGGLOMERATIVI** e funzionano così:

1. Si calcola la matrice di prossimità



2. Si assegna ogni punto ad un cluster diverso
3. Vengono uniti i cluster più vicini
4. Si aggiorna la matrice di prossimità (eliminando le righe e le colonne dei cluster uniti e aggiornando i valori di distanza)
5. Si ripetono i punti 3 e 4 fino a che non rimane 1 solo cluster

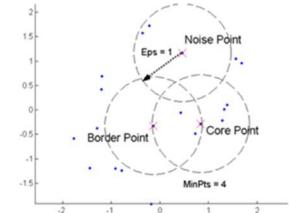
L'operazione chiave è il **calcolo della prossimità tra 2 cluster**, considerando le coppie di punti tali che 1 punto sia del 1° cluster e l'altro del 2°; ci sono diversi approcci:

- **Single link** (o MIN) = viene scelta come distanza fra i 2 cluster la distanza MIN tra le possibili coppie. Riesce ad identificare anche cluster con forma non globulare, ma rappresenta male i punti di contatto tra i gruppi (ignorando i dati rumorosi nel mezzo)
- **Complete linkage** (o MAX) = viene scelta come distanza fra i 2 cluster la distanza MAX tra le possibili coppie. Si comporta bene con dati rumorosi e outliers, ma tende a rompere cluster grandi e identifica spesso cluster di forma globulare
- **Group average** = la distanza fra 2 cluster è rappresentata dalla media di tutte le distanze delle possibili coppie di gruppi (i pro e i contro sono come il MAX)
- **Distanza fra i centroidi** = la distanza fra 2 cluster è la distanza fra i 2 centroidi; in caso di gruppi e punti, la distanza considerata è quella fra centroide e punto

Dato $N = n^{\circ}$ di punti, la complessità spaziale è $O(N^2)$ perché deve essere memorizzata la matrice di prossimità, mentre la complessità temporale è $O(N^3)$ perché ci sono N step e ad ogni step viene aggiornata la matrice.

- **DBSCAN** → algoritmo density-based che richiede 2 variabili in ingresso Eps e $MinPts$; ogni punto può essere classificato in 3 categorie (suddivisione per identificare rumore e outliers):

- **Core point** = ha più di $MinPts$ punti all'interno della circonferenza di raggio Eps
- **Border point** = ha meno di $MinPts$ punti all'interno della circonferenza di raggio Eps , ma è vicino ad un core point
- **Noise point** = è qualsiasi punto che non sia core o border



Dopo avere eliminato i **noise points**, viene fatto il clustering per tutti i punti rimanenti. Quindi per i core:

- Se il core point in questione non appartiene ad un cluster, gliene viene assegnato uno nuovo
- Il nuovo cluster viene assegnato anche a tutti i punti nel vicinato non ancora etichettati

Il DBSCAN è **resistente al rumore e agli outliers** e riesce a gestire **cluster di dimensioni/forme diverse**, ma non funziona bene con punti a densità variabile e per dati con molte dimensioni. La soluzione al problema della densità è il **multiple level clustering** (fare diversi run: nei primi presi i cluster a densità $> [Eps <]$, poi $<$). **La difficoltà del DBSCAN è la scelta dei parametri iniziali:** $MinPts = n^{\circ}$ minimo di punti che potremo avere in un cluster (e questo lo rende facile da scegliere); Eps viene scelto con il metodo k-dist (preso un k , si analizza la distribuzione delle distanze fra ogni punto e il suo k -esimo vicino; se c'è già un $MinPts$, $k = MinPts$).

Per verificare la **validità di un algoritmo di clustering**, bisogna confrontare i risultati ottenuti da algoritmi diversi o quelli ottenuti dallo stesso algoritmo ma con parametri diversi. Il clustering si può analizzare con un diagramma di dispersione dei centroidi, un radar graph, un boxplot e l'analisi di variabile esterne.

Gli **INDICI DI VALIDAZIONE** possono essere:

- ❖ **INTERNI** = sfruttano solo la conoscenza contestuale del clustering (senza utilizzo di dati aggiuntivi):

- **Cluster Cohesion** → quanto sono correlati gli oggetti in un cluster:

$$WSS = \sum_i \sum_{x \in C_i} (x - m_i)^2 \quad \text{con} \quad \begin{cases} x = \text{punto} \\ m_i = \text{centroide del punto} \\ C_i = \text{cluster} \end{cases}$$

- **Cluster Separation** → quanto un cluster è distinto dagli altri cluster:

$$BSS = \sum_i |C_i| (m - m_i)^2 \quad \text{con} \quad m = \text{valore medio del dataset}$$

- **Silhouette** → quanto ogni oggetto si trova bene nel proprio cluster (coesione $a(i)$ + separazione $b(i)$):

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \quad \text{con} \quad \begin{cases} a(i) = \text{dissimilità media dell'oggetto } i \text{ con gli altri del suo cluster} \\ b(i) = \min \text{ tra i valori } a(i) \text{ ma rispetto agli oggetti degli altri cluster} \end{cases}$$

- **Rand-index** → valuta quanto 2 partizionamenti sono simili:

$$Rand_{Index} = \frac{f_{00} + f_{11}}{f_{00} + f_{01} + f_{10} + f_{11}}$$

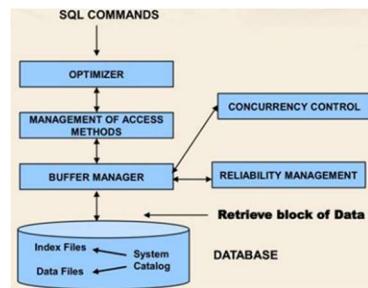
Dove f_{ij} = n° coppie di oggetti che appartengono alla stessa classe se $i=1$, appartengono allo stesso cluster se $j=1$ (0 se non appartengono)

- ❖ **ESTERNI** = misurano quanto le etichette del clustering corrispondono a etichette di classe note a posteriori:
 - **Entropia** (vogliamo minimizzata)
 - **Purezza** (vogliamo massimizzata)
- ❖ **RELATIVI** = comparano 2 clustering

10) DBMS

Il **DBMS** (Data Base Management System) è un software per memorizzare e gestire i database; è composto da:

- **Ottimizzatore** → riceve in input la query SQL (ne fa analisi lessicale, sintattica e semantica), ma la rappresenta con l'**algebra relazionale**. Sceglie un piano di esecuzione per l'accesso ai dati e l'esecuzione delle operazioni. Garantisce indipendenza dei dati (la forma con cui la query SQL viene scritta non influisce nel modo in cui il risultato viene calcolato)
- **Access Method Manager** → accede ai dati usando la strategia dell'ottimizzatore
- **Buffer Manager** → gestisce l'area riservata al DBMS nella memoria principale, usata per gestire le operazioni di lettura e scrittura delle pagine (unità di informazione in un DB). Il blocco di memoria è preallocato e condiviso tra le applicazioni che hanno accesso al DB
- **Concurrency Control** → gestisce l'accesso concorrente ai dati (le applicazioni non devono interferire tra loro)
- **Reliability Manager** → garantisce la correttezza del contenuto del DB in caso di crash, garantendo l'atomicità e la durabilità delle transazioni (2 delle proprietà ACID)



Una **TRANSAZIONE** è l'unità logica di lavoro eseguibile da un'applicazione: rappresenta una sequenza di 1 o più istruzioni SQL che eseguono operazioni di lettura/scrittura nel DB. Una nuova transazione inizia implicitamente quando viene riconosciuto il 1° comando SQL e può terminare in 2 modi:

- **COMMIT** → transazione eseguita correttamente
- **ROLLBACK** → transazione non eseguita per errore e il DB è stato ripristinato alla situazione pre-transazione [se richiesto dalla transazione stessa = suicide; se richiesto dal sistema = murder]

Una transazione ha le proprietà **ACID**:

- **Atomicity** = transazione non può essere divisa in unità più piccole. È garantita dalle primitive Undo (sistema disfa le operazioni fatte dalle transazione fino al punto corrente; usato per il rollback) e Redo (sistema riesegue le operazioni eseguite dalle transazioni committate; usato se failure)
- **Consistency** = esecuzione di una transazione non deve violare l'integrità del DB; se avviene violazione, il sistema deve poter fare il rollback dell'ultima transazione
- **Isolation** = esecuzione di una transazione indipendente dall'esecuzione corrente di altre transazioni
- **Durability** (Persistenza) = effetto di una transazione committata non deve essere perso nel tempo

Ora vediamo meglio le parti viste prima nel DBMS:

- **BUFFER MANAGER** → gestisce il trasferimento delle pagine dal disco alla memoria e viceversa. Il buffer è un blocco di memoria preallocato al DBMS condiviso fra tutte le transazioni in esecuzione; è organizzato in pagine la cui dimensione dipende dalla dimensione del blocco I/O dell'OS. Usa la **località temporale** (probabile che dati recenti vengano riusati a breve) e la legge empirica **20-80** (il 20% dei dati sono letti/scritti dall'80% delle transazioni). Per ogni pagina del buffer mantiene informazioni come:
 - **locazione fisica** della pagina nel disco (id del file e n° blocco)
 - variabili di stato (**count** = n° transazioni che stanno usando una certa pagina; **dirty bit** = se la pagina è stata modificata)

Il buffer manager fornisce **primitive**:

- **Fix** = usata dalle transazioni per richiedere accesso ad una pagina sul disco:
 - pagina richiesta viene cercata nel buffer

- se è nel buffer, count ++
 - se non viene trovata, bisogna identificare una posizione nel buffer in cui caricare la nuova pagina:
 - se non si trovano pagine libere, si seleziona la prima pagina libera che ha count=0 e se ha anche dirty=1, viene scritta su disco
 - l'indirizzo della pagina nel buffer viene restituito alla transazione che l'ha richiesta
- **Unfix** = dice al buffer manager che la transazione non sta più usando una certa pagina (count --)
- **Set dirty** = dice al buffer manager che la transazione ha modificato la pagina richiesta (dirty=1)
- **Force** = richiede trasferimento sincrono di una pagina nel disco (la transazione in esecuzione viene stoppata fino a che la primitiva non è stata eseguita; provoca sempre una scrittura su disco)
- **Flush** = trasferimento di una pagina su disco senza bloccare le transazioni (viene fatta quando la CPU è in idle; salvare le pagine con count = 0 e quelle non usate da tempo)

Il buffer manager ha diverse **strategie di scrittura** (solo steal e no force velocizzano, ma possono guastare):

- **Steal** = il buffer manager può selezionare una pagina bloccata (lock, bloccata da una transazione) con count=0 quando ha bisogno di uno slot libero. La politica steal salva su disco pagine dirty che appartengono a transazioni non committate (in caso di failure bisogna però poter fare rollback)
- **No steal** = il buffer manager non può selezionare pagine lockate se ha bisogno di uno slot libero
- **Force** = quando viene fatto commit, buffer manager scrive su disco le pagine attive di una transazione
- **No force** = dopo un commit, le pagine vengono scritte asincronicamente (con la flush). Le pagine che appartengono a transazioni committate potrebbero venire scritte su disco dopo la commit; in caso di failure prima del salvataggio su disco, i cambiamenti devono poter essere riapplicati con redo

⚠ Il buffer manager sfrutta diversi **servizi forniti direttamente dal filesystem** (creation/deletion of a file, open/close a file, read, sequential read, write, sequential write)

- **ACCESS METHOD MANAGER** → trasforma il piano di accesso scelto dall'ottimizzatore in una **sequenza di richieste di accesso fisico alle pagine del disco** (DB) usando diversi metodi di accesso. Ogni **metodo di accesso** è un modulo software specializzato in una tipologia di scrittura fisica e offre primitive di read/write; i metodi di accesso selezionano il blocco di un file che deve essere caricato in memoria sapendo a priori la distribuzione dei dati nei diversi file.

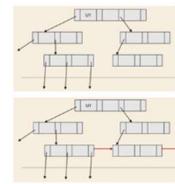
Una **pagina** nel disco è composta da spazio per la memorizzazione dei dati, spazio per le informazioni di controllo sui metodi di accesso e spazio riservato alle informazioni di controllo del filesystem; inoltre le **tuple** possono avere dimensioni diverse sia per i tipi di varchar sia per la presenza di valori nulli.

Le **STRUTTURE DI ACCESSO FISICO** dicono come i dati sono memorizzati nel disco; sono divise in:

- **Physical data storage**: contengono l'effettivo contenuto informativo:
 - **Strutture sequenziali** → tuple memorizzate nella pagina in maniera sequenziale:
 - **Heap file** = tuple messe in sequenza in base all'ordine di inserimento (quindi lettura e scrittura sequenziali efficienti); le operazioni di delete e update possono creare spazi vuoti inutili. Viene usato nei DBMS relazionali (insieme ad un indice unclustered per il supporto alle operazioni di ricerca e ordinamento)
 - **Struttura sequenziale ordinata** = ordine di scrittura delle tuple dipende da una sort key (formata da 1 o più attributi). È utile per ordinamento, group by, ricerca e join (tutte sulla sort key). Un problema è tenere l'ordinamento quando vengono inseriti nuovi dati; si risolve in 2 modi:
 - Lasciare libera una % di spazio in ogni blocco (nella creazione della tabella)
 - Predisporre file di overflow (con le tuple che non entrano nel blocco giusto)
 - Usano indici B⁺-Tree clustered e vengono usate per memorizzare risultati intermedi
 - **Strutture hash** → garantiscono accesso diretto ed efficiente ai dati basato completamente sul valore della chiave. La posizione del record viene determinata con un hash della chiave (che avrà valore compreso tra 0 e B-1 con B = n° blocchi), il quale determina il blocco in cui cercare/inserire la tupla [i blocchi non devono mai essere riempiti completamente perché dovrà inserire altri dati].
 - ✓ Pro = efficiente su query con uguaglianza sulla chiave; non richiesto ordinamento
 - ✗ Contro = inefficiente su query con predicati "range"; possono avvenire collisioni

- **Indici:** utili per incrementare l'efficienza di accesso:
 - **Strutture ad albero** (B-Tree, B⁺-Tree) → usate nei DBMS relazioni, offrono accesso diretto ai dati basato sul valore di un campo chiave (1 o più attributi). Un albero è composto da 1 nodo radice, ogni nodo ha diversi figli e sono i nodi foglia che forniscono l'accesso effettivo ai dati. Ci sono 2 tipi di strutture:

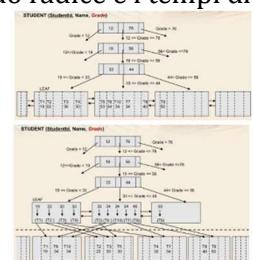
- **B-Tree** = pagine possono essere raggiunte solo visitando l'albero partendo dalla radice
- **B⁺-Tree** = struttura di collegamenti che consente accesso sequenziale, ordinato in base ai valori della chiave



In entrambi i casi, **B = bilanciato** (ogni foglia è alla stessa distanza dal nodo radice e i tempi di accesso sono sempre gli stessi per qualsiasi valore cercato).

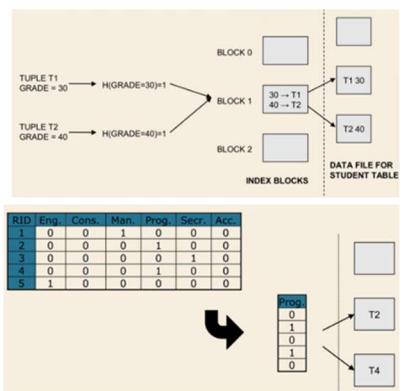
Se albero:

- **Clustered** = intero contenuto della tupla è nei nodi foglia e si usa per indexing rispetto alla primary key (indici primari)
- **Unclustered** = i nodi foglia contengono un sottoinsieme del contenuto della tupla e un *puntatore* alla tupla in memoria (usato per indici secondari)



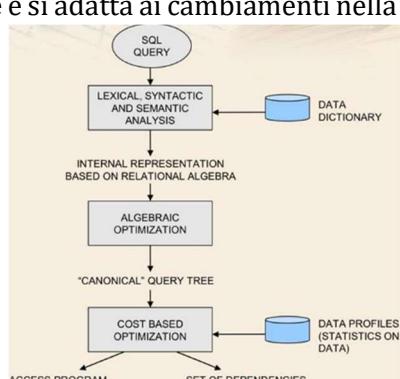
Quindi:

- ✓ **Pro** = efficiente su query con predicati "range", efficiente per scan sequenziale in ordine del campo chiave
- ✗ **Contro** = gli inserimenti potrebbero richiedere lo split di una foglia (richiede computazione), le eliminazioni potrebbero richiedere l'unione di nodi vuoti ed un possibile ribilanciamento
- **Unclustered hash index** = si differenzia dalle strutture hash semplici perché è unclustered (il blocco contiene solo un sottoinsieme delle informazioni della tupla e un puntatore alla tupla in memoria)
- **Bitmap index** = usa un campo chiave e si basa su una **matrice di bit**; questa matrice referenzia i record mediante il **RID** (Row IDentifier) e ha una colonna per ogni possibile valore diverso del campo chiave. Una cella viene messa a 1 se la tupla *i* ha valore *j*
 - ✓ **Pro** = efficiente per query con espressioni booleane di predicati e per attributi categorici; efficiente per il bitmapjoin e per attributi con dominio di valori limitato
 - ✗ **Contro** = non usabile per valori continui (va discretizzato) e con attributi con tanti possibili valori



- **OTTIMIZZATORE DELLE QUERY** → seleziona la **strategia migliore** per l'esecuzione della query e garantisce l'indipendenza dei dati (la forma con cui viene scritta la query SQL non influisce sul modo in cui viene calcolato il risultato) [riorganizzazione fisica non richiede riscrittura delle query]. Il piano di esecuzione generato è il più **efficiente** (vengono valutate alternative, statistiche, strategie e si adatta ai cambiamenti nella distribuzione dei dati). L'ottimizzatore fa **3 operazioni**:

- **Analisi lessicale** (parole scritte male), **sintattica** (errori sulla sintassi SQL) e **semantica** (riferimenti ad oggetti che non esistono nel DB [usa *data dictionary*]) → restituisce una rappresentazione della query in **algebra relazionale** (perchè mostra l'ordine con il quale gli operatori vengono applicati proceduralmente)
- **Ottimizzazione algebrica** → si fanno trasformazioni algebriche per rendere più efficienti le operazioni (es. anticipare selezione rispetto al join). L'output è una rappresentazione sempre in algebra relazionale chiamata "**canonical query tree**".



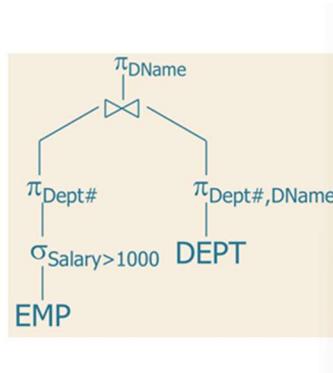
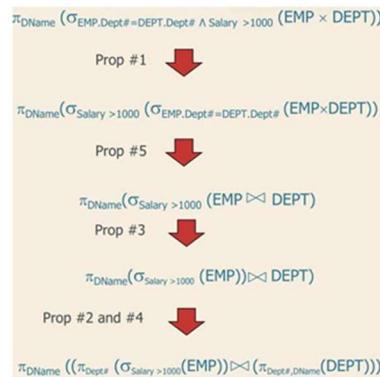
L'ottimizzazione algebrica si basa sulle **trasformazioni di equivalenza** (2 espressioni relazionali si dicono **equivalenti** [\equiv] se producono lo stesso risultato); queste trasformazioni possono **ridurre la dimensione dei risultati intermedi** per velocizzare le operazioni successive. Tipi di trasformazioni:

1. Atomizzazione della selezione
2. Proiezioni in cascata
3. Anticipazione (push down) della selezione rispetto alla join [F = predicato sugli attributi solo di E_2]
4. Anticipazione della proiezione rispetto alla join
5. Derivazione della join dal prodotto cartesiano [F contiene solo attributi contenuti in entrambe le tabelle]
6. Distribuzione della selezione rispetto all'unione
7. Distribuzione della selezione rispetto alla differenza
8. Distribuzione della proiezione rispetto all'unione
9. Distribuzione del join rispetto all'unione
10. Altre proprietà

⚠ Tutti gli operatori binari sono **commutativi e associativi**
eccetto la differenza

Esempio di ottimizzazione di una query:

```
SELECT DISTINCT DName
FROM EMP, DEPT
WHERE EMP.Dept#=DEPT.Dept#
AND Salary > 1000;
```



- **Ottimizzazione sulla base dei costi** → si seleziona il miglior metodo di accesso per ogni tabella, il miglior algoritmo per ogni operatore relazionale e viene generato il codice che ha la migliore strategia (solo qui si usano i **data profiles**, ovvero statistiche sui dati). I **data profiles** sono informazioni quantitative delle caratteristiche delle **tabelle** e delle colonne:

- Cardinalità di ogni tabella (+ stima della cardinalità dei risultati intermedi)
- Dimensione in Byte delle tuple T
- Dimensione in Byte di ogni attributo A_j in T
- Numero di valori distinti di ogni attributo in T (cardinalità del dominio attivo dell'attributo)
- Valori min e max per ogni attributo A_j in T
- Statistiche fini che caratterizzano la distribuzione vera e propria mediante istogrammi

I profili delle tabelle (data profiles) sono memorizzati nel **data dictionary** e vanno aggiornati periodicamente rianalizzando i dati nelle tabelle. Il comando per l'aggiornamento delle statistiche deve essere inserito dall'utente **on demand** (un'esecuzione immediata sovraccaricherebbe il sistema).

I profili sono usati anche per stimare la dimensione delle espressioni relazionali intermedi.

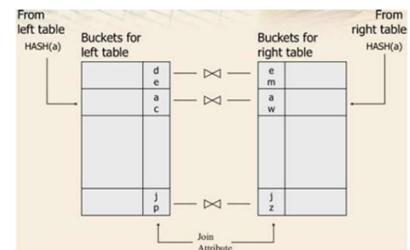
L'output è il **programma di accesso in formato eseguibile** dal DBMS e il **set di dipendenze** (condizioni dalle quali dipende la validità del piano di esecuzione [es. esistenza di un certo indice]).

⚠ Le query si possono **eseguire in 2 modi** (guarda poi piano di esecuzione):

- **Compile & Go** = dopo la compilazione, il piano di esecuzione viene direttamente eseguito, ma non viene memorizzato (non è poi disponibile; non sono necessarie le dipendenze)
- **Compile & Store** = il piano di esecuzione viene memorizzato con le dipendenze e viene eseguito on demand (se cambia la struttura deve essere ricompilato)

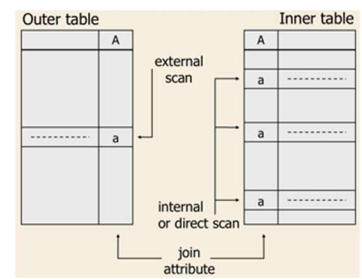
Parliamo ora del **QUERY TREE** ottenuto dall'ottimizzazione algebrica, che è la rappresentazione grafica di una espressione relazionale:

- **Foglie** = strutture fisiche (come tabelle o indici)
- **Nodi intermedi** = operazioni intermedie sui dati (come scan, join o group by):
 - **Scan sequenziale** = viene fatto un accesso sequenziale a tutte le tuple di una tabella (full table scan)
 - **Ordinamento** (costo > se > dimensione dei dati)
 - **Valutazione dei predicati di selezione** (accessi indicizzati per predicati selettivi):
 - **Uguaglianza** (semplice) → hash, B⁺-Tree o bitmap
 - **Range** → B⁺-Tree
 - **Congiunzione di predicati (AND)** → l'attributo più selettivo deve essere valutato per 1° (rendendo le valutazioni dei predicati dopo più snelle). Un'ottimizzazione è calcolare la intersezione dei bitmap/RID provenienti dagli indici per poi leggere dalla tabella solo le tuple individuate
 - **Disgiunzione di predicati (OR)** → si possono usare gli indici solo se tutti i predicati (tutti i loro attributi) hanno un indice, altrimenti si deve fare un full table scan

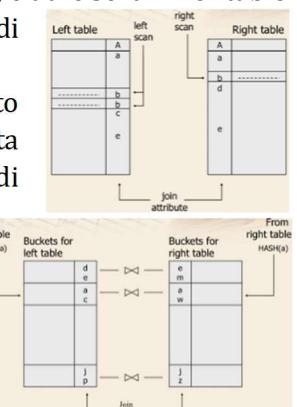


- **Join** (diversi algoritmi):
 - **Nested loop** → dopo aver trovato la inner table (la più piccola) e la outer table, per ogni tupla dell'outer table viene fatta una scansione completa della inner table [come nested loop con $i = \text{outer table}, j = \text{inner table}$]. L'accesso alla inner table si velocizza con caricamento in memoria della tabella oppure se viene usato un indice sull'attributo di join.

Non è simmetrico (cambia il costo di esecuzione se scambio le tabelle); è utile se la inner table è ordinata e abbastanza piccola da metterla in memoria (in assenza di indice)



- **Merge scan** → richiede che le 2 tabelle siano già ordinate (rispetto all'attributo di join) e consiste in 1 scan parallelo che di volta in volta genera le coppie di tuple che soddisfano la condizione di join. Il costo di esecuzione è simmetrico

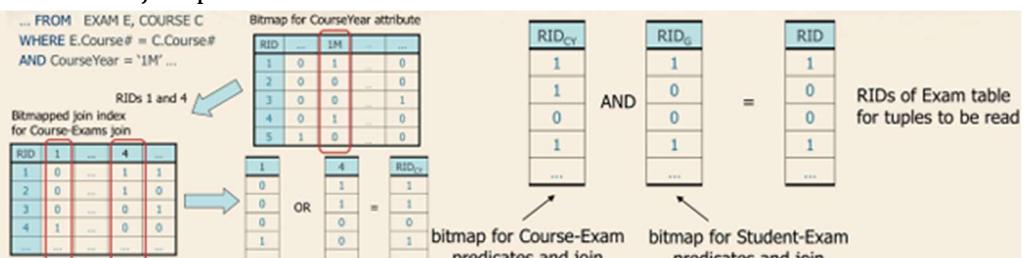


- **Hash join** → le tuple di entrambe le tabelle vengono bucketizzate rispetto all'attributo di join in modo che sia possibile fare un veloce ordinamento interno al bucket e poi fare il join tra le tuple all'interno degli stessi bucket delle 2 tabelle (utile per tabelle molto grandi)
- **Bitmapped join** → indice precompilato che consiste in 1 matrice di bit dove le righe e le colonne contengono i RID delle 2 tabelle (una cella contiene 1 se la riga i soddisfa la condizione di join con la colonna j). L'aggiornamento potrebbe risultare lento (usato nei sistemi di DW per la join fra la tabella dei fatti e le dimensioni). Può usare indici bitmapped definiti sull'attributo di join per entrambe le tabelle

RID	1	2	...	n
1	0	0	...	1
2	0	1	...	0
3	0	0	...	1
4	1	0	...	0
...	0

- **STUDENT (Reg#, SName, Gender)**
- **COURSE (Course#, CName, CourseYear)**
- **EXAM (Reg#, Course#, Date, Grade)**

```
SELECT AVG (Grade)
FROM STUDENT S, EXAM E, COURSE C
WHERE E.Reg# = S.Reg#
AND E.Course# = C.Course#
AND CourseYear = '1M'
AND Gender = 'M';
```



Tramite l'indice bitmapped su CourseYear vengono selezionati i RID corrispondenti al 1° anno magistrale (1M); poi, isolando nella matrice del bitmapped join le colonne corrispondenti ai RID e mettendole in OR, otteniamo la lista di RID che soddisfano il join. Alla fine, con un AND tra i RID corrispondenti ai 2 predicati di selezione, otteniamo la lista di RID di Exam che deve essere considerata per il calcolo della media. In questo modo l'accesso in memoria è veloce

- **Group by:**

- **Sort based** → tuple vengono ordinate in base all'attributo di join e poi calcolo gli aggregati
- **Hash based** → applico hashing per bucketizzare le tuple, poi ordinate e calcolo aggregati

- **Viste Materializzate** (VM) → se la funzione di query rewrite è attiva sulla vista, l'ottimizzatore può riscrivere le query in modo da sfruttare il raggruppamento preventivo effettuato da essa
 - ⚠ Se non ho ancora fatto sort e hash based, posso fare una group by “no-sort” o “no-hash”

Parliamo ora dell'**INDIVIDUAZIONE DEL PIANO DI ESECUZIONE** (dipende dal modo in cui i dati vengono letti dalla memoria, dall'ordine di esecuzione degli operatori, dall'implementazione degli operatori, da quando e se deve essere fatto ordinamento). L'ottimizzatore individua un **albero con le diverse alternative** in cui ogni nodo interno è una decisione su 1 dei fattori e le **foglie sono i diversi piani di esecuzione individuati**. **Esempio:** consideriamo 3 tabelle su cui applicare il join ($T_1 \text{ join } T_2 \text{ join } T_3$) → abbiamo quindi 4 tecniche di join da poter scegliere (per entrambi i join) e 3 diversi ordinamenti → albero con $4 \times 4 \times 3 = 48$ foglie (possibili piani di esecuzione).

⚠ L'ottimizzatore andrà a selezionare il nodo foglia con costo < e il costo viene calcolato come $C_{tot} = C_{I/O} * n_{I/O} + C_{cpu} * n_{cpu}$

→ PROGETTAZIONE FISICA

Qui ci **fermiamo un attimo sui componenti** (poi riprendiamo) e parliamo di **PROGETTAZIONE FISICA**: ha l'obiettivo di definire strutture dati per l'ottimizzazione dell'esecuzione delle query (non influisce sui risultati). Prima di farla, bisogna:

- Definire lo **schema logico**
- Individuare il **DBMS** usato e la sua lista delle features
- Analizzare il carico di lavoro (**workload**):
 - Le **query più importanti** e la loro frequenza stimata
 - Le **operazioni di update** e la loro frequenza stimata
 - Le **performance richieste** per le query rilevanti e per gli update

⚠ Mentre la progettazione logica non cambia, la progettazione fisica deve cambiare nel tempo per **adattarsi**

La progettazione fisica ha come **output**:

- Organizzazione delle **tabelle**
- **Indici** da usare
- **Parametri di setup** per lo storage e per il DBMS

Ci sono diverse **strutture fisiche**: **non ordinate** (heap), **ordinate** (clustered), **hashing**, **insiemi di relazioni** (tuple appartenenti a tabelle diverse possono essere interlacciate).

Riguardo al **workload**, la caratterizzazione del carico di lavoro deve avvenire:

- Per ogni **query**:
 - Tabelle usate
 - Attributi visualizzati
 - Attributi usati in operazioni di selezione e join
 - Selettività della selezione
- Per ogni **update**:
 - Attributi e tabelle coinvolti nella selezione
 - Selettività della selezione
 - Tipo di update (insert, delete, update) e attributi coinvolti

Dopo l'analisi del workload, vengono selezionate le **strutture fisiche** da usare per i **dati** e per gli **indici**. Per ogni **tabella** va individuata la **struttura dei file** (heap o clustered) e gli **attributi da indicizzare** (hash o B⁺-Tree, clustered o unclustered) [se modifichiamo modello logico, potrei dover modificare schema fisico]

Ci sono 2 criteri principali per la **selezione delle strutture dati**:

- Dato che la **chiave primaria** è spesso usata per selezione e join, è opportuno indicizzarla
- Si possono aggiungere indici anche per le **query più comuni**:
 - Selezionare una query frequente

- Considerare il suo piano di esecuzione corrente
- Definire un nuovo indice e considerare il nuovo piano di esecuzione (se i tempi migliorano, si tiene)
- Verificare l'effetto del nuovo indice sul workload e sulla memoria

⚠ Mai creare indici per:

- **Tabelle piccole** (dato che il caricamento dell'intera tabella richiede già poche letture)
- **Attributi con cardinalità di dominio bassa** (sono poco selettivi; non si applica ai DW in quanto si possono implementare mediante indici bitmap)

⚠ Per gli attributi appartenenti a **predicati semplici di una where:**

- Predicati di **uguaglianza** → hash (migliore) o B⁺-Tree
- Predicati **range** → B⁺-Tree

⚠ Per **where con molti predicati semplici** potrebbe essere utile usare **indici composti** (multi-attributo) selezionando l'ordine (prima il più selettivo) di chiavi appropriato e considerando che questo tipo di indici hanno un costo di manutenzione alto

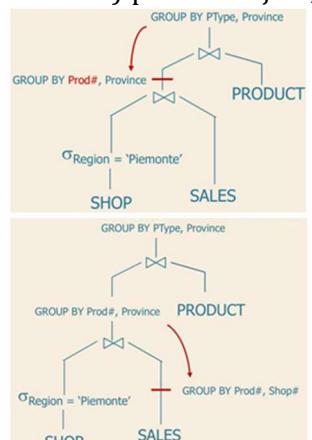
Si possono eseguire operazioni di:

- **JOIN** → mediante:
 - **Nested loop** = utile con tabelle sbilanciate; va indicizzata la inner table (vedi sopra funzionamento)
 - **Merge scan** = utile con tabelle grandi e già ordinate; conviene usare B⁺-Tree clustered (// //)
- **GROUP BY** → mediante:
 - **Ordinamento per generare i gruppi**
 - **Funzione di hash**

Per migliorare la group by si potrebbe indicizzare gli attributi della group by con hash o B⁺-Tree. La group by inoltre può essere automaticamente anticipata dall'ottimizzatore (**group by "push down"**) prima dei join, consentendo di produrre meno tuple su cui fare il join. **Esempio:**

```
PRODUCT (Prod#, PName, PType, PCategory)
SHOP (Shop#, City, Province, Region, State)
SALES (Prod#, Shop#, Date, Qty)

SELECT PType, Province, SUM (Qty)
FROM Sales S, Shop SH, Product P
WHERE S.Shop# = SH.Shop#
AND S.Prod# = P.Prod#
AND Region = 'Piemonte'
GROUP BY PType, Province;
```



Lo schema relazionale mostra l'ordine secondo cui la query deve essere eseguita:

1. Lettura delle tabelle
2. Predicato di selezione (su regione) che riduce cardinalità di Shop
3. Join fra Shop e Sales
4. Join con Product
5. Group by
6. Operatore di proiezione che seleziona solo i campi della select

Come si vede, anticipando la group by di volta in volta riduce il volume di dati prima delle join (non è possibile però in questo caso eliminare del tutto le group by superiori) [è sempre utile anticipare le group by quando si può]

Quando l'ottimizzatore fa il piano di esecuzione usa le statistiche derivate dai piani di esecuzione precedenti (questi dati rendono l'esecuzione efficiente ma vanno aggiornati periodicamente dall'utente).

Si possono usare gli **HINTS** (suggerimenti all'ottimizzatore rispetto agli indici da usare) [l'ottimizzatore in questo caso userà gli hints e non le statistiche]. **Esempi sulla creazione degli indici dato questo schema logico:**

```
EMP (Emp#, EName, Dept#, Salary, Age, Hobby)
DEPT (Dept#, DName, Mgr)
  • In EMP
    Dept# FOREIGN KEY REFERENCES DEPT.Dept#
  • In DEPT
    Mgr FOREIGN KEY REFERENCES EMP.Emp#
```

1. In questo caso è opportuno inserire un indice sull'attributo Salary (B⁺-Tree); in questo caso però l'indice potrebbe essere ignorato a causa dell'espressione aritmetica
2. Questa query è equivalente alla 1^a, ma senza espressione aritmetica. Dobbiamo considerare la selettività di questa condizione, considerando anche che, avendo * nella SELECT, avremo bisogno di salvare intere tuple nell'indice.
3. Supponiamo ora che la tabella EMP contenga 30 tuple per blocco:

```
SELECT *
FROM EMP
WHERE Salary/12 = 1500;
```

```
SELECT *
FROM EMP
WHERE Salary = 18000;
```

- a. $\text{Card(DEPT)}=50 \rightarrow$ usando l'indice dovrà leggere 1-2 blocchi per la lettura dell'indice e 1 blocco per la lettura della tupla dalla memoria; se uso accesso sequenziale, leggo direttamente **max 2 blocchi** dalla memoria (in questo caso la creazione dell'indice **non conviene**)
- b. $\text{Card(DEPT)}=2000 \rightarrow$ usando l'indice dovrà leggere i blocchi contenenti l'indice e 1 blocco per la lettura della tupla dalla memoria; se uso accesso sequenziale, leggo dalla memoria **max 70 blocchi** (qui l'indice **conviene se il predicato è selettivo**)

4. In questo caso si possono creare 2 indici:

- a. 1 hash index su DName per la condizione di selezione
- b. 1 hash index su E.Emp#, per ordinare la chiave di join per un nested loop con Emp = tabella interna

5. In questo caso si può usare un indice su Age, ma dipende dalla selettività della condizione

6. Abbiamo 2 alternative relative ai predicati di selezione:

- a. Hash index su Hobby
- b. B^+ -Tree su Salary

In questo caso verrà considerato dall'ottimizzatore solo 1 dei 2 indici. Per quanto riguarda la join ci sono 2 alternative:

- Hash join
- Nested loop = considerando DEPT come tabella interna in quanto i predicati di selezione sono su EMP, usiamo anche 1 indice su DEPT.Dept# che non è conveniente se DEPT è molto piccola

7. Dato che l'attributo Age non è molto selettivo usiamo un B^+ -Tree. Per la GROUP BY:

- a. Indice clustered su Dept# (evita l'ordinamento; la lettura degli impiegati è storicizzata in base al dipartimento)
- b. Indice secondario su Dept# (evita l'ordinamento, ma per ogni dipartimento nelle foglie devo fare l'accesso alla tabella EMP)

8. Indice unclustered su Dept#, dove saranno presenti n foglie per ogni Dept. Questo indice si dice **coprente** (risponde alle query in modo completo)

9. Può essere conveniente usare un indice **composto** Age, Salary; questo è il migliore se la condizione su Age è molto selettiva. Negli indici composti, l'ordine dei campi è importante, ma è difficile da mantenere

```
SELECT EName, Mgr
FROM EMP E, DEPT D
WHERE E.Emp#= D.Mgr
AND Dname = 'Toys';
```

```
SELECT EName, Mgr
FROM EMP E, DEPT D
WHERE E.Emp#= D.Mgr
AND Dname = 'Toys'
AND Age=25;
```

```
SELECT EName, Mgr
FROM EMP E, DEPT D
WHERE E.Emp#= D.Mgr
AND Salary BETWEEN 10000 AND 12000
AND Hobby='Tennis';
```

```
SELECT Dept#, Count(*)
FROM EMP
WHERE Age>20
GROUP BY Dept#;
```

```
SELECT Dept#, COUNT(*)
FROM EMP
GROUP BY Dept#;
```

```
SELECT AVG(Salary)
FROM EMP
WHERE Age = 25
AND Salary BETWEEN 3000 AND 5000;
```

Riprendiamo il discorso componenti e parliamo di **OTTIMIZZATORE DI ORACLE** [valuta espressioni, riscrive l'operazione SQL (**query rewrite**), sceglie l'**obiettivo** dell'ottimizzatore, i **percorsi di accesso ai dati** in base agli indici disponibili, l'**ordine delle join** e l'algoritmo di **esecuzione della join**].

Il recupero dei dati dal DB può avvenire con:

- **Full table scans** → leggere tutte le tuple di una tabella eliminando quelle che non rispettano i criteri di selezione (ovvero le WHERE). I blocchi da leggere sono tutti adiacenti e vengono letti in modo sequenziale; si possono fare letture multiblocco (leggere più blocchi in 1 chiamata).
Le decisioni dell'ottimizzatore sono influenzate dall'**index clustering factor** (indica quanto sono sparsi i dati relativi ai valori di colonne simili all'interno dei blocchi della tabella):
 - se è basso, le righe con valore dell'attributo simile saranno concentrate negli stessi blocchi (creazione di indice conveniente)
 - se è alto, le righe con valore dell'attributo simile sono distribuite tra i vari blocchi (creazione indice non conveniente perché fa riferimento a tutti i blocchi quindi conviene full scan)
- **Index scans** → si possono definire diversi tipi di indici primari (clustered B-Tree, hash) e secondari (B-Tree, bitmap, hash), ma non si può dire come devono essere fisicamente realizzati.
Gli indici contengono il **valore indicizzato** e i **RID** (row id) delle righe nella tabella che hanno quel valore; ci sono diversi tipi di index scan:
 - **Index Unique Scans** = dà max 1 RID per ogni valore indicizzato (attributo indicizzato è unique)
 - **Index Range Scans** = dà diversi RID ordinati in modo crescente (utile quindi se le operazioni dopo richiedono ordinamento rispetto all'attributo indicizzato)
 - **Index Full Scans** = è disponibile se il predicato contiene 1 delle colonne nell'indice o se non c'è un predicato. Dà i dati dell'indice ordinati in base ai valori memorizzati nell'indice

- **Fast Full Index Scans** = è un'alternativa al full table scan usata quando l'indice è coprente. Usa l'indice senza accedere alla tabella; l'indice viene letto con lettura multiblocco (quindi risultato non ordinato)
- **RID (o Rowid) scans** → accedere alla tabelle usando il RID per prelevare il blocco con la tupla scelta. L'indirizzo fisico del blocco viene calcolato partendo dal RID; è il più veloce per prendere 1 tupla
- **Bitmap index** → efficaci per query che contengono condizioni multiple nella WHERE. Nel contesto del piano di esecuzione viene prima letto il bitmap index per ogni attributo e poi si fa una AND per ottenere l'insieme di RID che soddisfano le condizioni

⚠ Oracle ha **EXPLAIN PLAN FOR <...query...>** che mostra il piano di esecuzione scelto!

Fra le **statistiche** a disposizione dell'ottimizzatore ci sono gli histogrammi che raccolgono stime accurate sulla distribuzione dei dati relativamente ad 1 attributo. Gli **istogrammi** possono essere di 2 tipi:

- **Height-balanced** = i possibili valori dell'attributo sono divisi in fasce con lo stesso n° di record (discretizzazione). Nella figura a lato, vediamo che l'istogramma in alto ha distribuzione uniforme, mentre quello in basso non uniforme. In Oracle, la 1^a colonna rappresenta la fascia considerata e la 2^a il valore massimo in essa
- **Frequency** = per ogni possibile valore ne viene specificato il n° di occorrenze. In Oracle la 1^a colonna è la frequenza dei valori, la 2^a il valore considerato

Si può scegliere in che modalità deve operare l'ottimizzatore:

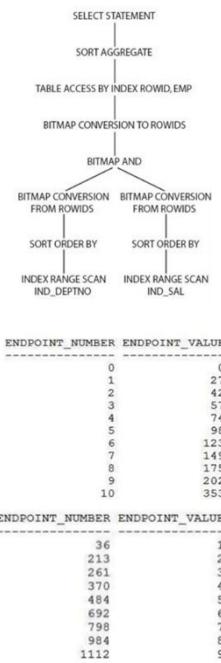
- ❖ Ottimizzazione per il **throughput** migliore (usare n° di risorse < possibile)
- ❖ Ottimizzazione per i **tempi di risposta** migliori (valori delle 1^a righe della query nel < tempo possibile)

Ora parliamo degli **HINT IN ORACLE** (ricorda hint = fornire istruzioni sulla scelta di un piano di esecuzione); ci sono diversi **tipi di hint differenziati per:**

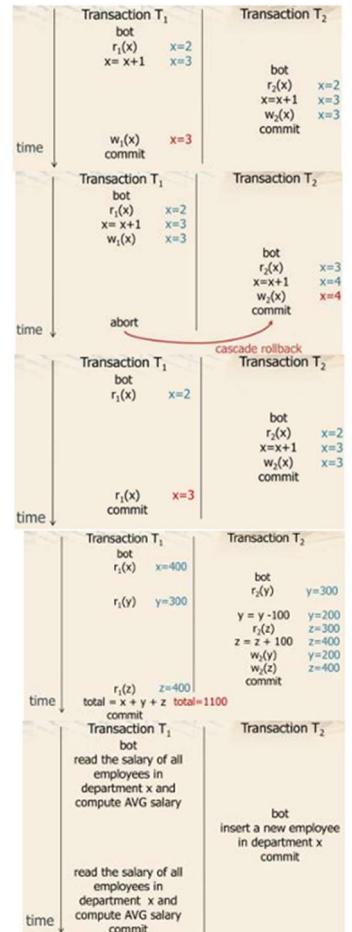
- **Obiettivo** e approccio di ottimizzazione:
 - **ALL_ROWS** = ottenere max throughput (min consumo risorse)
 - **FIRST_ROWS(n)** = risposta veloce (1^a n righe prodotto in modo efficiente)
- **Modalità di accesso:**
 - **FULL(table)** = usato il full table scan sulla tabella *table*
 - **INDEX (table indexName1...)** = index scan usando 1 degli indici specificati
 - **NO_INDEX(table indexName1...)** = viene evitato l'uso degli indici specificati
 - **INDEX_COMBINE (table indexName1...)** = usata modalità di accesso bitmap per gli indici scelti
 - **INDEX_FFS(table indexName1...)** = fast full index scan sugli indici scelti
 - **NO_INDEX_FFS(table indexName1...)** = esclude l'uso di fast full index scan sugli indici scelti
- **Trasformazione delle query**
- **Ordine di join:**
 - **ORDERED** = join fatto seguendo l'ordine nella FROM
 - **LEADING(table1, table2, ...)** = usa ordine scelto
- **Operazione di join usata** → [NO_]USE_NL(...), [NO_]USE_MERGE, [NO_]USE_HASH
- **Esecuzione parallela**
- Altri hint

- **GESTIONE DELLA CONCORRENZA** → il controllo della concorrenza fornisce la possibilità di accesso concorrente ai dati incrementando l'efficienza del DBMS che può **massimizzare il n° di transizioni al secondo** e **minimizzare il tempo di risposta medio** alle transazioni. Le operazioni di I/O elementari da gestire sono **lettura e scrittura** che possono richiedere l'accesso ad un'intera pagina.

Lo **SCHEDULER** gestisce la concorrenza e decide quando soddisfare le richieste di lettura/scrittura; senza di esso si possono verificare delle **anomalie**:



- **Lost update** = l'aggiornamento fatto da T2 su x , viene perso perché sovrascritto da quello fatto da T1 che ha letto x prima della scrittura di T2 (il valore di x giusto dovrebbe essere 4)



- **Dirty read** = T2 legge il valore di x in uno stadio intermedio che non diventa permanente, in quanto T1 fa abort
- **Inconsistent read** = la transazione T1 legge x 2 volte, ma i 2 valori letti sono diversi
- **Ghost update (a)** = T1 osserva solo parzialmente gli effetti di T2; infatti legge x e y prima dell'aggiornamento di T2 e dopo la fine di T2 legge z . Si è quindi perso l'aggiornamento su y fatto da T2 (il risultato sarebbe dovuto essere 1000)
- **Ghost update (b)** = le medie calcolate sono diverse e ciò è dovuto all'inserimento che rappresenta l'aggiornamento "fanstasma". La differenza rispetto all'inconsistent read è che in questo caso l'oggetto è stato inserito proprio durante T1. Non si può fare lock su un oggetto che ancora non esiste ma che è stato appena inserito

Ci sono definizioni da dare:

- **Transazione** = sequenza di operazioni di lettura e scrittura che hanno lo stesso **TID** ($r_1(x), w_1(y)$)
- **Schedule** = sequenza di operazioni di lettura e scrittura appartenenti a transazioni concorrenti (le operazioni appaiono nell'ordine di arrivo) ($r_1(z)r_2(z)w_1(y)w_2(z)$)
- **!** Lo scheduler accetta e rifiuta certe schedule per evitare le anomalie, senza sapere però l'esito finale della transazione
- **Commit projection** = applicata dallo scheduler, afferma che ogni transazione farà commit (non gestisce però l'anomalia di dirty read)
- **Schedule seriale** = le azioni di ogni transazione appaiono in sequenza, senza operazioni interlacciate di altre transazioni
- **Schedule corretta/serializzabile** = se fornisce gli stessi risultati di una qualsiasi schedule seriale S_j

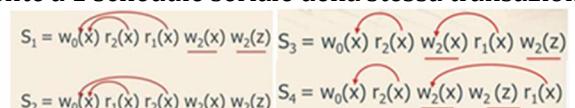
Parliamo quindi di **CLASSI DI EQUIVALENZA** fra 2 schedule:

- **Timestamp equivalence**
- **View equivalence** → se hanno gli stessi:
 - **reads-from** = $r_i(x)$ reads-from $w_j(x)$ se $w_j(x)$ precede $r_i(x)$ con $i \neq j$ e non c'è $w_k(x)$ fra esse
 - **final write** = $w_i(x)$ è una final write se essa è l'ultima scrittura di x nella schedule

Una schedule è view serializzabile se è view equivalente a 1 schedule seriale della stessa transazione:

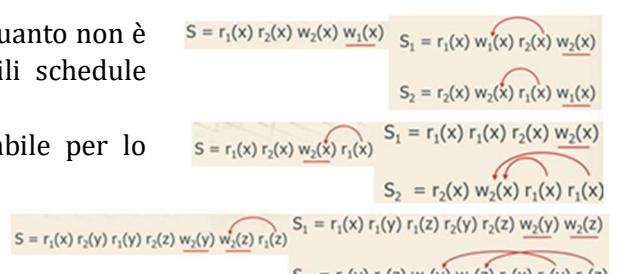
S_1 è view serializzabile perché è view equivalent a S_2

S_3 non è view equivalente a S_2 , ma lo è con S_4 , quindi è view serializzabile



Vediamo come si comporta rispetto alle anomalie:

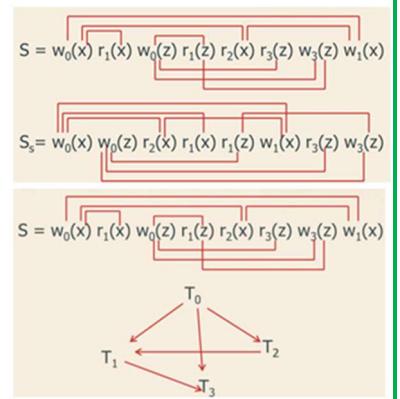
- ❖ **Lost update** = non è view serializzabile in quanto non è view equivalente a nessuno dei 2 possibili schedule seriali (viene rigettato)
- ❖ **Inconsistent read** = non è view serializzabile per lo stesso motivo (viene rigettato)
- ❖ **Ghost update (a)** = S anche qui non view serializzabile



La complessità di trovare la view equivalence per una schedule seriale è lineare, mentre la determinazione della view equivalence con uno schedule seriale non è realizzabile nella realtà (nei sistemi reali si sono scelte tecniche più veloci ma meno accurate)

- **Conflict equivalence** → 2 schedule sono conflit equivalent se hanno gli stessi conflitti e ogni coppia di azioni è nello stesso ordine in entrambe (2 azioni A_i e A_j sono in conflitto se operano sullo stesso oggetto e 1 delle azioni è una scrittura). Una schedule è **CSR** (conflict serializable) se è equivalente ad una schedule seriale scelta delle stesse transazioni.

Per trovare la conflict serializability si usa un grafo dei conflitti in cui ogni nodo è una transazione e ogni arco è il fatto che c'è almeno 1 conflitto tra A_i di T_i e A_j di T_j con A_i che precede A_j . Se il grafo è aciclico, lo schedule è CSR (l'equivalenza con l'uso del grafo ha complessità lineare rispetto alla dimensione del grafo).



Nei sistemi reali però nemmeno questo è ideale (**CSR** è sottoinsieme di VSR)

- **2 phase locking (2PL)** → un **lock** (ricorda OS semafori) è un blocco su una risorsa che ne previene l'uso da parte di altri. Le operazioni di lock possono essere **R-lock** (read, condivisa tra transazioni [semafori con conteggio]) o **W-lock** (write, esclusiva ad 1 transazione [mutex] [blocca ovviamente anche le altre read]); dopo l'uso della risorsa, si fa **unlock** (definiamo **lock escalation** = richiesta di una R-lock seguita da una W-lock sugli stessi dati).

Lo scheduler diventa quindi un lock manager:

- se richiesta di lock **concessa** = risorsa data alla transazione, che quando ha finito fa unlock
- se lock non **concesso** = transazione messa in attesa fino al prossimo unlock della risorsa

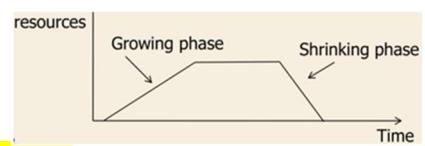
Per fare ciò il lock manager usa:

- **Lock table** (in memoria principale) = consente di decidere se lock può essere concesso o no. Per ogni oggetto di dati si usano 2 bit per rappresentare i 3 possibili stati (**free**, **r_locked**, **w_locked**) e un contatore per il n° di transazioni in attesa della risorsa
- **Conflict table** = gestire i conflitti tra le transazioni

Request	Resource State		
	Free	R-Locked	W-Locked
R-Lock	Ok/R-Locked	Ok/R-Locked	No/W-Locked
W-Lock	Ok/W-Locked	No/R-Locked	No/W-Locked
Unlock	Error	Ok/It depends (free if no other R-Locked)	Ok/Free

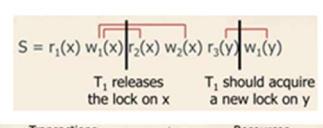
Il 2 phase locking (usato nella maggior parte dei DBMS commerciali) ha **2 fasi** (tra le 2 fasi, la transazione fa le operazioni sui dati):

- ❖ **Growing phase** = transazione può richiedere i lock
- ❖ **Shrinking phase** = rilasciati tutti i lock + non se ne possono richiedere di nuovi



Il 2PL garantisce serializzabilità (**2PL** sottoinsieme delle transazioni **CSR**).

Esempio: Dall'esempio si nota che la transazione analizzata è CSR in quanto il grafico è aciclico ma non 2PL in quanto già dopo la prima striscia nera si entra in fase di shrinking ma alla seconda striscia nera viene richiesto il lock di un'altra risorsa.



Guardando l'esempio successivo invece si nota che il 2PL riesce a gestire correttamente il Ghost update (a).

Transactions	Resources	Transactions	Resources
T ₁ : bot r ₁ (x) r ₁ (y)	x: free y: free z: free	T ₁ : commit unlock ₁ (x) unlock ₁ (y)	x: free y: free z: free
T ₂ : bot r ₂ (y) r ₂ (z)	1: read 2: read 1,2: read 2, read	T ₂ : w ₂ (y) w ₂ (z)	2: write free free

Lo **STRICT 2PL** è una restrizione che prevede che una transazione debba rilasciare le risorse solo dopo il commit/abort (quindi alla fine della transazione), in modo che i dati siano stabili ed evitare l'anomalia di dirty read.

Le **primitive** sono:

- **R-Lock** (T, x, ErrorCode, TimeOut)
- **W-Lock** (T, x, ErrorCode, TimeOut)
- **UnLock** (T, x)

Con T = TID, x = risorsa richiesta, $ErrorCode$ = parametro di ritorno che può essere Ok e Not Ok, $TimeOut$ = tempo di attesa max della transazione.

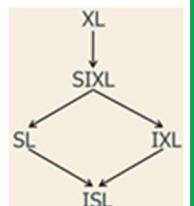
Quando una transazione richiede una risorsa x :

- ❖ Se la richiesta può essere **soddisfatta** = lock manager modifica lo stato di x nella sua tabella e restituisce *Ok* (piccola attesa, risorsa libera)
- ❖ Se la richiesta **non** può essere **soddisfatta al momento** = transazione messa in coda di attesa e quando la risorsa torna disponibile, la 1^a transazione in coda prende il lock della risorsa
- ❖ Se il **timeout scade** mentre la transazione è in attesa, il lock manager estrae la transazione dalla coda, la fa ripartire e restituisce *Not Ok*

La probabilità di avere conflitti è $\frac{K \cdot M}{N}$ con K = transazioni attive, M = n° medio di oggetti a cui ogni transazione fa accesso, N = n° oggetti nel DB

Esteso il 2PL, abbiamo il **LOCKING GERARCHICO** (posso acquisire lock a granularità diversa [tabelle, gruppi di tuple, etc...]). Abbiamo **primitive aggiuntive** (a dx il grafo di predecenza dei lock):

- **SL (Shared Lock)** = fa lettura
- **XL (eXclusive Lock)** = fa scrittura
- **ISL (Intention of Shared Lock)** = mostra l'intenzione di fare shared lock su un oggetto presente in un nodo figlio
- **IXL (Intention of eXclusive Lock)** = come ISL, ma per l'exclusive lock
- **SIXL (Shared Lock and Intention of eXclusive Lock)** = fa shared lock su oggetto corrente e mostra intenzione di fare exclusive lock su uno o più oggetti nei nodi figli



Il **request protocol** descrive come si comporta il lock manager per gestire questo tipo di scenari:

1. I lock vengono sempre concessi a partire dal nodo radice e scendendo fino ai nodi foglia
2. I lock vengono rilasciati a partire dal nodo con un lock a granularità maggiore a salire
3. Per richiedere SL o ISL di un nodo, transazione deve avere ISL o IXL sul suo nodo superiore
4. Per richiedere XL, IXL o SIXL su un nodo, transazione deve avere IXL o SIXL sul nodo superiore

Nella **matrice di compatibilità** vengono mostrate tutte le possibili combinazioni:

- In questo caso i due lock non sono compatibili in quanto XL è esclusivo e non è possibile accedere in lettura ad un figlio
- In questo caso SL non viene concesso perché qualcuno ha bloccato il figlio in scrittura
- In questo caso i due lock sono compatibili perché abbiamo due accessi in lettura
- In questo caso IXL non viene concesso perché la risorsa padre è bloccata in lettura e la transazione corrente vuole modificarne una parte.

Request	Resource State				
	ISL	IXL	SL	SIXL	XL
ISL	Ok	Ok	Ok	Ok	No
IXL	Ok	Ok	No	No	No
SL	Ok	No	Ok	No	No
SIXL	Ok	No	No	No	No
XL	No	No	No	No	No

⚠ La scelta della granularità dipende dal tipo di applicazione:

- ✓ Se fa lettura e update localizzati di pochi oggetti → granularità dettagliata
- ✓ Se fa lettura e update massivi → granularità grezza (overhead del lock manager)

Il **Locking dei predicati** consente di risolvere l'anomalia di **Ghost Update (b)**: di base in 2PL una read non è in conflitto con l'inserimento di una tupla perché essa non può essere bloccata in anticipo. Il locking del predicato permette invece di **bloccare tutti i dati che soddisfano un certo predicato** (implementato nei sistemi reali bloccando gli indici).

Nello standard SQL2, le transazioni possono essere read-write o read-only. Il **livello di isolamento** di una transazione specifica come questa interagisce con le altre transazioni in esecuzione e viene settato con comandi SQL:

- **SERIALIZABLE** = isolamento >, include il locking dei predicati e blocca tutte le anomalie
- **REPEATABLE READ** = strict 2PL senza locking dei predicati, non protegge da Ghost Update(b)
- **READ COMMITTED** = nessun 2PL, il R-lock viene rilasciato non appena l'oggetto è stato letto; viene evitata la lettura di stati intermedi evitando il dirty read
- **READ UNCOMMITTED** = nessun 2PL, i dati vengono letti senza lock (lettura sporche), è consentito solo per transazioni read-only

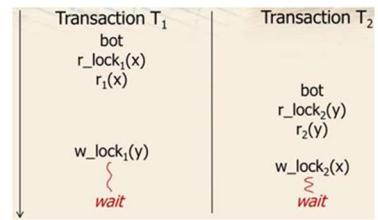
Il livello di isolamento può essere settato con:

SET TRANSACTION [ISOLATION LEVEL <level>] [READ ONLY] [READWRITE]

Può essere **ridotto solo per operazioni di lettura**, le operazioni di write vengono sempre fatte in almeno strict 2PL con lock esclusivo.

Parliamo quindi di **DEADLOCK** (lascia 2 transazioni bloccate in attesa che l'altra faccia l'unlock sulla risorsa richiesta). La soluzione più usata è l'uso di un **timeout** dopo cui la transazione in attesa riceverà una risposta negativa e farà rollback (un timeout lungo genera deadlock lunghi, un timeout corto sovraccarica il sistema). Altre soluzioni sono:

- **Pessimistic 2PL** = tutti i lock vengono presi prima dell'inizio della transazione (non realizzabile, rallenta troppo il sistema)
- **Timestamp** = solo le transazioni più giovani possono rimanere in attesa che la transazione più vecchia venga abortita



La rilevazione dei deadlock è basata sul **wait graph** dove ogni freccia rappresenta uno stato di attesa tra 2 transazioni (un ciclo nel grafo è un deadlock) [un grafo di questo tipo è però complicato e viene usato solo in DBMS distribuiti]

- **GESTIONE DELL'AFFIDABILITÀ** (Reliability Manager) → si occupa di far rispettare le proprietà ACID di atomicità e durabilità. Usa un **file di log** (con tutte le attività svolte dal DBMS in ordine cronologico) memorizzato in una **memoria stabile** (probabilità di failure ~ 0 [non possiamo perdere il file di log]). Le operazioni (come già detto) vengono scritte nel log **sequenzialmente in ordine cronologico** e possono essere:

- **Record transazionali:**

- **Begin** $B(T)$
- **Commit** $C(T)$
- **Abort** $A(T)$
- **Insert** $I(T, O, AS)$
- **Delete** $D(T, O, BS)$
- **Update** $U(T, O, BS, AS)$

Dove T = TID, O = RID (id del record), **BS** (before state), **AS** (after state)

⚠ Se faccio:

- **undo** di:

- Insert O = eliminare O
- Update O = scrivere il **BS** di O
- Delete O = scrivere il **BS** di O

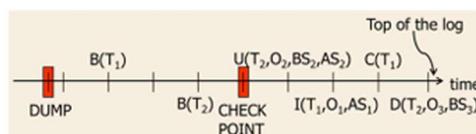
- **redo** di:

- Insert O = scrivere l'**AS** di O
- Update O = scrivere l'**AS** di O
- Delete O = eliminare O

Queste undo e redo sono utili in caso di crash e devono essere **idempotenti** (posso ripetere le undo e redo quante volte voglio ma devo ottenere lo stesso stato finale)

- **Record di sistema:**

- **Checkpoint CK(T_1, T_2, \dots, T_n)** = specifica l'insieme delle transazioni attive (usando i TID) e memorizza in maniera sincrona sul file di log le pagine relative alle transazioni conclusive [ovvero il record di checkpoint] (durante queste operazioni nessuno può fare commit)



- **Dump** = crea una copia completa della base dati memorizzandola in una memoria stabile (fatta quando i sistemi sono offline). Alla fine viene scritto un record di dump nel file di log, con il timestamp e il dispositivo di dump usato

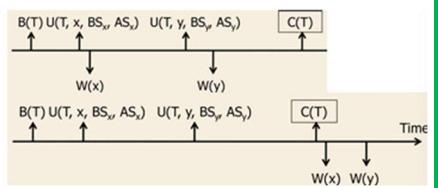
Ci sono 2 regole per la scrittura dei log:

- A. **Write Ahead Log (WAL)** = il **BS** dei dati in un record di log deve essere scritto prima che il dato venga scritto nel DB (ciò consente operazioni di undo su dati parzialmente scritti nel disco)
- B. **Commit precedence** = l'**AS** dei dati in un record di log deve essere scritto prima dell'esecuzione dell'operazione di commit (ciò consente operazioni di redo per transazioni che hanno già fatto il commit, ma che non sono state scritte nel DB)

Nell'effettivo, **BS e AS vengono scritti insieme** in quanto decidiamo di scrivere il log prima del **record** nel DB e prima del **commit** (in maniera sincrona). Posso scrivere il log in maniera asincrona per **abort/rollback**. Perciò il **record di commit** relativo ad una transazione rappresenta il limite prima di cui la transazione deve essere sfatta in caso di failure, dopo cui la transazione deve essere rieseguita in caso di failure.

Ci sono diversi **protocolli** per la scrittura del log e del DB:

- Tutte le write del DB su disco vengono fatte prima del commit (non richiede il redo delle transazioni committate)
- Tutte le write vengono fatte dopo il commit (non richiede undo di transazioni non committate)
- Le write avvengono sia prima sia dopo il commit (richiede sia undo sia redo [usato nei sistemi reali])

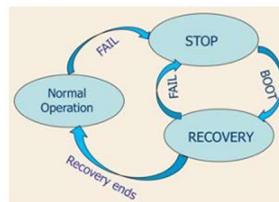


⚠️ Usare un protocollo robusto nella scrittura dei log è costoso, ma assicura le proprietà ACID e ottimizza la scrittura dei log (compatti, parallelismo e commit di gruppi di transazioni)

Parliamo del **RIPRISTINO**. Ci sono 2 tipi di failure:

- **System failure** = guasto sw o di alimentazione; perdita di memoria principale (ma non del disco)
- **Media failure** = guasto del dispositivo che gestisce la memoria secondaria; perdita contenuto su disco, ma non del file di log

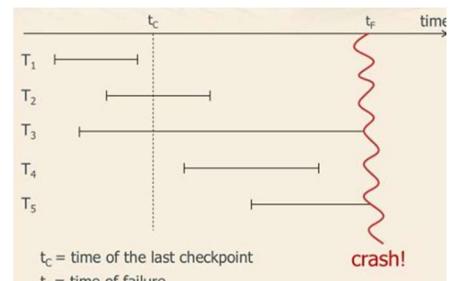
Quando avviene un guasto il sistema di ferma (**FAIL-STOP**).



Quando il sistema riparte entra in **recovery**; 2 tipi:

- ❖ **WARM RESTART** = usato per system failure. Se il sistema si trova nella situazione rappresentata in figura, notiamo che al momento del guasto:

- L'unica transazione **completata** prima del checkpoint è T_1 (non richiesto il recovery)
- Le transazioni **committate** (ma di cui non ne è certa la scrittura) sono T_2 e T_4 (richiesta redo)
- Le transazioni **attive** sono T_3 e T_5 (non hanno committato, quindi richiesta undo)



Come funziona quindi il warm restart?

1. **Lettura all'indietro del log fino all'ultimo checkpoint** (non obbligatorio, ma se non ci fosse dovrei rileggere fino all'ultimo dump)
2. **Individuo delle transazioni su cui fare undo/redo**
 - a. Al checkpoint creo le liste
 - i. **UNDO** (transazioni attive al checkpoint)
 - ii. **REDO** (vuota)
 - b. Lettura in avanti del log
 - i. In UNDO metto le transazioni per cui viene trovato il begin record
 - ii. In REDO sposto da UNDO le transazioni per cui viene trovato il commit record
 - iii. Le transazioni che hanno fatto ABORT restano in UNDO

Alla fine di questa fase avrò 2 liste delle transazioni su cui fare UNDO/REDO
3. **Ripristino dei dati**
 - a. Il log viene letto all'indietro disfacendo le operazioni delle transazioni nella lista di UNDO fino al begin record della transazione più vecchia (anche se è precedente al checkpoint) nella lista di UNDO
 - b. Il log viene riletto in avanti e le operazioni svolte dalle transazioni nella lista REDO vengono applicate al DB

- ❖ **COLD RESTART** = usato quando viene danneggiato il DB su disco. Gli step:

1. Accesso all'ultimo dump per ripristinare la porzione di memoria danneggiata

2. Partendo dall'ultimo record di dump, il log viene letto in avanti e vengono rifatte tutte le operazioni sul DB, con le operazioni di commit/rollback
3. Viene fatto un warm restart per rivedere le operazioni successive all'ultimo checkpoint, eseguite dal dump facendo UNDO/REDO quando serve

Alternativamente si può fare il REDO solo delle operazioni relative a transazioni committate. Ciò richiederebbe 2 letture del log: 1 per la creazione della lista REDO (con le transazioni committate), 1 per l'esecuzione del REDO rispetto a questa lista

Quando la recovery termina, il sistema torna disponibile.

11) DBMS DISTRIBUITI

I **vantaggi** di un'architettura distribuita (dati e computing distribuiti su più macchine) sono > performance, > availability, > affidabilità.

Ci sono diversi **tipi di architetture distribuite**:

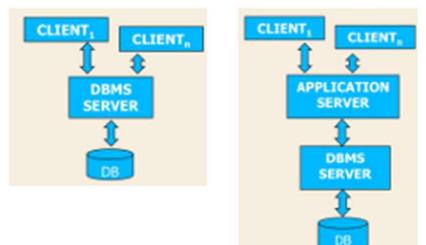
- **Client/Server** → server gestisce il DB, mentre il client gestisce l'interfaccia utente
- **Distributed database system** → ci sono diversi server DBMS installati su diversi nodi della rete che agiscono autonomamente (i DBMS cooperano ma le proprietà ACID sono difficili da garantire)
- **Data Replication** → una replica dei dati viene memorizzata in un nodo di rete diverso (il server di replica gestisce autonomamente l'update della replica) [come le DW rispetto ai DB]
- **Architetture parallele** → migliorano le performance e possono essere multicore o cluster di CPU
- **Data Warehouse (DW)** → server specializzati per il supporto decisionale che eseguono operazioni OLAP

I DBMS distribuiti hanno **proprietà**:

- **Portabilità** = spostare un programma da un sistema ad un altro (garantita da SQL standard)
- **Interoperabilità** = cooperare su specifici task sfruttando protocolli come OBDC e X-Open-DTP

Parliamo delle **ARCHITETTURE CLIENT/SERVER**; possono essere di 2 tipi:

- **2-Tier** = composto da:
 - **DBMS server** (fornisce accesso ai dati)
 - **Thick client** (contiene anche logica applicativa)
- **3-Tier** = composto da:
 - **DBMS server** (fornisce accesso ai dati)
 - **Server applicativo** (logica applicativa e web server)
 - **Thin client** (browser)



Ha 2 modalità di esecuzione:

- **Compile & Go** = dopo che la query viene ricevuta, il server crea il piano di esecuzione, esegue direttamente la query e restituisce il risultato
- **Compile & Store** = dopo che la query viene ricevuta, il server genera il pianon di esecuzione e lo memorizza per possibili usi futuri; poi esegue la query e restituisce il risultato (efficace con query parametriche ripetute)

Parliamo ora dei **DATABASE DISTRIBUITI**, ovvero quando una transazione (client) può dover accedere a diversi server DBMS e dove ogni nodo (server DBMS) gestisce i suoi nodi in modo autonomo (**local autonomy**). **Vantaggi**:

- ✓ Localizzazione appropriata dei dati e delle applicazioni
- ✓ Aumento dell'**availability** (potrebbero essere più richiesti i blocchi locali memorizzati nel DBMS interrogato)
- ✓ Miglioramento della **scalabilità** (architettura modulare)

Com'è il **DESIGN di database distribuiti**? Questi sistemi sfruttano la **FRAMMENTAZIONE dei dati**, ovvero data una relazione R si può avere un **Frammento**:

- **Orizzontale** = contiene un **subset di tuple** (righe) con gli stessi attributi di R; ottenuta con una selezione σ_p (p = predicato di partizionamento), i frammenti non sono sovrapposti. La ricostruzione di R si fa con l'**unione di tutti i frammenti**

- **Verticale** = contiene un **subset di colonne**; ottenuta con una proiezione π_x (x = subset di attributi di R) a cui si aggiunge la chiave primaria per ricostruire R. Vengono incluse tutte le tuple (righe) e i frammenti si sovrappongono sulla chiave primaria. La ricostruzione di R si fa con una **join sulla chiave primaria**

I frammenti devono avere:

- **Completezza** = ogni unità presente in R è presente in almeno un frammento R_i
- **Correttezza** = il contenuto di R può essere ricostruito partendo dai suoi frammenti

I DBMS distribuiti usano la **frammentazione per distribuire i dati fra i diversi nodi della rete**, in quanto ogni frammento viene salvato in file diversi, memorizzati su server diversi. Lo **schema di allocazione** dice come i frammenti sono distribuiti nei nodi:

- **Mapping non ridonante** = ogni frammento è memorizzato in 1 nodo
- **Mapping ridondante** = frammenti replicati su server diversi (> availability, ma gestione più complessa per aggiornare le repliche)

Cosa si vede di ciò in **SQL**? I **livelli di trasparenza** descrivono il livello di conoscenza sulla distribuzione dei dati:

- **Fragmentation transparency** = le applicazioni conoscono l'esistenza delle tabelle ma **non ne conoscono la frammentazione**
- **Allocation transparency** = le applicazioni conoscono l'esistenza dei frammenti, ma **non dove sono allocati** né se sono replicati
- **Language transparency** = le applicazioni conoscono **sia i frammenti che la loro allocazione**; il programmatore deve specificare sia il frammento sia la sua allocazione

```

SELECT SName
FROM S
WHERE S#= :CODE
SELECT SName
FROM S1
WHERE S# = :CODE
IF(NOT FOUND)
    SELECT SName
    FROM S2
    WHERE S# = :CODE

SELECT SName
FROM S1@xxx.torino.it
WHERE S# = :CODE
IF(NOT FOUND)
    SELECT SName
    FROM S2@xxx.roma1.it
    WHERE S# = :CODE
  
```

Una **TRANSAZIONE** si può classificare in:

- ☆ **Richiesta remota** = richiesta di lettura (quindi solo operazioni di SELECT) ad 1 solo server remoto
- ☆ **Transazione remota** = esecuzione di qualsiasi operazione su 1 solo server remoto
- ☆ **Transazione distribuita** = esecuzione di qualsiasi operazione, in cui ogni operazione viene fatta su 1 solo server; necessita il rispetto dell'atomicità globale del DB (protocollo 2-phase-commit)
- ☆ **Richiesta distribuita** = ogni operazione può essere riferita a dati memorizzati su server diversi (solo qui viene usata la fragmentation transparency)

Vediamo quindi le **TECNOLOGIE PER I DBMS DISTRIBUITI**. Partiamo riprendendo le **proprietà ACID** qui:

- **Atomicità** = richiede tecniche distribuite come il 2-phase-commit
- **Consistenza** = i vincoli vengono imposti solo localmente
- **Isolamento** = richiede 2PL e 2-phase-commit
- **Durability** = richiede l'estensione delle procedure locali per gestire l'atomicità in presenza di failure

⚠ L'**ottimizzazione delle query** distribuite viene fatta dal DBMS che riceve la query: la partiziona in subquery (ognuna indirizzata ad 1 solo DBMS), sceglie il piano di esecuzione e coordina le operazioni e lo scambio di informazioni tra i nodi interessati

La gestione dell'atomicità è basata sul fatto che tutti i nodi partecipanti ad una transazione distribuita devono prendere le **stesse decisioni**, che vengono coordinate dal protocollo **2-PHASE-COMMIT**: questo consente di coordinare il **commit** (la conclusione) di una transazione distribuita, la quale ha 1 **TM** (**Transaction Manager**) e diversi **RM** (**Resource Managers**) che partecipano alla transazione e hanno un log personale.

I **record in un TM** possono essere di:

- **Prepare** = contiene la lista (NodeID + ProcessID) dei RM che partecipano alla transazione; indica che la transazione è pronta a fare commit
- **Global commit/abort** = la decisione finale sull'esito della transazione
- **Complete** = indica la fine del protocollo

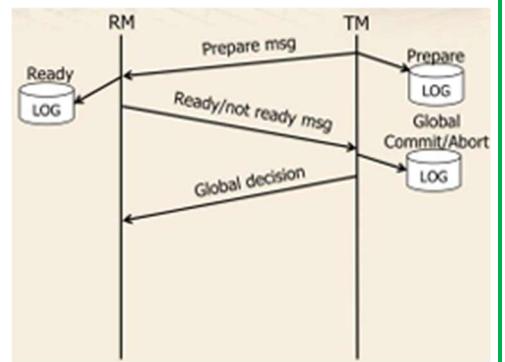
I **record in un RM** possono essere di:

- **Ready** = l'RM è pronto ad eseguire un'operazione di commit: questa decisione non può cambiare e da ora l'RM perde l'autonomia sulla transazione corrente (questo stato si può salvare nel log e può essere inoltrato al TM solo se il nodo si trova in uno stato affidabile [WAL, commit precedence e le risorse devono essere bloccate])
- **Commit/Abort**

Il protocollo 2-phase-commit ha **2 fasi**:

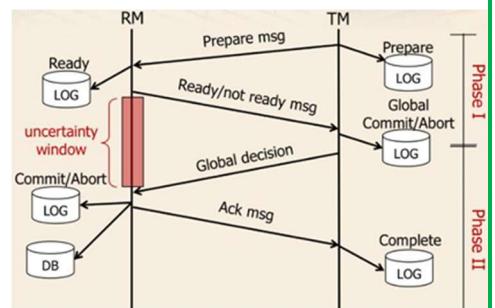
1. FASE 1:

- Il **TM** scrive il **Prepare** record sul suo log, invia il Prepare a tutti gli RM partecipanti e setta un **timeout** dove rimane in attesa di una risposta
- Dopo avere ricevuto il Prepare, gli **RM** se sono:
 - Stato **affidabile** = scrivono il record di **Ready** nel log e inviano il Ready al TM
 - Stato **non affidabile** = inviano un messaggio di **Not Ready** al TM, terminano il protocollo e fanno un rollback locale
 - RM **spento** = non viene inviata alcuna risposta
- Il **TM** memorizza tutti i messaggi in ingresso:
 - Se ha ricevuto Ready da tutti gli RM, scrive sul log il record di **Global Commit**
 - Se riceve almeno 1 Not Ready o il timeout termina, scrive il record di **Global Abort**



2. FASE 2:

- Il **TM** invia la **decisione globale** a tutti gli RM e setta un altro **timeout** per la ricezione delle risposte
- Dopo avere ricevuto la decisione globale, gli **RM** eseguono il commit/abort ricevuto e inviano un ack al TM
- Il **TM** memorizza tutti i messaggi in ingresso:
 - Se riceve tutti gli ack, scrive il record di **Complete** sul log
 - Se mancano alcuni ack allo scadere del timeout, la decisione globale viene reinviata e viene resettato il timeout (ciò si fa fino a che non si ricevono tutti gli ack)



⚠ Ogni RM ha una **finestra di incertezza** (nella figura “uncertainty window”) che parte dal momento in cui viene mandato il Ready e finisce alla ricezione della decisione globale. In questo tempo, le risorse locali dell'RM sono **bloccate** (quindi **deve essere il < possibile**).

In caso di **FAILURE** di:

- **1 RM** → la procedura di warm restart viene modificata aggiungendo un nuovo caso: se l'ultimo record nel log (relativo alla transazione *T*) è **Ready**, allora non è nota la decisione globale relativa a *T*.
In fase di recovery, viene creata la **Ready List** con tutti gli id delle transazioni in Ready; al restart, per ogni transazione della lista, viene richiesta la decisione globale al TM (“**remote recovery request**”)
- **TM** → in fase di recovery:
 - se l'ultimo record nel log del TM è **Prepare**, viene inviato il global abort
 - se l'ultimo record è la **decisione globale**, viene ripetuta la fase 2
- **Rete** → se avviene in:
 - fase 1, i messaggi di **Prepare** o **Ready** non vengono ricevuti, viene quindi fatto un **Global Abort**
 - fase 2, la **decisione globale** o gli **ack** non vengono ricevuti, viene ripetuta la fase 2 dall'inizio

⚠ **X-Open-DTP** è un protocollo per la coordinazione di transazioni distribuite basato su 1 client, 1 TM e più RM; implementa il 2-phase-commit e garantisce interoperabilità delle transazioni distribuite su DBMS eterogenei

12) ElasticSearch

ElasticSearch (ES) è un motore di ricerca/analisi distribuito, scalabile ed efficiente costruito su “**Lucene**”, che consente diverse tecniche di ricerca e esplorazione dei dati: **ricerca full-text**, **ricerca strutturata** e **analitiche** (usato da GitHub, Wikipedia, StackOverflow...). Le ricerche sono rappresentate da un **documento json** (può anche contenere dati complessi come date, dati geografici, testo, array...).

I suoi punti di forza sono:

- ✓ Scaling orizzontale
- ✓ **RESTful API** (consentono interfacciamento con qualsiasi linguaggio di programmazione)
- ✓ **JSON-based** (è sia machine sia human-friendly)
- ✓ **Replicazione** (ogni modifica viene storizzata nei log, memorizzati su tanti nodi per evitare di perdere dati)

La **COLONA** di SQL è chiamata “**field**” (possono essere di diversi tipi e possono contenere valori multipli dello stesso tipo); la **RIGA** di SQL è chiamata “**document**” (ha formato flessibile ed è identificato univocamente dalla coppia [Index, ID] con Index = dove viene memorizzato il documento, ID = id del documento). La **TABELLA** di SQL è chiamata “**index**”; gli index vengono raggruppati in **cluster**, che rappresentano il **DB** di SQL.

⚠ Se usato come **sostantivo**, **index** = tabella (e non come la **struttura fisica accessoria** vista nella progettazione fisica che accelera le prestazioni [qui chiamato **inverted index**] [**in ElasticSearch sono ricercabili solo i field che sono indicizzati con questa struttura**]); se usato come **verbo**, **index** = inserimento di un documento in un index (tabella)

La ricerca (searching) può essere di **3 tipi** (sopra citati):

- **Full text** = trova tutti i documenti che matchano con le keywords e li ordina secondo la loro “rilevanza”
- Ricerca **strutturata su field specifici**
- **Combinazione dei 2**

I concetti principali sono:

- **Mapping** → ElasticSearch genera dinamicamente un mapping fra i dati nella query e quelli nei documenti anche se questi sono di tipi diversi (es. riconosce un “date” nella query che di base è una stringa)
- **Analysis** → la stringa di query viene processata e standardizzata per poter essere usata nella ricerca
- **Query DSL** → linguaggio usato in ElasticSearch (Domain Specific Language)

Se abbiamo **dati tradizionali** (**int**, **float**, **date**) il valore nel documento deve matchare esattamente il valore indicato nella query (in questo caso non serve il concetto di rilevanza). Se invece abbiamo **dati testuali** (**full text**) dobbiamo considerare la **RILEVANZA** dei documenti rispetto alla query (bisogna infatti riconoscere il senso della parola nella fase e quindi riconoscere abbreviazioni, singolare/plurale, coniugazioni del verbo, sinonimi, ordine [di solito si usano plugin apposta per questo]).

Nell'**indexing full text**, ES crea un inverted index, per ogni campo testuale, che contiene la lista di tutte le parole che compaiono in qualsiasi documento della collezione e, per ogni parola, la lista di tutti i documenti in cui è presente.

Prima di generare la lista di parole da inserire nell'inverted index, l'**Analyzer** dell'ES fa analisi:

- **Tokenizzazione** = testo diviso in termini individuali mediante Tokenizers
- **Normalizzazione** = termini normalizzati in una forma standard (portati alla radice, al singolare, al sinonimo...)

Per fare ciò l'Analyzer ha diverse funzionalità:

- **Filtro dei caratteri** (pulisce la stringa prima della tokenizzazione)
- **Tokenizers** = dividono la stringa in parole singole (considerando la punteggiatura e gli spazi)
- **Filtri dei token** = operano sui termini singoli (modificano, aggiungono, rimuovono termini)

⚠ Un **filter** è usato per la ricerca del tipo esatto, mentre le **query** sono usate per la ricerca full-text (e danno anche informazioni sulla rilevanza del match). Inoltre i filter, essendo più efficienti, vengono usati anche per ridurre il numero di documenti che devono poi essere esaminati dalla query

Le query ES vengono espresse in **Query DSL scritte in formato JSON** nel body di una richiesta HTTP; vediamo degli **ESEMPI**:

Esempio (match query): Cercare tutti i documenti dell'indice department che hanno un field "name" contenente il valore "John".

Esempio 2 (query composta da diversi criteri di matching):

bool: serve a specificare che la query è composta
should: consiste nella condizione OR
must: corrisponde con la AND
must_not: specifica la NOT

```
POST departments/_search
{
  "query": {
    "match": { "name": "John" }
  }
}
```

```
POST departments/_search
{
  "query": {
    "bool": {
      "should": [
        {"match": { "name": "John" }},
        {"match": { "name": "Mark" }}
      ],
      "minimum_should_match": 1,
      "must": [
        {"match": { "title": "developer" }}
      ],
      "must_not": [
        {"match": { "lastname": "Smith" }}
      ]
    }
  }
}
```

La match query può essere usata sia per la ricerca del tipo esatto (in cui viene ritornato sempre un punteggio di rilevanza `_score = 1`) sia per la ricerca full-text (prima di essere eseguita, la query viene analizzata dall'analyser e le viene assegnato il punteggio di rilevanza `_score`).

⚠️ In una **bool** query, viene combinato lo `_score` di ogni must e should che matchano; si possono anche specificare diversi indici in cui fare la ricerca (separandoli con la virgola)

Con ES si può anche **modificare i dati**:

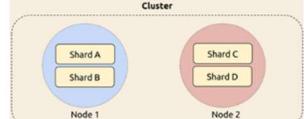
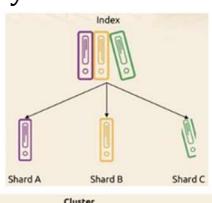
- **INSERT** → inserimento di un nuovo documento fatto con una POST con il nome dell'indice e l'id (opzionale) nell'URI e il documento nel body (es. `POST /index_name/<id> { JSON document }`)
- **UPDATE** → i documenti in ES sono **immutabili**, quindi per essere modificato deve essere **reindicizzato**: durante l'update, il documento viene recuperato, copiato, eliminato e reindicizzato. Per fare l'update, si usa una PUT request in cui vengono specificati il nome dell'indice, l'ID del documento nell'URI e i field da cambiare (con il nuovo valore) nel body (es. `PUT /index_name/123/_update { "color" : "red", }`)
- **DELETE** → la delete viene fatta usando una DELETE request in cui vengono specificati il nome dell'indice e l'ID del documento nell'URI (eliminazione dei documenti fatta in **background**) (es. `DELETE index_name/id`)

In ES la rilevanza di uno `_score` si rappresenta con un float calcolato per ogni documento che matcha la query (la rilevanza è vista in modo decrescente). **Come si calcola lo `_score`?** Vediamo il processo di **SCORING**:

1. Si **CALCOLA LA SIMILARITÀ** fra la query e ogni documento, calcolandone lo `_score`:
 - a. Vengono selezionati i documenti che **matchano** la query
 - b. Viene valutata l'**importanza di ogni termine** rispetto al documento e alla collezione:
 - i. Importanza del termine valutata con **TF/IDF** (**TermFrequency/InverseDocumentFrequency**):
 1. **Term Frequency** = frequenza del termine all'interno del singolo documento
$$Tf(t \text{ in } d) = \sqrt{\text{frequency_of_t_in_d}}$$
 2. **Inverse Document Frequency** = frequenza del termine nella collezione
$$Idf(t) = 1 + \log(\text{numDocsInIndex}/(\text{numDocContainingTerm} + 1))$$
 3. **Field-length norm** = normalizzazione rispetto alla lunghezza del field (più è lungo il field, < probabilità che le parole in esso siano rilevanti)
$$\text{norm}(d) = 1/\sqrt{\text{numTermsInField}}$$
- ii. Il documento e la query vengono rappresentati in forma vettoriale (**Vector Space Model [VSM]**) dove la dimensione rappresenta il numero di termini e ogni elemento contiene il peso di ogni termine calcolato con TD/IDF
- c. Viene valutata la similarità della rappresentazione vettoriale (VSM) della query e del documento, usando la **cosine similarity** (considera l'angolo fra documento e la sua query per calcolare similarità)
2. Vengono selezionati gli **hits** (10 documenti più rilevanti, `_score >`)
3. [Opzionale] Viene fatto un **rescoring** dei documenti più rilevanti che hanno matchato la query

Per la **scalabilità orizzontale** è importante il concetto di **SHARDING**, ovvero suddividere un indice in piccole parti (shard) che contengono diversi documenti. Quando i dati vengono scritti in una shard devono essere memorizzati su disco in un segmento Lucene (ogni secondo) prima di diventare disponibili alle query.

Un **cluster** è l'insieme di macchine dove sono memorizzate le diverse shard. Lo sharding consente scalabilità e distribuzione delle operazioni sui nodi presenti nel cluster, migliorando le performance grazie alla **parallelizzazione**, migliorando l'**availability**.



ES fa un **controllo della concorrenza ottimistico** (assume che i conflitti siano improbabili) **non applicando locking**. Se i dati vengono modificati tra lettura e scrittura, l'update fallisce.

È probabile che ci siano diverse repliche di una shard nel cluster (più sicuro), quindi vanno propagati gli update alle varie copie: per fare ciò, ES usa il **versioning** mediante **_version** che evita la sovrascrittura da parte di una vecchia versione del documento (le API di modifica o cancellazione accettano anche **_version** come parametro).

13) NoSQL & MongoDB

Le caratteristiche principali di un database **NoSQL (Not Only SQL)** sono **assenza di join, assenza di uno schema** (no tabella/schema fisso), **scalabilità orizzontale**, linguaggio custom e adatto per **dati complessi**.

Ci sono **4 tipi di DB NoSQL**:

- **Key-value** → matcha una key ad un value (semplice, veloce)
- **Column-oriented** → ogni colonna è un attributo (righe ricostruibili con i singoli valori delle colonne); le coppie key-value sono memorizzate e recuperate con chiave su sistemi paralleli (come gli indici)
- **Graph** → nodi e archi (teoria dei grafi) [es. amicizie su Facebook]
- **Document** → documenti che rappresentano liste di attributi key-value (valori possono essere altri documenti) [struttura gerarchica e auto-descrittiva] (es. **MongoDB**, **CouchDB** e RavenDB)

CouchDB è **document-oriented**, interrogabile e indicizzabile con **MapReduce**; offre **replicazione** (salvare gli stessi dati in posti diversi [evitare single point of failure]) incrementale con identificazione e risoluzione dei conflitti tra i nodi. È scritto in Erlang (sistemi distribuiti, scalabile) e offre API JSON RESTful che permettono **accessibilità** da qualsiasi ambiente che supporti **richieste HTTP**.

MapReduce sposta la computazione ai nodi distribuiti alleggerendo il carico del nodo che sta eseguendo la query. È fatta di 2 funzioni:

- **Map (SELECT+WHERE)** = processo i singoli record per avere coppie chiave-valore; viene chiamata per ogni documento (indipendenza da informazioni esterne)
- **Reduce (GROUP BY)** = partendo dalle coppie key-value della Map, genera una coppia chiave-valore per ogni chiave diversa (aggregazione) [re-reduce = reduce ricorsiva su chiave composta da n° di attributi via via <]

DB	Map		Reduce		Rereduce			
	doc.id	Key	Value	Key	Value	Key	Value	
Id: 1 Exam: Database Student: s123456 AYear: 2015-16 Date: 31-01-2016 Mark>29 CFU=8	Id: 8 Exam: Database Student: s987654 AYear: 2014-15 Date: 28-06-2015 Mark>25 CFU=8	6	Bioinformatics, 2015-16	30	[Bioinformatics, 2015-16]	30	Bioinformatics	30
Id: 6 Exam: Bioinformatics Student: s123456 AYear: 2015-16 Date: 18-09-2016 Mark>30 CFU=6	Id: 4 Exam: Database Student: s1234321 AYear: 2014-15 Date: 26-07-2015 Mark>26 CFU=8	2	Computer architectures, 2015-16	24	[Computer architectures, 2015-16]	25.5	Computer architectures	25.5
Id: 5 Exam: Software engineering Student: s123456 AYear: 2014-15 Date: 14-02-2015 Mark>21 CFU=8	Id: 7 Exam: Software engineering Student: s654321 AYear: 2015-16 Date: 28-06-2016 Marks>18 CFU=8	3	Computer architectures, 2015-16	27				
Id: 3 Exam: Computer architectures Student: s654321 AYear: 2015-16 Date: 26-01-2016 Mark>27 CFU=10	Id: 2 Exam: Computer architectures Student: s123456 AYear: 2015-16 Date: 03-07-2015 Mark>24 CFU=10	4	Database, 2014-1015	26	[Database, 2014-15]	25.5	Database	27.25
		8	Database, 2014-15	25				
		1	Database, 2015-16	29	[Database, 2015-16]	29		
		5	Software engineering, 2014-15	21	[Software engineering, 2014-15]	21		
		7	Software engineering, 2015-16	18	[Software engineering, 2015-16]	18	Software engineering	19.5

⚠ Per scrivere una query in CouchDB si usa la view (prodotta da Map&Reduce) che contiene l'id del documento come key e l'intero documento come value

La **Replicazione** può essere:

- **Master-Slave** → server master fa write, update e insert; servers slave fanno letture (master = single point of failure; slaves = scalabili) [CouchDB supporta replicazione Master-Master]
- **Sincrono** → prima di fare un commit, il master aspetta che tutti gli slave facciano commit [poco ottimizzato]
- **Asincrono** → master fa commit localmente senza aspettare gli slaves. Se il master ha una failure nella fase di aggiornamento degli slave, alcuni dati sono persi [veloce ma meno affidabile]

Un database distribuito è composto da macchine autonome che lavorano insieme per gestire un dataset comune. Le 3 criticità dei DB distribuiti sono **consistency** [C] (tutti i DB forniscono gli stessi dati all'applicazione), **availability** [A] (1 failure non blocca il funzionamento degli altri nodi) e **partition tolerance** [P] [3 criticità non risolvibili insieme (**Teorema CAP**)]:

- Consistenza locale (CA senza P) = caso centralizzato
- Transaction locking (CP senza A) = caso relazionale
- Best effort/Optimistic locking (AP senza C) = caso non relazionale

⚠ Nella pratica, C, A e P variano continuamente (non si possono avere al 100% quando lo si vuole)

Nei DB non relazionali si considerano le proprietà **BASE** (piuttosto che le ACID) [es. usate nei DNS]:

- **Basically available** (B A)
- **Soft state** (S) = lo stato del sistema può variare anche senza input (a causa dell'Eventual consistency)
- **Eventual consistency** (E) = il sistema inizia a diventare consistente quando smette di ricevere input

⚠ Hadoop gestisce molti dati distribuiti in modo decentralizzato e sfrutta MapReduce

Per “**design recipe**” si intende che nei DB NoSQL è conveniente rappresentare tutte le transazioni tramite documenti contenenti mittente, destinatario e quantità (infatti se gestissimo la transazione come 2 operazioni separate, potremmo avere delle inconsistenze)

Parliamo ora di **MongoDB**, un database NoSQL document-based open source, flessibile e veloce; facendo una trasposizione da SQL, abbiamo:

- Tabella → **Collection**
- Record → **Document** (documenti **BSON** = JSON binari)
- Column → **Field**

Ogni documento ha una chiave primaria autogenerata “**_id**”.

```
{  
  _id: ObjectId("5099803df3f4948bd2f98391"),  
  name: { first: "Alan", last: "Turing" },  
  birth: new Date('Jun 23, 1912'),  
  death: new Date('Jun 07, 1954'),  
  contribs: [ "Turing machine", "Turing test", "Turingery" ],  
  views : NumberLong(1250000)  
}
```

Ogni istanza di MongoDB può contenere diversi DB composti da diverse collezioni; ogni collezione contiene una serie di documenti (che possono avere struttura e field diversi). Ci sono diversi comandi per la **gestione** dei DB:

- **show databases** = mostra i DB aperti
- **use databaseName** = usa il DB selezionato o lo crea se non esiste
- **db.dropDatabase** = elimina DB corrente
- **db.createCollection(collectionName, options)** = crea collezione
- **show collections** = mostra tutte le collezioni presenti nel DB
- **db.collectionName.drop()** = elimina la collezione

Operazioni:

- **Create**
 - **db.collectionName.insertOne(documento con coppie chiave-valore)** [i valori delle coppie possono essere sia array che documenti]
 - **db.collectionName.insertMany([lista di documenti separati da virgola])**
- **Delete**
 - **db.collectionName.deleteMany(documento con la condizione da cercare)** [esempio: db.people.deleteMany({status: "D"}) elimina le persone con status == D]
- **Read (Query)**
 - **db.collectionName.find({condizioni}, {campi di interesse})** [campi di interesse è una lista di tuple con chiave il nome del campo e come valore 1 per includerlo o 0 per escluderlo] [esempio: db.people.find({status: "A"}, {user_id:1, status:1, _id:0}) voglio l'user_id e lo status delle persone che hanno status "A"]
 - **db.collectionName.findOne({condizioni}, {campi di interesse})** [restituisce solo il 1° trovato]

Per eseguire le operazioni di **join** ci sono 2 approcci:

- ❖ **Object_ID** (“Manual Reference”) = salvare l’_id di un documento, in un campo di un altro documento
- ❖ **DBRef** = standard per rappresentare un documento che indica una relazione; ha 3 campi:
 - **\$ref** = id della collezione del documento che vogliamo collegare

- **\$id** = id del documento
- **\$db** = nome del DB in cui si trova il documento (se non c'è, collezione si trova nel DB corrente)

Gli operatori di comparazione sono **\$eq**, **\$gt**, **\$gte**, **\$in**, **\$nin**, **\$lt**, **\$lte**, **\$ne** (!=, oppure i documenti che non hanno proprio il campo specificato nella comparazione), **\$exists** (solo i documenti che hanno il campo specificato), **\$type** (solo i documenti che hanno il tipo specificato sul campo specificato).

⚠ AND si rappresenta con “,” (ovvero condizione1, condizione2...) mentre OR si rappresenta con **\$or**

La find restituisce un cursore che può essere usato per applicare filtri e funzioni al risultato come:

- ✓ cursor.sort() = ordina il risultato in base all'attributo selezionato (1 = ascendente, -1 = descendente)
- ✓ cursor.count() = # di documenti che hanno fatto match

- **Update**

- **db.collectionName.updateOne(filter, update, options)** [filter = quali documenti aggiornare; update = quali campi del documento devo aggiornare]
- **db.collectionName.updateMany(filter, update, options)**

Esempio: vengono modificati size.uom, status e lastModified in tutti i prodotti con una quantità minore di 50.

```
db.inventory.updateMany(
  { "qty": { $lt: 50 } },
  {
    $set: { "size.uom": "in", status: "P" },
    $currentDate: { lastModified: true }
  }
)
```

- **Aggregazioni** → **db.collectionName.aggregate({set-of-stages})** dove gli stage sono molti tra cui:

- **\$match** = filtro (predicato) usabile anche in uno stadio intermedio; se usato dopo una **\$group** si comporta come un HAVING
- **\$group** = come GROUP BY

- **Indici** → normalmente MongoDB fa uno scan di ogni documento nella collezione quando fa Read; gli indici li abbiamo già visti nella progettazione fisica; qui sono simili, ovvero memorizzano valori ordinati di un field e il collegamento al documento originale (tipo unclustered) [MongoDB crea di **default** un indice su `_id` appena creo la collezione]. Un nuovo indice si crea con **db.collectionName.createIndex(indexKeys, options)** dove le options possono includere name (nomeIndice) o unique (indica se accetta inserimento di documenti con chiavi duplicate o no).

Si possono creare indici a campo singolo, composti, multi-chiave, geospaziale, di testo e hashed (non supporta range)