

Appunti Database

Brendon Mendicino

January 10, 2023

Contents

1	Introduzione	4
2	Data Warehouse	4
3	Analisi	5
3.1	Finestra di calcolo	6
3.2	Sintassi ORACLE	8
3.3	Esercizi	9
4	Viste materializzate	12
4.1	Documentazione Oracle	12
5	Alimentazione dei Data Warehouse	15
5.1	Estrazione	15
5.2	Pulitura	15
5.3	Trasformazione	15
5.4	Caricamento	16
6	Data Lake	18
7	Data processing	19
7.1	Data preprocessing	20
7.2	Similarità	22
7.3	Correlazione	23
8	Regole di associazione	24
8.1	Algoritmi Di Estrazione Di Itemsets Frequenti	25
8.1.1	Forza Bruta	25
8.1.2	Apriori	25
8.1.3	FP-Growth	27
8.2	Effetto delle soglie	29
9	Classificazione	30
9.1	Alberi di Decisione	30
9.2	Random Forest	32
9.3	Rule-based Classification	32
9.4	Classificazione Associativa	33
9.5	K-Nearest Neighbor (KNN)	33
9.6	Bayesian Classification	34
9.7	Support Vector Machines	34
9.8	Artificial Neural Network	35
9.9	Model evuation	36

9.10 Curva ROC	38
10 Trigger	39
11 Lab 03	42
12 Clustering	44
12.1 K-means	44
12.2 Bisecting K-means	45
13 Introduzione ai DBMS	45
13.1 Buffer Manager	47
13.2 Accesso alla Memoria	48
13.3 Progettazione Fisica	50
13.4 Ottimizzatore delle query	51
13.4.1 Ottimizzazione algebrica	52
13.4.2 Ottimizzazione basata sui costi	53
13.5 Operatori di accesso	53
13.6 Piani di esecuzione	54
13.7 Regole per valutazione di ottimizzazione	56
13.8 Concurrency Control	60
13.8.1 View Equivalence	63
13.8.2 Conflict Equivalence	63
13.8.3 2 Phase Locking	64
13.8.4 Locking Gerarchico	65
13.8.5 Predicate Locking	67
13.8.6 Locking in SQL	67
13.8.7 Deadlock	68
13.9 Reliability Manager	68
13.9.1 Warm restart	70
13.9.2 Cold Restart	71
14 DBMS Distribuiti	72
15 NoSQL	73
15.1 CouchDB	74
15.1.1 MapReduce	74
15.1.2 Replication	74
15.1.3 Distribuzione	74
15.2 MongoDB	75

1 Introduzione

KDD: Knowledge Discovery from Data

Tecniche di data mining

- Regole di associazione: usate per trovare delle relazioni frequenti all'interno del database. Ad esempio: chi compra pannolini compra anche birra, il 2% degli scontrini contegno entrambe gli oggetti, il 30% degli scontrini che contengono pannolini contengono anche birra. Grazie alle regole di associazione si possono fare dei tipi di analisi come la basket analysis, ma puo essere utile anche per le raccomandazioni.
- Classificazione: i classificatori predicono etichette discrete, esempio: nella posta elettronica alcune mail vengono segnate come spam. La classificazione definisce un modello per definire le predizioni, a volte non è sempre possibile creare dei modelli interpretabili ovvero dare una ragione per una determinata scelta.
- Clustering: gli algoritmi creano dei gruppi che raggruppano gli oggetti in esame, senza però dare delle motivazioni delle scelte effettuate.

2 Data Warehouse

Un DW è una base dati di supporto alla decisioni, che è mantenuta separatamente dalla base di dati operativa dell'azienda. I dati al suo interno sono:

- orientati ai soggetti di interesse;
- integrati e consistenti;
- dipendenti dal tempo;
- non volatili;
- utilizzati per il supporto alle decisioni aziendali;

Per la progettazione concettuale di un DW, non esiste un formalismo universale, il modello ER non è adatto ma viene invece utilizzata il modello **Dimensional Fact Model**.

Il DFM è composto da:

- Fatto: modella un insieme di eventi di interesse, che evolvono nel tempo (che può avere diversa granularità).
- Dimensioni: sono gli attribuiti di un fatto, generalmente sono categorici.

- Misure: descrive una caratteristica numerica di un fatto.

Sulle dimensioni si possono definire delle gerarchie, che definiscono di fatto una dipendenza funzionale tra gli attributi, quindi di 1 a n. Ad esempio: **data** ha un arco **mese**, una data ha uno ed un solo mese (1 a n).

I costrutti avanzati sono:

- archi opzionali;
- dimensioni opzionali;
- attributo descrittivo: sono delle informazioni utili all'utente ma su cui non verteranno le interragazioni (ad esempio non si farà mai la group by su un indirizzo);
- non-additività: non si può fare la somma sulla metrica, il motivo è che non è modellato in modo tale da fare la somma;
- Fatto: fenomeno di studio;
- Misure: attributi del fatto;
- Dimensioni: tabelle collegate al fatto;

Schema a stella:

Snowflake scheme: si esplicitano le dipendenze funzionali, questo però comporta un aumento delle operazioni di join.

Nella realtà lo snowflake è raramente utilizzato, il motivo è che il costo delle join può diventare oneroso. Un caso di utilizzo dello snowflake è quando si hanno dei dati condivisi.

Archi multipli:

Dimensioni degeneri: sono delle dimensioni con un solo attributo, questo si perché nello stato attuale non si hanno delle specifiche per quell'attributo ma nel futuro si potrebbe facilmente estendere. Un'altra soluzione potrebbe essere un push down delle dimensioni degeri nella tabella dei fatti.

Junk Dimension: si può creare una dimensione che contenga tutte le dimensioni degeneri, le informazioni sono collegate semanticamente, è anche possibile unire delle informazioni scorrelate ma non è una scelta poco corretta, una soluzione potrebbe essere avere più junk dimensions.

3 Analisi

Sfruttando solo l'SQL è molto difficile fare delle analisi su un dw, infatti volendo calcolare delle operazioni per due argomenti diversi si devono fare più query. Estendendo il SQL si può, ad esempio, effettuare più operazioni leggendo una sola volta la tabella, ed effettuando il minor numero di join possibile.

Analisi OLAP I tipi di operazione sono:

- roll up: riducendo il livello di dettaglio del dato, ovvero eliminare una o più clausole della groupby o navigare la gerarchia verso l'esterno;
- drill down: si aumenta il livello di dettaglio oppure si aggiunge una dimensione di analisi;
- slice and dice: consentono di ridurre il volume dei dati selezionando un sottogruppo dei dati di partenza;
- tabelle pivot: come viene mostrato il dato;
- ordinamento: ordinamento in base agli attributi;

Queste operazioni possono essere fatte con più o una query.

3.1 Finestra di calcolo

Una finestra di calcolo fa dei calcoli a partire da una query sottostante, la finestra ha 3 operazioni sottostanti:

- partizionamento (**partition by**): partizionamento dei dati, divide i record in gruppi a partire dall'attributo selezionato;
- ordinamento (**order by**): si definisce il criterio di ordinamento delle righe all'interno dei partizionamenti;
- finestra di aggregazione (**over**): porzione di dati, specifica per ogni riga di dato, su cui effettuare dei calcoli;

Example 3.1

Data la tabella Vendite(Città, Mese, Importo), calcolare per ogni città la media delle vendite per il mese corrente ed i due precedenti.

```

1 SELECT Città, Mese, Importo,
2      AVG(Importo) OVER (PARTITION BY Città)
3          ORDER BY Mese
4              ROWS 2 PRECEDING)
5      AS MediaMobile
6 FROM Vendite;

```

Quando la finestra è incompleta il calcolo è effettuato sulla parte presente, è possibile specificare che se la riga non è presente il risultato deve essere NULL.

Si può definire un intervallo fisico, superiore o inferiore.

```
1      ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
```

È possibile definire la tupla corrente e quella che la precedono e che la seguono

```
1 ROWS UNBOUNDED PRECEDING (o FOLLOWING)
```

Il raggruppamento fisico è specifico per quando i dati non hanno delle interruzioni.

Per definire un intervallo logico si utilizza il costrutto **range**.

```
1 SELECT Citta, Mese, Importo,
2     Importo / SUM(Importo) OVER () AS PerOverMax,
3     Importo / SUM(Importo) OVER (PARTITION BY Citta) AS PerOverCity,
4     Importo / SUM(Importo) OVER (PARTITION BY Mese) AS PerOverMonth
5 FROM Vendite
```

Se una **group by** è presente all'interno della query allora, tutte le entry che possono comparire nella finestra di calcolo sono solo quelle che compaiono nella group by.

Funzione di ranking La funzione di ranking serve a creare delle classifiche

- **rank()**: la funzione rank in presenza di più oggetti nella stessa posizione salta al prossimo record;
- **denserank()**: la funzione denserank tiene tutte righe con la stessa posizione;

...

```
1 SELECT Citta, Mese, SUM(Importo) AS TotMese,
2     RANK() OVER (PARTITION BY Citta
3                     ORDER BY SUM(Importo) DESC)
4
5 FROM Vendite, ...
6 WHERE ...
7 GROUP BY Citta, Mese
```

Estensione della group by

- **rollup**: consente di calcolare le aggregazioni su tutti i possibili gruppi, eliminando una colonna alla volta, da destra verso sinistra, esempio: calcola le vendite per: (Citta, Mese, Prodotto), (Citta, Mese), (Citta):

```
1 SELECT Citta, Mese, Prodotto, SUM(Importo) AS TotVendite
2 FROM ...
3 WHERE ...
4 GROUP BY ROLLUP (Citta, Mese, Prodotto)
5
```

- **cube**: consente di calcolare tutte le possibili combinazioni del raggruppamento;
- **grouping sets**: serve a definire degli aggregati su gruppi specifici definiti dall'utente;

3.2 Sintassi ORACLE

Raggruppamento fisico:

Example 3.2

Selezionare, separatamente per ogni città, per ogni data l'importo e la media dell'importo dei due giorni precedenti.

```

1 select citta, data, importo,
2       avg(importo) over (partition by citta
3                           order by data
4                           rows 2 preceding
5                   ) as mediaMobile
6 from vendite
7 order by citta, data;
8

```

Raggruppamento logico:

Example 3.3

```

1 select citta, data, importo,
2       avg(importo) over (PARTITION BY citta
3                           ORDER BY data
4                           RANGE BETWEEN INTERVAL '2'
5                           DAY PRECEDING AND CURRENT ROW
6                   ) as mediaUltimi3Giorni
7 from vendite
8 order by citta, data;
9

```

Example 3.4

```

1 select COD_A, sum(Q) as sommaPerArticolo,
2       rank() over (order by sum(Q) desc) as graduatoria
3 from FAP
4 group by COD_A
5

```

All'interno di oracle sono preseti delle funzionalità aggiuntive oltre alla funzione di rank, come:

ROW_NUMBER Assegno un numero progressivo ad ogni elemento in una partizione.

```

1 select tipo, peso,

```

```

2     row_number over (partition by tipo
3                     order by tipo)
4 from ...
5 where ...;
```

CUME_DIST Consente di calcolare le distribuzine cumulativa all'interno di una partizione, permette di definire un valore sulla distribuzione dei valori.

NTILE(n) Una funzione che da la possibilità di dividere le partizioni in sottogruppi

```

1 select tipo, perso,
2       ntile(3) over (partition by tipo order by peso) as ntile3peso
3 from ...
4 where ...;
```

3.3 Esercizi

Cliente(CodCliente, Cliente, Provincia, Regione)

Categoria(CodCat, Categoria)

Agente(CodAgente, Agente, Agenzia)

Tempo(CodTempo, Mese, Trimestre, Semestre, Anno)

Fatturato(CodTempo, CodCliente, CodCatArticolo, CodAgente, TotFatturato, NumArticoli, TotSconto)

1. Visualizzare per ogni categoria di articoli

- la categoria
- la quantità totale fatturata per la categoria in esame
- il fatturato totale associato alla categoria in esame
- il rank della categoria in funzione della quantità totale fatturata
- il rank della categoria in funzione del fatturato totale

```

1 select categoria, sum(numArticoli),
2       sum(totFatturato),
3       rank() over (order by sum(numArticoli) desc),
4       rank() over (order by sum(totFatturato) desc)
5 from fatturato f, categoria c
6 where f.codCatArticolo = c.codCat
7 group by categoria;
```

2. Visualizzare per ogni provincia

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

```

1 select provincia, regione,
2     sum(totFatturato) as fatturatoPerProvincia,
3     rank() over (partition by regione
4                     order by sum(totFatturato) desc
5             ) as rankFatturatoPerRegione
6 from cliente c, fatturato f
7 where c.codCliente = f.codCliente
8 group by provincia, regione;

```

3. 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

```

1 select provincia, regione, mese,
2     sum(totFatturato) as fatturatoPerProvinciaPerMese,
3     rank() over (partition by mese
4                     order by sum(totFatturato)
5             ) as rankFatturatoPerMese
6 from cliente c, fatturato f, tempo t
7 where c.codCliente = f.codCliente and t.codTempo = f.codTempo
8 group by provincia, regione, mese;

```

4. Visualizzare per ogni regione e mese

- la regione
- il mese
- il fatturato totale associato alla regione nel mese in esame
- l'incasso cumulativo al trascorrere dei mesi, separato per ogni regione
- l'incasso cumulativo al trascorrere dei mesi, separato per ogni anno e regione

```
1 select regione, mese,
2     sum(TotFatturato) as fatturatoPerMese,
3     sum(TotFatturato) over (
4         partition by regione
5         order by mese
6         rows unbound preceding
7     ) as incassoCumulativoTot,
8     sum(TotFatturato) over (
9         partition by regione, anno
10        order by mese
11        rows unbounded preceding
12    ) as incassoCumulativoPerAnno
13 from cliente c, fatturato f, tempo t
14 where c.CodCliente = f.CodCliente and t.CodTempo = f.CodTempo
15 group by regione, mese, anno;
```

4 Viste materializzate

Le viste materializzate sono necessarie per ridurre la lentezza delle operazioni di group by per grandi moli di dati, le viste materializzate sono dei sommari precalcolati della tabella dei fatti.

Le VM usano con costruto principale la group by, quando si crea una VM è conveniente includere anche le dimensioni a granularità superiori, in modo da poter riutilizzare la tabella.

Per rappresentare le dipendenze delle viste materializzate si utilizza un **reticolo multidimensionale**. Più ci si trova in alto al reticolo più ci si avvia alle dimensioni della tabella dei fatti, più si va in basso più si trova un granularità maggiore.

La scelta delle viste viste tra tutte le possibili combinazioni è data da:

- si sceglie una sola vista da cui è possibile raggiungere tutti gli attributi;
- creo una vista per ogni query;
- scelgo delle viste intermedie che possono portare a ripondere e più query;

4.1 Documentazione Oracle

Riducono i tempi di esecuzione delle group by e non si eseguono più le join. Nel DBMS Oracle esiste la **query rewriting**, che permette grazie all'ottimizzatore di interpretare le query e se i risultati corrispondono alle condizioni di creazioni delle viste, allora la query viene riscritta con la vista.

```
1 create materialized view NAME
2 [build {immediate|deferred}]
3 [refresh {complete|fast|force|never}
   {on commit|on demand}]
4 [enable query rewrite]
5 as
6
7   QUERY
```

- immediate: lo schema della tabella viene popolata immediatamente, dato dallo schema di attributi presenti nella select;
- deferred: la vista viene creata, ma viene popolata successivamente;
- complete: i dati vengono presi interamente dal database;
- fast: i dati vengono presi in modo incrementale;
- force: se possibile viene eseguito il refresh in modalità fast, oppure in modalità complete;
- never: la vista non viene mai aggiornata;

- on commit: ogni volta che viene fatto un commit sulla tabella della query anche la vista viene aggiornata;
- on demand: viene definito dall'utente quando aggiornare la vista;
- enable query rewrite: abilita il dbms ad usare la vista per accellerary le query;

Quando si ha bisogno del **fast refresh**, la vista meterializzata ha bisogno di informazioni aggiuntive, che vengono fornita da tabella di log che informano delle nuove operazioni (insert, delete, update) effettuate sul db, la **materialized view log** è associata ad una tabella che ha subito delle variazioni:

```

1 create materialized view log on TABELLA
2 with sequence, rowid
3 (COLONNA, ...)
4 including new values;
```

- squence: istante temporale in cui è avvenuta la modifica;
- rowid: indica la tupla che ha subito una modifiche;

Su queste keyword si deve definire una lista di attributi da monitorare, si aggiunge **including new values** per supportare l'inserimento di nuove tuple.

Per poter usare insieme l'opzione **fast refresh**, il log deve essere creato su tutte le colonne utilizzate all'interno della query della vista meterializzata, esempio:

```

1 CREATE MATERIALIZED VIEW VIEW1
2 BUILD IMMEDIATE
3 REFRESH FAST ON COMMIT
4 ENABLE QUERY REWRITE
5 AS
6   SELECT
7     TRI,
8     SEM,
9     ANNO,
10    SETTORE,
11    UNIVERISTA,
12    STATO,
13    SUM(NUM_PUBBLICAZIONI)
14  FROM
15    PUBBLICAZIONI P,
16    AUTORE_MAIN A,
17    DATA D
18  WHERE
19    P.IDDATA = D.IDDATA
20    AND P.IDAUTORE = A.IDAUTORE
21  GROUP BY
22    TRI,
23    SEM,
```

```
24      ANNO ,  
25      SETTORE ,  
26      UNIVERSITA ,  
27      STATO ;  
28  
29 -- logs  
30 CREATE MATERIALIZED VIEW LOG ON DATA WITH ROWID ,  
31 SEQUENCE (  
32     IDDATA ,  
33     TRI ,  
34     SEM ,  
35     ANNO  
36 ) INCLUDING NEW VALUES ;  
37  
38 CREATE MATERIALIZED VIEW LOG ON AUTORE_MAIN WITH ROWID ,  
39 SEQUENCE (  
40     IDAUTORE ,  
41     SETTORE ,  
42     UNIVERSITA ,  
43     STATO  
44 ) INCLUDING NEW VALUES ;  
45  
46 CREATE MATERIALIZED VIEW LOG ON PUBBLICAZIONI WITH ROWID ,  
47 SEQUENCE (  
48     IDDATA ,  
49     IDLOUGO ,  
50     IDAUTORE ,  
51     NUM_PUBBLICAZIONI  
52 ) INCLUDING NEW VALUES ;
```

5 Alimentazione dei Data Warehouse

Essendo dei dati derivati, la prima operazione da effettuare è l'ETL, se questo è complesso si va a definire un area di staging in cui il dato viene mantenuto temporaneamente. Il processo di ETL va gestito sei per il popolamento del DW sia per quando verrà aggiornato con dati nuovi.

5.1 Estrazione

L'estrazione statica è la prima estrazione effettuata per popolare il DW. Per fare l'estrazione incrementale si possono:

- creare delle applicazioni ad hoc per i sistemi legacy;
- usando dei log, che non vanno ad interferire con il carico del db;
- usando dei trigger: sono procedure che si attivano quando degli si effettuano delle operazioni specifiche;
- basata su timestamp: dove i recordi hanno il timestamp dell'ultima modifica effettuata su di essi;

5.2 Pulitura

Quando si effettua una estrazione ci si potrebbe trovare di fronte a:

- dati duplicati;
- dati mancanti;
- campo non previsto;
- valori errati o impossibili;
- inconsistenza del valore;

Ogni errore richiede una tecnica specifica per essere risolto, le più comuni sono l'uso di dizionari per controllare errori di battitura, oppure il **join approssimato**, ad esempio: due database non hanno una chiave condivisa per identificare un utente dall'ordine effettutato, allora per fare una join si dovranno prendere i campi comuni, controllandone sempre la consistenza, oppure i problemi di **merge/purge**, ad esmpio: facendo il merge di due db le informazioni potrebbero essere duplicate ...

5.3 Trasformazione

Conversione dei dati nel formato di quelli presenti nel data warehouse.

5.4 Caricamento

In fase di caricamento i dati si caricano nel seguente ordine:

- dimensioni;
- fatti;
- indici e viste;

Problem 5.1 – Progettazione Magazzini

Tabelle:

Tempo(codT, data, mese, 3m, 4m, 6m, anno)

Magazzino(codMa, magazzino, citta, provincia, regione)

Modello(codMo, modello, categoria)

UsoMtqMagazzino(codMa, codT, mtqLiberi, mtqTot)

UsoProdMagazzino(codMa, codMo, codT, numeroProdottiTotale, valoreTotaleProdotti)

Query:

1. Relativamente al primo trimestre dell'anno 2013, considerando solo i magazzini della città di Torino, trovare per ogni coppia (magazzino,data) il valore complessivo di prodotti presenti in tale data nel magazzino e il valore complessivo medio giornaliero di prodotti presenti nel magazzino nel corso della settimana precedente la data in esame (data in esame inclusa):

```

1 select magazzino , data ,
2      sum(valoreTotProdotti) as valoreTot ,
3      avg(sum(valoreTotProdotti)) over (
4          partition by magazzino
5          order by data
6          range between interval '7'
7          day preceding and current row
8      ) as valoreMedioSuGiornoCorrenteESettimanaPrecedente
9
10 from UsoProdMagazzino u , Tempo t , Magazzino m
11 where u.codT = t.codT and
12     u.codMa = m.codMa and
13     citta = 'torino' and
14     anno = 2013 and
15     3m = 1
16 group by magazzino , data;

```

2. Relativamente all'anno 2004, trovare per ogni coppia(città,data) la percentuale di superficie liberagiornaliera nella città. Associare ad ogni coppia un attributo di rank legato alla percentuale disuperficie libera giornaliera nella città (1 per la coppia con la più bassa percentuale di superficie libera giornaliera).

```

1 select citta, data,
2     sum(mtqLiberi) / sum(mtqTot) * 100 as
3     percentualeMtqLiberi,
4     rank() over (
5         order by sum(mtqLiberi) / sum(mtqTot) * 100
6     ) as rankLowestPercentuale
7 from Tempo t, Magazzino m, UsoMtqMagazzino u
8 where t.codT = u.codT and
9     m.codMa = u.codMa and
10    anno = 2004
11 group by citta, data;

```

3. Relativamente ai primi sei mesi dell'anno 2014, trovare per ogni coppia (magazzino,data) la percentuale di superficie libera giornaliera.

```

1 select magazzino, data,
2     100 * sum(mtqLiberi) / sum(mtqTot) as
3     percentualeMtqLiberi
4 from Tempo t, Magazzino m, UsoMtqMagazzino u
5 where t.codT = u.codT and
6     m.codMa = u.codMa and
7     anno = 2014 and
8     mese <= 6
9 group by magazzino, data;

```

4. Relativamente all'anno 2013, trovare per ogni coppia (magazzino,mese) il valore complessivo medio giornaliero di prodotti presenti.

```

1 select distinct magazzino, mese,
2     avg(sum(valoreTotProdotti)) over (
3         partition by magazzino, mese
4     ) as valoreMedioGiornalieroComplessivo
5 from UsoProdMagazzino u, Tempo t, Magazzino m
6 where u.codMa = m.codMa and
7     u.codT = t.codT and
8     anno = 2013
9 group by magazzino, mese, data;

```

5. Relativamente all'anno 2015, trovare per ogni regione il valore complessivo medio giornaliero di prodotti presenti nella regione.

```

1 select regione, mese,
2     avg(sum(valoreTotProdotti)) over (
3         partition by regione, mese
4     ) as valoreMedioGiornalieroComplessivo
5 from UsoProdMagazzino u, Tempo t, Magazzino m
6 where u.codMa = m.codMa and
7     u.codT = t.codT and
8     anno = 2015
9 group by regione, mese, data;

```

6. Relativamente all'anno 2014, trovare per ogni coppia(mese, regione) la percentuale di superficie libera giornaliera nella regione.

```
1 select regione, mese,
2      avg (100 * sum(mtqLiberi) / sum(mtqTot)) over (
3          partition by regione
4      )
5 from UsoMtqMagazzino u, Tempo t, Magazino m
6 where u.codMa = m.codMa and
7       u.codT = t.codT and
8       anno = 2014
9 group by regione, mese, data;
```

6 Data Lake

I data lake sono dei repository di dati, storicizzati per utilizzo futuro così come sono disponibili, in qualsiasi formato. Questi raw data potrebbero essere utilizzati in futuro.

I data lake danno la possibilità di storicizzare i dati per un uso futuro, inoltre tutti i dati sono contenuti in un repository comune, infatti è caratterizzato da bassi costi di storage e mantenimento, però può essere difficile estrapolare dei dati.

7 Data processing

Una collezione è costituita da oggetti di dato,

- **Attributo:** è una proprietà dell'oggetto;
- **Tipi di Attributo:** possono essere nominali, ordinali, intervalli, rapporti;
- **Proprietà dei valori degli attributi:** possiamo definire equivalenza, ordine, addizione, moltiplicazione;
- **Attributi discreti e continui:** discreti hanno un numero finito di valori, i continui hanno dei valori reali;

Tipi di dato da analizzare:

- record: sono i dati presenti in una tabella;
- grafi: come la struttura di una pagina web;
- ordinato: dati in cui esiste il concetto di sequenza;

Esistono vari tipi di dato:

- **Document Data** Per ogni riga ho un documento, per ogni colonna ho degli attributi che descrivono delle parole chiave all'interno del documento, ogni riga è un array che contiene la pesatura degli attributi, la pesatura può essere calcolata con algoritmi specifici;
- **Dato transazionale** Un dato transazionale è formato da un insieme di items all'interno della tabella, ogni transazione è identificata da un ID, nelle tabelle transazionali non esiste il concetto di ordine né tra le riche, né all'interno della transazione;
- **Dato a grafo:** ad esempio le pagine web hanno dei link ad altre pagine, potrei considerare i link come gli archi del grafo, ognuno con un peso specifico, e la singola pagina come un nodo del grafo;
- **Qualità del dato:** nella maggior parte dei casi i dati presentano degli errori, questi possono essere causati da rumori e outliers, dati mancanti o duplicati, ...;
- **Outliers:** dati che escono al di fuori del comportamento medio e quindi molto rumorosi, l'obiettivo dell'analisi di outlier detection è l'individuazione di questi dati e dell'eliminazione;

7.1 Data preprocessing

Le tecniche di preprocessing sono:

- **data aggregation:** Consente di combinare più record o più attributi, si fa questo per effettuare una riduzione della quantità di dati, ed avere dei dati più stabili con una variabilità minore.
- **data reduction:** Si possono effettuare due tipi di riduzione: riduzione degli attributi o riduzione dei valori. Per effettuare queste riduzioni esistono delle tecniche specifiche.

Un'altra tecnica di data preprocessing è il **sampling** è una tecnica di statistica di analisi, ad esempio nella pipeline di datascience il sampling serve per trovare delle rappresentazioni adatte al dataset, facendo N esperimenti su un sample preso dal dataset si andranno a fare dei test sull'intero dataset per verificare che le ipotesi siano confermate, in figura si può vedere come il grafico col numero più piccolo di campia non sia più rappresentativo del dataset originale.

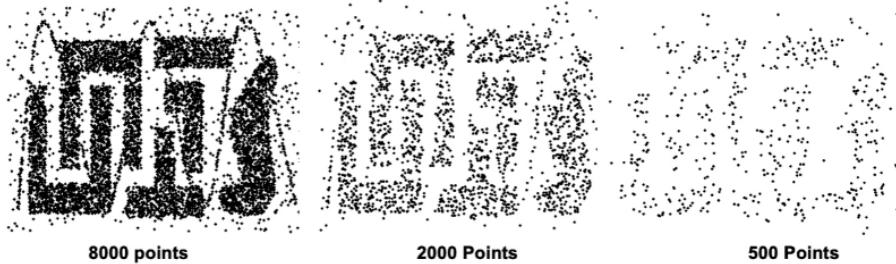


Figure 1: Sampling Example

I tipi di sampling possono essere:

- **randomici:** un dato estratto non viene reinserito;
- **con rimpiazzo:** un dato estratto può essere riestratto;
- **sampling stratificato** (più utilizzato): si può stratificare il dataset a partire da uno o più attributi (le partizioni vengono detti bucket), e poi vengono presi dei valori casuali;

Il problema con questi tipi di approccio è che all'aumentare delle dimensioni i dati diventano più distanti tra loro, questo fenomeno viene detto **curse of dimensionality** e causa tutte le tecniche di identificazione di outliers o dati sparsi inutile, il motivo è che per trovare questi dati le tecniche si basano su calcoli di distanze ma, se tutti i dati sono molto distanti tra di loro queste vengono rese inefficaci. Per ridurre gli effetti di

avere troppe dimensioni si possono applicare delle tecniche di riduzione dimensionale. Per questo motivo vengono utilizzate tecniche di **dimensionality reduction**, alcune di queste sono:

- **pca, svd, ...**: tecniche statistiche;
- **feature selection**: si vuole avere la proiezione del dato su un nuovo attributo in modo da aumentare la varianza, in qualche modo ridurre le feature ridondanti, le tecniche di feature selection sono:
 - **brute force**;
 - **embedded approach**: tramite un albero di selezione si selezionano solo i dati più significativi, può essere fatto quando gli attributi non sono elevati;
 - **filter**: basiti sull'analisi di correlazione, per verificare se esistono delle correlazioni lineari;
 - **wrapper**: viene utilizzato del data mining come black-box per identificare delle combinazioni di feature;
 - **feature creation**: consiste nel combinare più variabili in una solo variabile, viene effettuato anche un processo di feature selection;
- **data creation**: consente di creare nuovi attributi a partire da vecchi attributi che rappresentano meglio il dato:
 - **discretizzazione**: per effettuare una discretizzazione si deve mappare un valore continuo in un range di numeri discreti, una tecnica è quella di generare degli intervalli di uguale lunghezza e se la variabile ricade un in intervallo gli viene associato il simbolo corrispondente, questo potrebbe modellare dei valori outliers o rumorosi, si potrebbe anche usare del clustering, ovvero l'aggregazione di dati in base alla distanza tra vari valori, solitamente vengono usate due tecniche per validare i dati dopo la pipeline. Un caso particolare della discretizzazione è la **binarizzazione** che consiste nel discretizzare una variabile e poi viene fatto il one-hot encoding (i valori vengono mappati su una bitmap);
 - **trasformazione**: un attributo va trasformato quando si vogliono riportare i valori in un'altra scala, una delle tecniche più comuni è la normalizzazione, ad esempio negli algoritmi di clustering viene definito uno spazio normato per calcolare la distanza tra i valori, le normalizzazioni più usate sono:

Theorem 7.1 – min-max

$$v' = \frac{v - \min}{\max - \min} (\text{new_max} - \text{new_min}) + \text{new_min}$$

Theorem 7.2 – z-score

$$v' = \frac{v - \text{mean}}{\text{stand_dev}}$$

Theorem 7.3 – decimal scaling

$$v' = \frac{v}{10^j}$$

j intero più piccolo tale che: $\max(|v'|) < 1$

7.2 Similarità

La similarità e la dissimilarità ci permettono di dire quando degli attributi sono simili o dissimili tra di loro, la similarità viene espresso in un intervallo $[0, 1]$, con 1 = identici, per definire la similarità si definisce un concetto di distanza, ed una **matrice di similarità**, in cui ogni riga e colonna sono presenti i valori, ogni celle rappresenta le distanze tra i due valori. Le tre distanze utilizzate sono: Manhattan, Euclidea, Minkowski. In caso queste distanze non soddisfino i criteri si definisce una distanza attraverso uno spazio vettoriale. Alcune delle distanze sono:

- **manhattan**;
- **euclidean**;
- **minkowski**;
- **mahalanobis**: una distanza importante è la **distanza di mahalanobis**, che mostra quanto due punti sono distanti in una distribuzione.

Theorem 7.4 – Mahalanobis

$$\text{Maha}(\mathbf{x}, \mathbf{y}) = (\mathbf{x} - \mathbf{y})^T \Sigma^{-1} (\mathbf{x} - \mathbf{y})$$

Σ matrice delle covarianze.

Un altro modo per misare la similarità tra due vettori (di valori binari) sono:

- **Simple Matching**: SMC;
- **Coefficiente di Jaccard**;
- **Cosine Similarity**: dati due vettori \mathbf{a} e \mathbf{b} è definito come il prodotto scalare fratto le norme:

Theorem 7.5 – Cosine Similarity

$$\cos(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$$

7.3 Correlazione

Si cercano delle correlazioni negli attributi di un tabella per poter effettuare una riduzione di essi, infatti se due attributi sono correlati uno di loro può essere eliminato. Per computare la correlazione di due attributi (colonne = vettori) esiste il coefficiente di correlazione:

Definition 7.1 – Coefficiente di Pearson

$$coeff(x, y) = \frac{cov(x, y)}{stddev(x)stddev(y)}$$

Più il valore si avvicina a 1 più sono correlati (-1 sono inversamente correlati), mentre vicino allo 0 non sono correlati.

8 Regole di associazione

L'estrazione delle regole di associazione è un modo di trovare delle associazioni tra i valori prenti in un database transazionale. Per decidere queste associazioni solitamente si guardano le ricorrenze statistiche di valori comuni.

Una regola di associazione si definisce come:

Theorem 8.1 – Regola di associazione

$$A, B \implies C$$

Dove degli insiemi di oggetti (**itemset**) possono implicare altri insiemi.

- A, B = corpo della regola;
- C = testa della regola;

La freccia indica la **co-occorrenza**, indica che il copro è legato alla testa nelle transazioni del db. Per esempio: coca, pannolini \implies latte.

Se si lavora con un db relazione possiamo estrarre una transazione associando ad ogni valore il suo attributo.

Definizioni:

Definition 8.1 – k-itemset

È un itemset che contiene k oggetti.

Definition 8.2 – support count

È la frequenza con cui una trasazione appare.

$$\#Roma, Colosseo = 2$$

$$sup(Roma, Colosseo) = 2$$

Data una regola di associazione $A \implies B$:

Definition 8.3 – Supporto

$$\frac{\#A, B}{|T|}$$

È la frazione di transazioni che contengono sia A e B. $|T|$ è la cardinalità del db.

Definition 8.4 – Confidenza

$$\frac{sup(A, B)}{sup(A)}$$

È la frequenza delle transazioni B che contengono anche A.

Per creare dei modelli per estrarre delle relazioni, vengono definiti due parametri che indicano la frequenza con la quale devono apparire le relazioni:

- supporto > **minsup** threshold;
- confidenza > **minconf** threshold;

Questo viene fatto per limitare il numero di relazioni che vengono estratte, perché nella maggior parte dei casi i db sono molto grandi.

L'estrazione si compone di due fasi:

1. si estraggono gli itemset frequenti, attraverso il vincolo sul supporto;
2. si estraggono le regole di associazione, attraverso il vincolo sulla confidenza;

Il passo più oneroso è l'estrazione degli itemset frequenti.

Il **candidato** è l'oggetto che potrebbe essere estratto, il **frequente** è l'oggetto che supera il valore di minsup.

8.1 Algoritmi Di Estrazione Di Itemsets Frequenti

8.1.1 Forza Bruta

Si compara ogni possibile candidato con il supporto di ogni elemento nel db. I possibili candidati sono 2^d dove d sono il numero di items. La complessità è $O(|T| \cdot 2^d \cdot lunghezzaTransazione)$.

8.1.2 Apriori

L'algoritmo di **Apriori** si basa sul principio di quanto un itemset è frequente, infatti il principio di apriori dice che:

"Se un itemset è frequente, allora tutti i suoi sottoinsiemi devono essere frequenti"

La porzione di lattice (spazio delle soluzioni possibili) che ha come radice un itemset frequente (ovvero il suo sottoinsieme) può essere esplorata, altrimenti no.

$$A \subset B \implies sup(A) \geq sup(B)$$

8.1 Algoritmi Di Estrazione Di Itemsets Frequenti REGOLE DI ASSOCIAZIONE

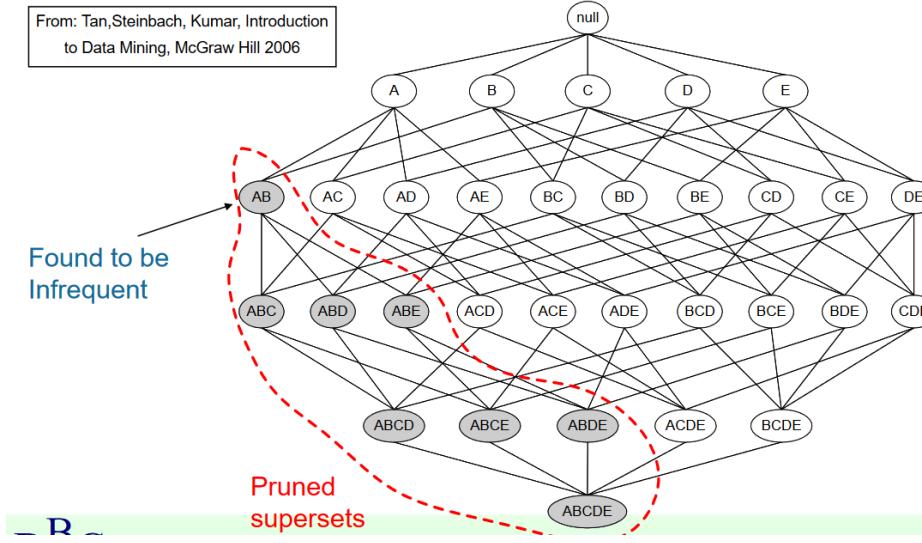


Figure 2: Principio Di Apriori

L’algoritmo di Apriori si basa sulla suddivisione degli itemset in livelli, in ogni livello si prendono dei candidati, facendo il join tra candidati frequenti (che superano la minconf) di livello k si generano candidati di livello k+1, per itemset che non rispettano il criterio di frequenza viene fatto il pruning dell’albero delle scelte.

Sui candidati di lunghezza 2 va applicato il pruning (apriori). Alla fine si troverà l’insieme delle soluzioni che sarà l’unione di tutti gli itemset estratti. Pseudocodice:

```

1 candidateItemset = new List<List<Transaction>, List<Integer>>;
2 frequentItemset = new List<List<Transaction>>;
3
4 frequentItemset[1] = {frequentItems};
5 for (int k = 1; frequentItemset[k].notEmpty(); k++) {
6     condidateItemset[k+1] = frequentItemset[k];
7
8     for (Transaction t : db) {
9         if (transaction of candidateItemset[k+1] is contained in t)
10     {
11         increase the coutnter of that transaction in
12         candidateItemset[k+1];
13     }
14
15     frequentItemset[k+1] = candidateItemset[k+1].filter(t -> t has
16     count greater than minsup);
17 }
17 return frequentItemset.union();

```

Le limitazioni principali di questo algoritmo sono i costi di scansione del database, infatti dovrà essere letto più volte, inoltre se le transazioni sono molto lunghe

l'algoritmo dovrà essere ripetuto $n+1$ volte ($n = \text{lunghezza transazione}$), per superare questo limite si possono usare degli algoritmi per diminuire i costi legati alla lettura.

8.1.3 FP-Growth

Sono state proposte delle varianti dell'algoritmo per ottimizzare i problemi. Negli anni 2000 è stato proposto un nuovo algoritmo basato sulla memorizzazione. L'algoritmo **FP-growth**, instanzia in memoria un albero dove si trova la proiezione del database originale che considera gli item che soddisfano la soglia di supporto, una volta creata la struttura di supporto in memoria non si accede più in memoria secondaria, ottimizzando la lettura degli item. La struttura in memoria prende il nome di **FP-tree**. L'algoritmo funziona nel seguente modo:

1. le transazioni nel db vengono ordinati in base alla cardinalità;
2. viene creata una header table con le transazioni in ordine decrescente, con supporto maggiore della soglia (minsup);
3. viene scansionato per l'ultima volta il database per creare l'FP-tree;
4. per ogni transazione viene preso l'item ed inserito nell'albero, ogni nodo contiene l'item ed il numero di volte che è stato trovato per ogni transazione letta;
5. l'inserimento nell'albero viene fatto a partire dall'ordine degli item nella transazione (simile agli alberi formati a partire da ogni lettera di parola), ogni volta che si passa da un nodo già inserito il suo contatore aumenta, altrimenti si crea un nodo nuovo con count = 1;
6. è importante collegare la header table ai nodi nell'albero, questo viene fatto attraverso una **node link chain**: l'header punta ad un nodo con lo stesso item, quando percorrendo l'albero si trova un altro item il nodo precedente avrà come nodo successivo il nodo corrente;

Viene proposto anche un algoritmo di visita per estrarre gli itemset:

1. viene letta la header table dall'item col supporto più basso;
2. viene creato un **conditional pattern base** (CPB) di un item, una proiezione dell'fp-tree condizionato ad un item;
3. il CPB viene visitato ricorsivamente per trovare i candidati;
4. ad ogni nodo visitato viene recuperato il path dell'albero fino ad esso, se il supporto del nodo è minore del minsup il path viene scartato, e si passa al prossimo nodo della node-link chain;

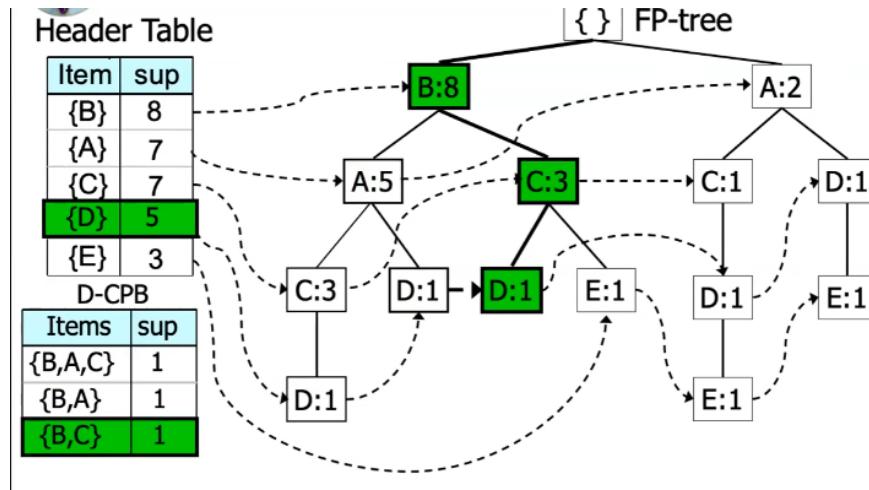


Figure 3: Cpd di D al terzo step

5. data la cpb va creata una Conditional header table, da questa header table, si crea un’altro fp-tree, questo si applica ricorsivamente, ogni chiamata è condizionata (ovvero la cpd sarà preceduta dall’itemset del chiamante: $D \rightarrow DC \rightarrow \dots$);
6. quando non si riesce a creare una header table si torna al chiamante e si passa alla entry superiore nella header table del chiamante;
7. prima di creare la nuova cpb, si prende l’itemset che arriva dal chiamante e se il valore nell’header tabel del item corrente è maggiore del minsup allora l’itemset concatenato all’item corrente viene inseriti negli itemset frequenti;

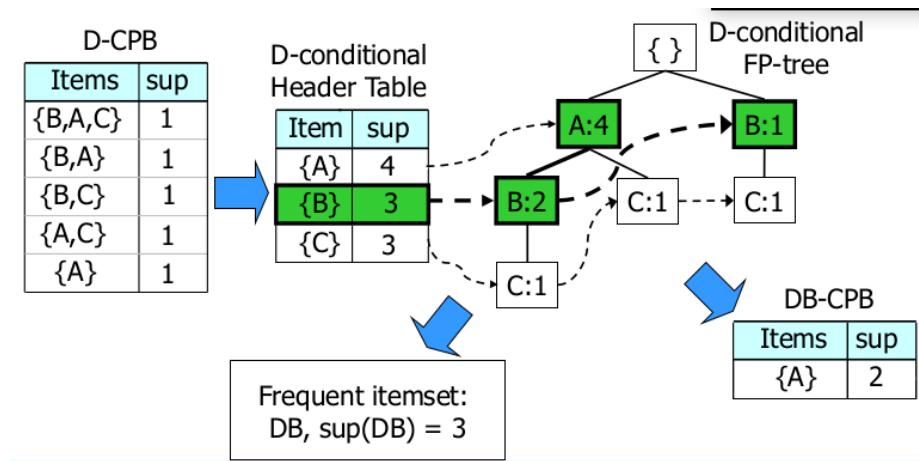


Figure 4: Esempio Aggiunta Itemset Nei Valori Frequenti

Questo algoritmo funziona molto bene se la memoria non viene saturata (bottleneck).

Un altro problema sono l'alto numero di itemset che vengono estratti, infatti anche con db molto piccoli si possono trovare un numero molto elevato di itemset frequenti, per questo si opta per i **itemset massimali frequenti**, IMF per definizioni sono gli itemset che hanno il massimo numero di figli, e sono unici nel loro sottolivello, oppure esistono i **closed itemset** che sono gli itemset con nessuno dei suoi immediati superset hanno lo stesso valore.

$$\text{ItemsetFrequentiMassimali} \subset \text{ItemsetFrequentiChiusi} \subset \text{ItemsetFrequenti}$$

8.2 Effetto delle soglie

La scelta dei valori di supporto deve essere idoneo, infatti con un valore troppo basso non si identificano delle relazioni che portrebbero risultare interessanti e con valori troppo alti emergono relazioni molto deboli. Anche la confidenza comporta dei problemi, se il valore di cui si calcola la confidenza ha un valore molto ampio, si rischiano di ottenere valori errati, per evitare di incrociare queste informazioni si utilizza la regola del lift.

Definition 8.5 – Lift

Dato $r : A \implies B$, allora la Correlazione o lift è:

$$C = \frac{P(A, B)}{P(A)P(B)} = \frac{\text{conf}(r)}{\text{sup}(B)}$$

- $C = 1$: indipendenza statistica;
- $C < 1$: correlazione negativa;
- $C > 1$: correlazione positiva;

Un esempio di regola di associazione potrebbe anche essere l'aggregazione di dati, andano ad accoppiare una tassonomia ai valori, possiamo, aggregando gli attribuiti, vedere il loro supporto crescere, rappresentando in modo generalizzato un comportamento, per andare a soddisfare un servizio.

9 Classificazione

Le classificazioni cercano, attraverso dei modelli di assegnare dei tag ai dati, attraverso delle tecniche supervised (vuol dire che abbiamo già a disposizione un pool di dati da cui possiamo estrapolare le informazioni per assegnare un tipo di tag).

Per applicare la classificazione si ha bisogno di dati di training che hanno già dei tag con il quale si va a generare un modello, per classificare dei nuovi dati si parte dandoli in pasto al modello e partendo dai valori degli attributi si generano delle nuove etichette.

Per poter realizzare un modello di classificazione si ha bisogno di dati di training, usati per generare il modello, e dati di test, usati per validare il modello, ognuno di questi dati ha già associato ad essi una classe di tag. Una volta che si trova un modello adatto, si può inserirlo in un'applicazione per predirre i tag.

Gli algoritmi che generano i modelli hanno delle caratteristiche:

- accuratezza;
- interpretabilità;
- incrementalità: il modello può essere aggiornato all'arrivo di nuovi dati;
- efficienza;
- scalabilità: performance dell'algoritmo rispetto al numero di dati;
- robustezza: capacità dell'algoritmo di operare in presenza di dati rumorosi o mancanti;

9.1 Alberi di Decisione

Attraverso un albero di decisione, dati i dati di input è possibile inferire la classe di etichetta.

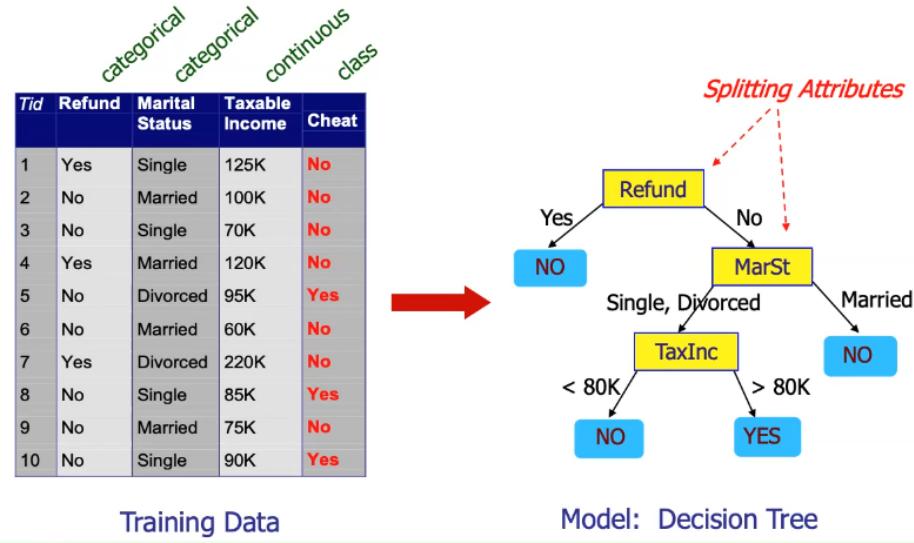


Figure 5: Albero Di Descisione

In un albero di decisione le foglie corrispondono all’etichetta di classe che sarà assegnata all’item in esame. Questo modello dipende molto dai dati, infatti se gli attributi cambiano anche il modello va modificato opportunamente, questo è dovuto al fatto che gli alberi di decisione non sono incrementali. Per generare gli alberi di decisione esistono vari algoritmi, uno di questi è l’**algoritmo di Hunt**, l’algoritmo parte dal leggere il database, il primo passo è quello di individuare l’attributo che riesce a dividere il db in due gruppi più omogenei possibili rispetto al tag. Successivamente si cerca il prossimo attributo che effettua il prossimo miglior partizionamento, fino al raggiungimento delle condizioni di terminazione. Questo albero non è aggiornabile all’arrivo di nuovi dati.

Per effettuare lo split bisogna partire dai tipi di dati con cui si lavora:

- attributo categorico: si possono effettuare n-split diversi per ogni valore, se il tag è binario allora i valori vengono partizionati in due gruppi;
- attributo numerici: si può usare la discretizzazione, oppure si può usare una condizione di test;

Per stimare la purezza (grado di omogeneità) dei nodi che vengono generati, esistono delle metriche per calcolare l’impurità: gini index, entropia, missclassification index. Il primo caso è quello di utilizzo del **Gini index**, con questo metodo viene misurata l’impurità prima e dopo lo split, il gini index si misura con:

$$GINI(t) = 1 - \sum_j (p(j|t))^2$$

$p(j|t)$ = frequenza della classe j al nodo t. Più il gini index si avvicina allo zero, più la classe è **pura**, più il valore si avvicina a $(1 - \frac{1}{\text{numro di classi}})$ più il nodo è impuro, viene

preso l'attributo che genera il gini index col valori più basso. Nelle implementazioni reali vengono fatti dei test utilizzando metriche diverse con successive validazioni.

Per la creazione di un albero va definito anche un criterio di stop:

- minamal gain;
- pre-pruning: se un nodo è quasi puro rimuovo non scendno più nell'albero;
- post-pruning: generato l'albero completo vado a tagliare i rami dell'albero con delle caratteristiche troppo specifiche;

9.2 Random Forest

I random forest sono un'estensione degli alberi di decisione (più alberi di descisione), dato un db si creano n sottoinsiemi dei dati originali, su ogni sottoinsieme si crea un albero di decisione, ognuno di questi modelli per decidere l'etichetta, successivamente si seglie l'etichetta finale in base ai voti (possono anche essere pesati).

Si parte con il **Bootstrap** che va a estrarre gli n sottoinsiemi (con ripescamento) di oggetti randomici, base alla cardinalità, poi si va a creare un albero di decisione applicando una selezione degli attributi (non tutti gli alberi potrebbero utilizzare le stesse feature), ma si scelgono gli attributi che megli modellano i dati che si vogliono analizzare.

9.3 Rule-based Classification

Si parte da un database di training. Le classificazione rule-based sono composte da: (condizione) \Rightarrow y, dove la condizione è in insieme di condizioni booleane e y è la classe. Un esempio è:

(Blood type == warm) && (can fly == yes) \Rightarrow Bird

Dato il db di training si genera il modello, che è composto da n regole R.

Ogni regola R che matcha la entry che si sta elaborando, viene associata la classe all'istanza, se esistono delle collisioni, si può utilizzare la mutua esclusione, e soprattutto (idealmente) le regole devono essere esaustive (non devono esistere casi non gestiti).

Per creare le regole si può partire da un albero di decisione, in questo modo si risolve la mutua esclusione ma non l'esaustività. La strategia che solitamente si segue è il semplificamento di queste regole, le regole possono essere semplificate al completamento dell'albero di decisione, oppure si potrebbe semplificare l'albero con del pruning e poi estrarre le regole, oppure estrarre le regole direttamente dal db. Quando le regole non sono esaustive si una classe di default che solitamente è la maggioritaria.

Le caratteristiche di questi modelli sono:

- accuratezza maggiore degli alberi di decisione;
- interpretabili;
- non sono incremental;
- molto efficienti;
- sono scalabili sia sul training set che sul numero di attributi;
- sono robusti agli outliers;

9.4 Classificazione Associativa

Si utilizzano le regole di associazione per fare delle previsioni, le regole di associazione però vanno ristrette, per la classificazione le uniche regole che interessano sono (condizione) \implies regola di classe (un sottoinsieme delle regole possibili). In questo caso l'estrazione è diversa dalle regole di associazione, infatti la classificazione associativa è composta da regole ordinate con degli indice di qualità, l'ordinamento è fatto sulle regole di associazione (soglia, confidenza, lift). Anche in questo caso non si vuole fare dell'overfitting sul training set, quindi in fase di creazione si dovrà decidere dove fare del pruning.

Le caratteristiche sono:

- ha un accuratezza maggiore alla rule-based;
- modello interpretabile;
- non incrementale;
- efficienza bassa: dato dall'estrazione delle regole di associazione (dato dal min-sup);
- la scalabilità dipende dai dati: dimensione dell'FP-growth (ad esempio);
- non soffre di missing value e robusti agli outliers;

9.5 K-Nearest Neighbor (KNN)

È un algoritmo di classificazione che non ha un modello, ma si basa sul dataset di training, in questo modo non viene effettuato alcuna fase di training. Quando si vuole classificare un nuovo item, si cercano delle similarità dal dataset di training e poi si assegna la classe. Per decidere quali dati dall'itemset di training sono i più simili al nuovo item si utilizza un concetto di vicinanza, trovati i suoi vicini si assegna la classe, il numero di vicini che vengono scelti è dato dal valore K, la scelta di questo valore

influisce col rumore che possono causare i vicini. Le misure di distanza dipende dal tipo di dato.

Le caratteristiche sono:

- l'accuratezza è simile ai precedenti;
- il modello non è interpretabile;
- il training set può essere incrementale;
- tempi di classificazione lunghi;
- la scalabilità è determinata dalla cardinalità dell'training set;

9.6 Bayesian Classification

Theorem 9.1 – Teorema di Bayes

$$P(C, X) = P(X) \cdot P(C|X)$$

La classificazione bayesiana si basa sul teorema di Bayes, questo tipo di classificazione presuppone un'indipendenza statistica dei dati tra di loro, che solitamente è l'ipotesi sbagliata. Quando si vuole classificare una nuova tupla X rispetto ad un classe C si calcola $P(X|C)P(C)$ per ogni classe, il valore più alto trovato corrisponderà alla classe di appartenenza delle tupla. Per calcolare $P(X|C)P(C)$ si calcola:

$$P(C) \prod_{i=1}^N P(x_i|C)$$

dove x_i è un attributo della tupla X (è per questo motivo che si richiede l'indipendenza statistica, altrimenti quella moltiplicazione non sarebbe possibile).

9.7 Support Vector Machines

È una tecnica non interpretabile che va identificare un iperpiano che va a separare le classi di interesse. Questo iperpiano deve massimizzare il suo margine (distanza tra l'iperpiano i punti più vicini delle classi). È possibile definire anche delle curve non-lineari modificando il kernel dello spazio vettoriale.

Le caratteristiche sono:

- performance tra le migliori;
- non interpretabile;
- non incrementale;

- molto efficiente con il giusto tuning;
- scalabilità media;
- robusti ad outlier e rumori;

9.8 Artificial Neural Network

L'algoritmo utilizza delle unità di computazioni detti **neuroni**, questi neuroni sono collegati tra di loro attraverso **sinapsi**. Esistono vari modelli di rete neurale:

- Feed Forward NN: la più basilare;
- Convolutional NN, prima viene modellato il dato utilizzando dei filtri, che poi viene dato ad un FFNN per la computazione finale;
- Rcurrent NN,
- auto-encoder: vengono fatte delle elaborazione di dato dove viene fatto del denoising;

FFNN si ha un livello di input ed un livello di output ovvero l'etichetta di classe che deve essere predetta, questo modello è fully connected, ogni nodo genera un output che va in input ad ogni nodo del livello successivo, queste connessioni sono pesate. Ogni nodo è composto da: la somma di tutti gli output pesati, poi questo valore moltiplicato per un coefficiente e poi dato in pastore ad un funzione di attivazione che genera l'output. Le funzioni di attivazione decidono come elaborare i valori, le più utilizzate sono la sigmoide e la tangente iperbolica. Altre funzioni sono il binary step o la rampa. La funzione utilizzata per l'output finale è la softmax.

Costruire un algoritmo di rete neurale viene fatto con un approccio iterativo:

1. ai vettori peso vengono assegnati dei valori casuali;
2. l'istanza processata viene processata da tutti i layer della rete, l'output viene confrontato con il vero valore dell'etichetta;
3. ad ogni iterazione viene fatta una backpropagation per aggiustare i valori dei pesi e dei nodi;
4. l'iterazione finisce quando si raggiunge una certa percentuale di guess corretti o quando si finisce il dataset di training;

Le caratteristiche sono:

- performance migliori;
- non interpretabile;

- non incrementale;
- il training è molto lento ma la classificazione è molto veloce;
- riechede grandi moli di dati ma il training diventa più lento;
- robusti in presenza di dati rumorosi ed outliers, questo è ottenuto con un grande pool di dati di test;

Un'architettura utilizzata sono le **Convolutional NN**, la parte di computazione convoluzionale vanno ad effettuare delle feature selection sui dati in input, questi dati vengono poi mandati in input in una rete FFNN. Per estrarre le feature un layer convoluzione fa operazione:

1. convoluzione;
2. funzione di attivazione;
3. pooling;

I dati sono rappresentati in tensori (matrice multidimensionale), ogni matrice di input è un tensore, l'output è anch'esso generato in un matrice tensoriale. La convoluzione viene effettuando applicando una finestra di filtro, che viene fatta scivolare su tutto il tensore, solitamente viene aggiunto un padding ai bordi, ogni elaborazione viene effettuata considerando il vicinato di un punto che genera un singolo punto in output. Dopo la convoluzione viene applicata la funzione di attivazione, solitamente nelle CNN viene utilizzata la ReLU (rampa). La fase di pooling è una fase di downsampling, viene fatto applicando dei filtri al tensore in output, il risultato finale è un assegnazione di tag ai vari oggetti presenti nel dato.

Un'altra architettura è quella basato sulle **Recurrent NN** applicate sui tipi di dato con un concetto di tempo, queste architettura hanno un concetto di memoria, infatti all'n-esima elaborazione prendono in considerazione l'elaborazione n-1.

Support Vector Machines

9.9 Model evuation

Nella letteratura esistono delle diverse tecniche per decidere quali dati faranno parte della fase di training di un per la creazione di un modello e quali per la validazione del suddetto. Supponiamo di avere un db con 100 oggetti, si vuole utilizzare una parte di questi oggetti per il training associandogli le caratteristiche dell'input e dell'output, si utilizzano gli oggetti rimanenti per validare il modello creato, verificando il corretto assegnamento delle etichette.

La prima operazione da effettuare è quella di misurare la distribuzione dei tag nei diversi oggetti, calcolare i costi di missclassification e misurare la cardinalità del training set.

Una tecnica di partizione dei dati è l'**hold out**, dove si definiscono delle percentuali fisse tra training e test, generalmente l'80% dei dati vengono usati per il training ed il 20% per la validazione. Queste percentuali vanno mantenute per tutte le classi presenti, serve dunque un sampling di tipo stratificato attraverso tutte le classi, questo vuol dire che la percentuale con cui appaiono le diverse classi deve essere mantenuta nei partizionamenti. Questa tecnica è utilizzata con db di grosse dimensioni.

La tecnica della **cross validation** presenta nativamente la ripetizione della fase di training. I dati vengono divisi in k **fold**, su k-1 fold viene fatto il training mentre sul rimanente viene fatta la validation, questo viene ripetuto per ogni fold. Quando un db è piccolo (e.g. nei casi medici), si prendono il numero di fold uguali al numero di entry nel database, quindi prendendo un singolo oggetto viene fatto il training su tutti gli altri, questa tecnica viene detta **leave-one-out**.

Per poter stimare correttamente un modello deve essere validato in base ai parametri di input (analisi delle sensitività), si devono selezionare i parametri di input del modello e fare delle validazioni, il motivo per il quale il dataset iniziale viene diviso in tre parti:

- parte di training 60%;
- parte di validation (fatto sulla sensitività dei parametri) 20%;
- parte di test 20%;

L'hold-out viene utilizzato per dividere il db in training-validation e test, mentre il cross validation divide il db in training e validation.

Esistono delle metriche per poter stimare in modo oggettivo le predizioni fatte dal modello, per far questo viene utilizzata la **matrice di confusione**. La metrica dell'accuratezza non è sempre affidabile, consideriamo un esempio:

classe	record
0	9900
1	100

assegnando un classificatore di default che assegna sempre la classe 0 la sua accuratezza sarà del 99%, mentre la classe 1 non sarà mai predetta, questo è un esempio di caso in cui le classi sono sbilanciate, il nostro interesse in questi casi è predirre con precisione la classe minoritaria, come nella vita reale solo l'1% della popolazione contrae una malattia, ma quell'1% deve essere predetto con la maggiore precisione. Generalmente si danno delle metriche di valore per gli attributi di interesse, queste metriche calcolate per una classe di interesse vengono dette, **recall** (numero di oggetti correttamente assegnati alla classe / numero di oggetti che appartengono alla classe) e **precision** (numero di oggetti correttamente assegnati alla classe / numero di oggetti assegnati alla classe). Queste metriche vanno massimizzate attraverso l'**F-measure** = $\frac{2rp}{r+p}$.

9.10 Curva ROC

ROC sta per (Receiver Operating Characteristics), questa curva caratterizza il trade-off tra numero positive di hits e falsi positivi. La curva ROC viene plottata su due assi:

- **TPR**: true positive rate (y);
- **FPR**: false positive rate (x);

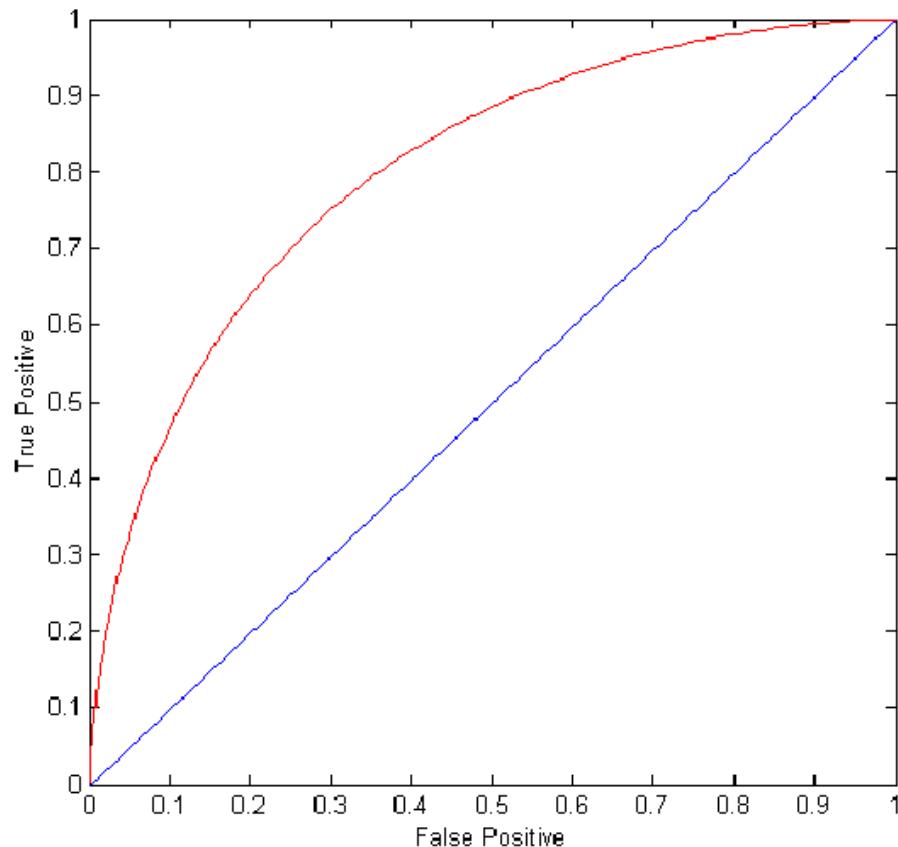


Figure 6: Roc

Generalmente un curva è buona se si trova al di sopra della diagonale.

10 Trigger

Consideriamo un esempio di tema di esame.

a:

```
1 misure: sum(incasso), sum(#consulenze)
2 tabelle: INCASSO, TEMPO, SERVIZION, SEDE-CONSULENTI
3 gb: semestre, tipologia-servizio
4 selezione: regione = 'Lombardia'
```

b:

```
1 misure: sum(incasso), sum(#consulenza)
2 tabella: INCASSO, SEDE-CONSULENTI, SERVIZIO, TEMPO, AZIENDA
3 gb: regione, servizio, anno
4 selezione: Nazionalita = 'Italia' or
      Nazionalita = 'Germania'
```

c:

```
1 misure: sum(incasso), sum(#consulenze)
2 tabelle: INCASSO, SEDE-CONSULENTI, SERVIZIO, TEMPO
3 gb: tipologia-servizio, regione, semestre
4 selezione: anno >= 2017 and anno <= 2019
```

Date le query precedenti si crei una vista materializzata:

```
1 -- Query blocco A
2 select servizio,
       tipologia-servizio,
       semestre,
       anno,
       regione,
       nazionalita,
       sum(incasso),
       sum(#consulenze)
10 from incasso i, tempo t, azienda a, servizio s, sede-consulenze sc
11 where condizioni di join
12 group by servizio,
       tipologia-servizio,
       semestre,
       anno,
       regione,
       nazinalita
```

L'identificare minimale sarà: (servizio, semestre, regione, nazionalita)

Punto 2: ...

Punto 3:

```
1 insert into viewIncassi(servizio,
2     tipologia-servizio,
3     nazionalista,
4     semestre,
5     anno,
```

```

6     regione,
7     incassotot,
8     numconsulenzetot)
9 (blocco A)
```

Punto 4:

```

1 create trigger refreshViewIncassi
2 after insert on incasso
3 for each row
4 declare
5     varServizio varchar(20);
6     varTipologiaServizio varchar(20);
7     varSemestre varchar(20);
8     varAnno varchar(20);
9     varNazionalita varchar(20);
10    varRegione varchar(20);
11    n int;
12 begin
13 -- leggere le tabelle dimensionale x recuperare
14 -- i valori dell'identificatore della vista materializzata:
15 -- servizio, nazionalita, semestre, regione
16 select servizio, tipologia-servizio into varServizio,
17     varTipologiaServizio
18 from servizio
19 where idServizio = :NEW.idServizio;
20
21 select semestre, anno
22 from tempo
23 where idTempo = :NEW.idTempo;
24
25 select nazionalita into varNazionalita
26 from azienda
27 where idCategoriaAzienda = :NEW.idCategoriaAzienda
28
29 select regione into varRegione
30 from sede-consulenti
31 where idSede = :NEW.idSede;
32
33 -- verifoco se esiste una tupla in viewIncassi con
34 -- associati i valori estratti
35
36 select count(*) into n
37 from viewIncassi
38 where nazionalita = varNazionalita and
39     semestre = varServizio and
40     servizio = varServizio and
41     regione = varRegione;
42
43 if (n > 0) then
44     update viewIncassi
45     set incassotot = incassotot + :NEW.incasso
```

```

46     numConsulenze += :NEW.#consulenze
47     where servio = varServizio and
48         nazionalita = varNazionalita and
49         semestre = varSemestre and
50         regiono = varRegione;
51 else
52     insert into viewIncassi ( ... , )
53     values (varServizio,
54             varTipoServizio,
55             varNazionalita,
56             varSemestre,
57             varAnno,
58             varRegione,
59             :NEW.incasso,
60             :NEW.#consulenze);
61 end if;
62 end;

```

Punto 5:

```

1 create trigger updateViewIncassi
2 after update of tipologiaServizion on servizio
3 for each row
4 declare
5     typeServizio varchar(20);
6 begin
7
8
9 end;

```

Punto 6:

```

1 create materialized view log on incasso
2 with sequence, row id
3 (...)
4 including new values;
5
6 create materialized view log on servizio
7 with sequence, row id
8 ( ... )
9 including new values;
10
11 create materialized view log on tempo
12 with sequence, row id
13 (. ...)
14 including new values;

```

11 Lab 03

1.

```
1 select mese, tariffa,
2     sum(prezzo),
3     sum(sum(prezzo)) over () as prezzo_comp,
4     sum(sum(prezzo)) over (partition by mese) as prezzo_per_mese,
5     sum(sum(prezzo)) over (partition by tipo_tariffa) as
6         prezzo_per_tariffa
7 from tempo te, fatti f, tariffa t
8 where te.id_tempo = f.id_tempo and
9     f.id_tar = t.id_tar and
10    anno = 2003
11 group by mese, tipo_tariffa;
```

2.

```
1 select mese
2     sum(chiamate) as chiamte_tot,
3     sum(sum(prezzo)) as incasso_tot,
4     rank() over (
5         order by sum(prezzo) desc
6     ) as rank_più_chiamate
7 from fatti f, tempo t
8 where f.id_tempo = t.id_tempo
9 group by mese;
```

3.

```
1 select mese
2     sum(chiamate) as chiamte_tot,
3     sum(prezzo) as incasso_tot,
4     rank() over (
5         order by sum(chiamate) desc
6     ) as rank_più_chiamate
7 from fatti f, tempo t
8 where f.id_tempo = t.id_tempo and
9     anno = 2003
10 group by mese;
```

4.

```
1 select tipo_tariffa
2     sum(prezzo)
3 from fatti f, tariffa t
4 where f.id_tar = t.id_tar and
5     mese = 'luglio' and
6     anno = 2003
7 group by tipo_tariffa;
```

5.

```
1 select mese
2     sum(prezzo) as incaso_tot,
```

```

3     sum(sum(prezzo)) over (
4         order by mese
5         rows unbounded preceding
6     ) as incasso_da_inizio_anno
7 from fatti f, tempo t
8 where f.id_tempo = t.id_tempo
9 group by mese;

```

6.

```

1 select tipo_tariffa, mese,
2     sum(sum(prezzo)) over (partition by mese) as incasso_per_mese,
3     sum(sum(prezzo)) over (partition by tipo_tariffa) as
4         incasso_per_tariffa,
5     100 * sum(sum(prezzo)) over (partition by mese) /
6         sum(sum(prezzo)) over (partition by tariffa)
7 from fatti f, tempo t
8 where f.id_tempo = t.id_tempo and
9     anno = 2003
9 group by tipo_tariffa, mese;

```

vista:

```

1 create materialized view view_fatti
2 build immediate
3 as
4 select mese,
5     tipo_tariffa,
6     anno,
7     sum(prezzo) as prezzo,
8     sum(chiamate) as chiamate
9 from fatti f, tempo te, tariffa t
10 where f.id_tempo = te.id_tempo and
11     f.id_tar = t.id_tar and
12     anno = 2003
13 group by mese,
14     tipo_tariffa,
15     anno,
16     prezzo ,
17     chiamate;

```

12 Clustering

Il clustering consiste nel raggruppamento di oggetti similari tra di loro, questa tecnica viene detta di **unsupervised learning** perchè per effettuare questa suddivisione parte solo da informazioni presenti nei dati. Le tecniche del cluster analysis si pone l'obbiettivo di partizionare il db in sottogruppi, dove gli oggetti in un sotto gruppo sono vicini tra di loro mentre sono distanti con gli oggetti degli altri sottogruppi. Col termine clustering si definiscono dei cluster, ovvero degli insiemi di dati, il clustering si suddivide in: partizionale (i dati appartengono ad uno ed un solo gruppo), gerarchico (elementi rappresentati da un albero gerarchico, che identifica i partizionamenti detto dendogramma). Gli algoritmi di clustering possono suddivisi per i gruppi di cluster che si vengono a formare, uno di questi è il partizionamento esclusivo vs non-esclusivo un punto può appartenere a più sottogruppi, fuzzy vs non-fuzzy dove un punto è associato ad ogni gruppo ma con un peso per ognuno di essi, parziale quando il partizionamento viene assegnato solo ad sottogruppo escludendo i dati rumorosi, completa quando ad elemento viene associato un tag. I gruppi identificati possono essere caratterizzati da: cardinalità, densità, forma; esistono degli algoritmi in grado di identificare gruppi a densità omogenea ed eterogenea, per ogni caratteristica esiste un tipo di algoritmo. I gruppi che si ottengono possono essere:

- ben separati: distanza massimizzate tra gruppi diversi e minimizzate tra elementi di un gruppo;
- center-based: gruppi rappresentati da un punto medio, **centroide** media dei punti in cluster, **medoide** media dei punti più rappresentativi del cluster;
- cluster continui;
- density-based: i cluster hanno densità uguale;
- cluster concettuali;

12.1 K-means

Ogni cluster è associato con un centroide, l'algoritmo prende in input un parametro K che corrisponde al numero di cluster. L'algoritmo è formato da:

```

1 seleziona K centroidi casuali
2 do
3   si formano K cluster a partire dai centroidi assegnando i valori
     piu vicini
4   si ricalcolano i centroidi a partire dai cluster
5 while (i centroidi non cambiano)

```

Questo algoritmo converge abbastanza velocemente infatti ha $O(\text{num-punti} * K * \text{iterazioni} * \text{num-attributi})$, per questo motivo l'algoritmo viene fatto eseguire molteplici

volte per eliminare il problema dell'assegnazione casuale dei centroidi che potrebbe a portare anche a cluster vuoti. Per valutare quanto è buono un partizionamento si usa l'SSE, Sum of Squared Error.

Definition 12.1 – SSE

$$SSE = \sum_{i=1}^K \sum_{x \in C_i} dist^2(m_i, x)$$

Questa metrica è molto buona per calcolare la validità dell'algoritmo su stessi K (o anche confrontare le prestazioni di diversi algoritmi), se però si decide di aumentare K allora l'SSE è sempre confrontabile ma i valori saranno più piccoli al crescere di K , perché la base dati rimane la stessa, dunque con più partizionamento i dati sono molto più coesi tra di loro.

Oltre ad utilizzare l'approccio delle **run multiple** si possono utilizzare come centroidi utilizzando punti del database stesso usando altre tecniche. Si possono anche utilizzare centroidi maggiori di K e poi al termine delle run scegliere tra questi i centroidi iniziali, si può anche utilizzare del postprocessing o un bisect K-means.

Esiste il problema dei cluster vuoti quando si fanno delle iterazioni, per evitare che vengano creati cluster vuoti si prende un punto dal cluster con l'SSE più grande oppure scegliere il punto con l'SSE più grande, riassegnando il centroide col un cluster vuoto al punto estratto si risolve questo problema, se più cluster sono vuoti si ripete fino ad ottenere tutti con almeno un elemento.

Per ridurre il rumore dei dati eliminando gli outliers si possono applicare operazioni di pre-processing. Una volta finito l'algoritmo si possono applicare operazioni di post-processing come: eliminare cluster piccoli, eliminare cluster con SSE molto grande, oppure fare il merge di cluster vicini tra di loro, per le operazioni di post-processing non esistono delle linee guida, ma solitamente dipende dal contesto.

12.2 Bisecting K-means

Il bisecting

...

13 Introduzione ai DBMS

Il DBMS permette di avere una gestione concorrente delle informazioni, oltre ad avere delle routine che operano sui dati, la sua struttura è molto complessa ed è strutturata in moduli, l'entry point è una query.

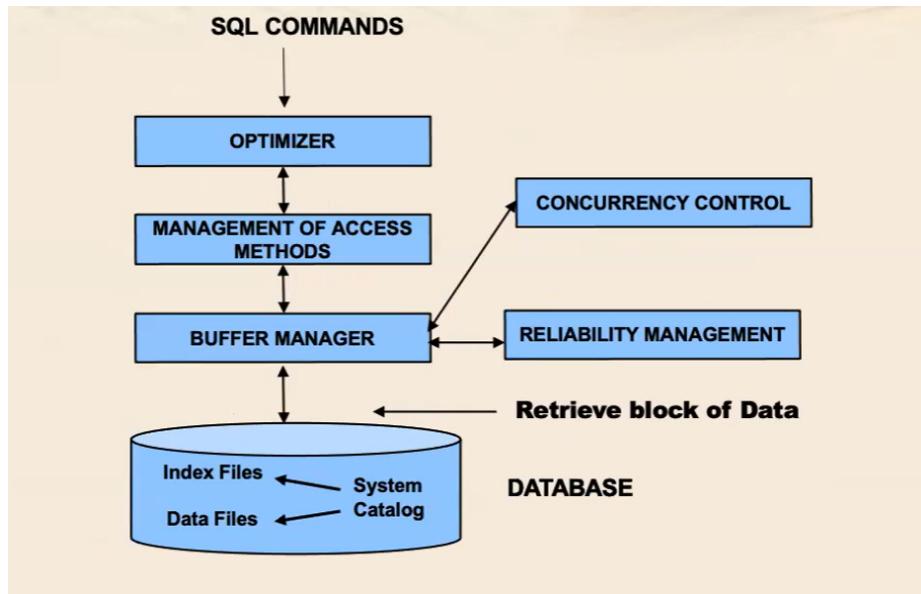


Figure 7: Architettura Dbms

Il DBMS è composto da:

- ottimizzatore: prende delle decisioni in base alla distribuzione dei dati, oltre a fare il parsing della stringa;
- access method manager: viene eseguito un metodo specifico per leggere la memoria;
- buffer manager: buffer nella memoria principale che ottimizza le operazioni di I/O;
- concurrency control: gestisce l'accesso concorrente ai dati;
- reliability manager: garantisce la correttezza del contenuto dei dati del db, utilizzando dei file di log in cui vengono scritte le operazioni effettuate;

Il concetto fondamentale nel contesto dei db sono le **transazioni**: ovvero di una serie di operazioni che rappresentano una singola unità di lavoro. Una transazione termina con COMMIT o ROLLBACK. Le transazioni sono caratterizzate da ACID:

- **atomic**: tutte le operazioni devono andare a buon fine o nessuna, le operazioni che controllano lo stato del sistema sono **UNDO** o **REDO**;
- **consistency**: non vanno violati i vincoli di consistenza del DBMS, vincoli di chiave, chiave esterna, ...;
- **isolation**: le transazioni operano in modo indipendente tra di loro, i dati intermedi non sono visibili all'esterno;
- **durability**: i dati di una transazione non possono essere persi, questa proprietà è garantita dal reliability manager;

13.1 Buffer Manager

Il buffer manager ha accesso ad una parte della memoria principale a cui gli applicativi hanno accesso. Il buffer è organizzato in **parole** o **pagine** di dati, che è l'unità minima trasferibile dalla memoria secondaria, per essere efficace nelle operazioni di IO le pagine devono essere presenti nel buffer per velocizzare le operazioni, per questo motivo si utilizza il principio di località. Ogni pagina deve avere:

- l'ID del file;
- l'ID del blocco;

Per ogni pagina esistono anche due stati:

- **count**: numero di transazioni che stanno utilizzando la pagina;
- **dirty bit**: settato quando la pagina è stata modificata e non è stata ancora caricata nella memoria secondaria;

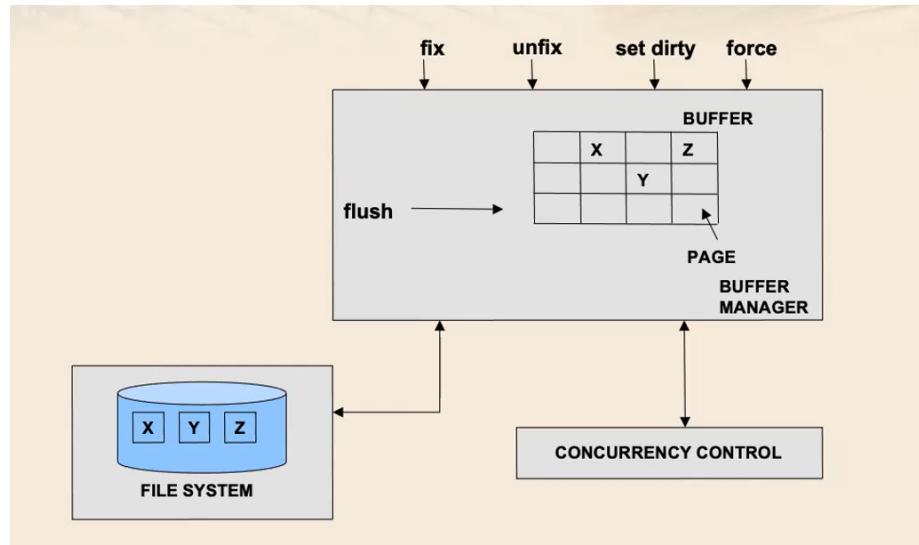


Figure 8: Buffer Manager

Il buffer manager ha a disposizione una parte della memoria principale in cui salva delle pagine di dati dal DB quando viene fatta una query. Esistono delle primitive per poter accedere a questa memoria:

- **fix**: richiesta dalla transazione, se la pagina è disponibile allora viene passata, altrimenti si fanno delle operazioni di IO per recuperare la pagina, se il buffer è pieno un'altra pagina deve essere rimpiazzata, solitamente sarà un pagina con il **count=0**, questa pagina viene detta **vittima** e verrà rimpiazzata con

qualla richiesta dalla transazione; se durante la transazione una pagina ha il **dirty bit = 1** allora deve essere fatta un'operazione di IO per sincronizzare la pagina. Prima che la lettura inizii il count viene incrementato di 1;

- **unfix**: viene richiesta dalla transazione quando la pagina non deve essere utilizzata;
- **set dirty**: setta il valore dirty ad una pagina quando viene fatta un'operazione di scrittura;
- **force**: richiede un trasferimento sincrono della pagina al disco, quindi si scrive la pagina, non è detto che la pagina venga rimossa;
- **flush**: richiede un trasferimento asincrono della pagina al disco;
- **steal**: quando si utilizza steal il buffer manager può selezionare una pagina locked con un count=0, detta pagina **vittima**, e la rimuove dal buffer, se non-steal è settato non si può selezionare una pagina su cui sta avvenendo una transazione;
- **force**: tutte le pagine di una transazione sono scritte in modo sincrono sul disco;
- **no-force**: utilizza flush per scrivere le pagine in modo asincrono sul disco;

Inoltre il buffer manager mette a disposizione:

- sequential read;
- write and sequential write;
- directory;

13.2 Accesso alla Memoria

L'access methods manager riceve dall'ottimizzatore il piano di esecuzione e decide quale metodo di accesso utilizzare per leggere o scrivere i dati, vengono poi selezionati i blocchi dei file da prendere dalla memoria principale, questi dati sono richiesti al buffer manager.

Le strutture fisiche che si possono utilizzare sono dipendenti dal tipo di operazione che si effettuano su di loro (select, update, delete, ...), queste strutture sono dette di tipo **accessorio**, in particolare gli **indici**, che sono strutture definite all'interno del db per velocizzare le operazioni di IO. I dati fisici possono essere salvati in strutture sequenziali o strutture di heap, mentre gli indici utilizzano strutture ad albero, bitmap, unclustered hash.

Nelle strutture squenziali le tuple sono storicizzate nella pagina seguendo l'ordine di inserimento, il vantaggio è che lo spazio occupato è massimo all'interno della pagina e che l'inserimento è molto veloce, quando si fanno delle operazioni di delete si vengono a creare dei buchi, mentre con l'update si rischia di scrivere più dati di quelli che avevo prima.

Gli **heap file** utilizzano delle strutture secondarie per avvalersi di ricerca veloce, tutti i tipi di operazione sono identiche a quelle delle strutture sequenziali.

Le **ordered sequential structures** sono delle strutture sequenziali ordinate in base ad una chiave di ordinamento. I dati ordinati servono per velocizzare le operazioni per quelle interrogazioni che usano la stessa chiave di ordinamento. Come contro le operazioni di delete e inserimento sono molto onerose. Viene lasciato dello spazio in più per eventuali operazioni di update, oppure viene implementato un file di **overflow**, dove vengono salvati la parte di dati che dopo un'update o insert non entra nella pagina.

Nelle **strutture ad albero** nei nodi si trovano solo le chiavi per effettuare la ricerca, negli indici unclustered le foglie si trovano parte dei valori della tuple e il puntatore alla tupla originale, mentre in una struttura clustered le foglie corrispondono ai puntatori della tupla, inoltre da una foglia si può passare all'altra essendo tutte ordinate in modo seequenziale (come nei **B+-Tree**).

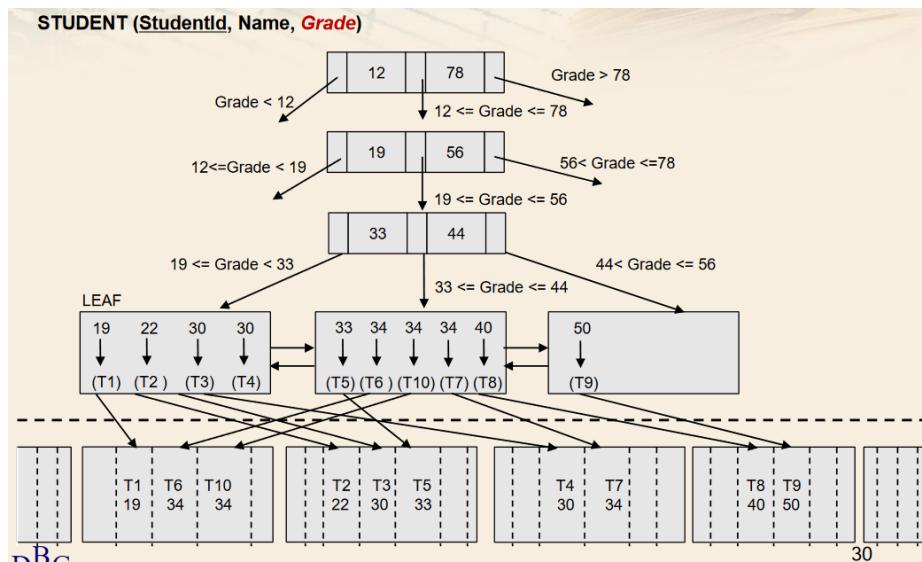


Figure 9: Unclustered B+-Tree Index

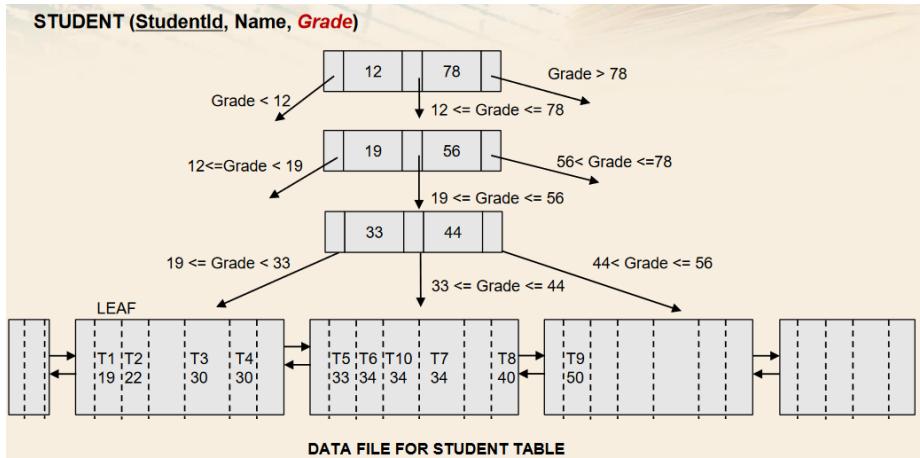


Figure 10: Clustered B+-Tree Index

Su una tabella può essere creato un unico indice di tipo clustered, mentre possono essere create più strutture unclustered.

Un'altra struttura utilizzabile è la **struttura di hash**, utilizza una funzione di hash per assegnare ad ogni tupla secondo una chiave, un blocco tra quelli predefiniti della struttura, anche le strutture di hash possono essere clustered o unclustered.

I **bitmap index** sono delle strutture unclustered, che si portano bene a rappresentare gli attributi categorici.

13.3 Progettazione Fisica

Per fare una buona progettazione fisica servono

- il progetto logico;
- il carico di lavoro in termine di query e frequenza di esecuzione;
- engine del database;

Nella progettazione fisica dato il carico delle query vengono definiti uno o più indici e le strutture dati.

Theorem 13.1 – Euristiche sulla progettazione fisica

- mai indicizzare una tabella piccola;
- mai indicizzare un attributo con bassa cardinalità;
- analizzare i predicati nelle clausole **where**;
- quando si creano degli indici composti si deve considerare il costo del mantenimento;

- per migliorare le operazioni di **join** si usano il **nested loop** (quando si joina un tabella piccola con una grande), ed il **merge scan** (vengono ordinate le tabelle prima di fare il join);
- per le **group by** si usa un ordinamento una struttura di hash, l'ottimizzatore spesso anticipa le operazioni di group by (**group by push down**) diminuendo di molto la cardinalità dei dati da analizzare;

Nelle condizioni di join vengono definite due tabelle:

- **outer**: tabella letta in modo sequenziale;
- **inner**: tabella letta n volte, tante quante le righe della tabella outer;

Le alternative per il join sono:

- hash join;
- nested loop: se la tabella è piccola comunque non si indicizza;

Se si decide di creare un indice composto è meglio che l'indice sia **coprente**, ovvero che leggendo l'indice si riesce a rispondere alla query senza leggere la tabella, altrimenti diventa troppo oneroso manterlo, diventa più efficace un indice su un solo attributo.

13.4 Ottimizzatore delle query

L'ottimizzatore delle query garantisce efficienza ed indipendenza dai dati. L'ottimizzatore genera un piano di esecuzione, basandosi su delle statistiche, come sulla distribuzione dei valori e di come quei valori sono distribuiti nei vari blocchi fisici, le soluzioni sono dinamiche, infatti al cambiamento dei dati può cambiare anche il piano di esecuzione.

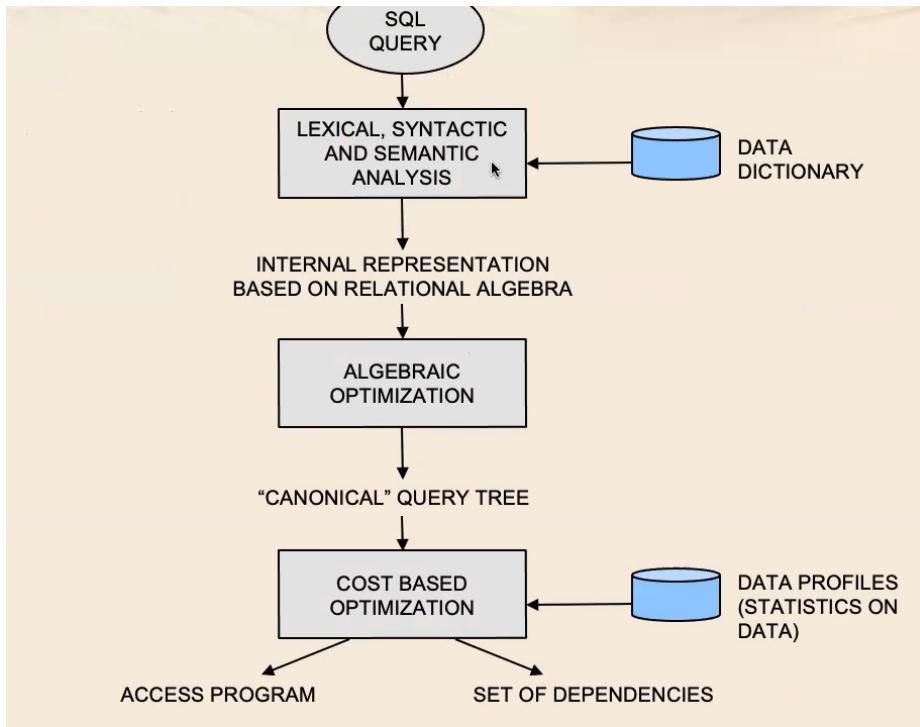


Figure 11: Schema Ottimizzatore

L'ottimizzatore fa un controllo su:

- errori lessicali: misspelled keywords;
- errori sintattici: errori nella grammatica dell'sql;
- errori semantici: viene utilizzato il data dictionary per controllare se un oggetto esiste;

Effettuati i controlli genera un rappresentazione interna in algebra relazionale. Il passaggio successivo è l'ottimizzazione algebrica, creando un albero di esecuzione. Un volta creato l'albero viene fatta un'ottimizzazione basata sui costi di accesso, quindi ad ogni parte dell'albero vengono assegnati dei punti di accesso. L'ottimizzatore può essere utilizzato in due modalità:

- **compile and go**: una query non viene salvata, ma viene ricompilata ogni volta, utile quando i dati variano abbastanza frequentemente nel tempo;
- **compile and store**: la query viene compilata e salvata per riutilizzi successivi;

13.4.1 Ottimizzazione algebrica

Si basa su regole di ottimizzazione dell'algebra relazionale e sulle statistiche dei dati. Esistono delle trasformazioni che si possono applicare alle equazioni dell'algebra relazionale:

-

-

13.4.2 Ottimizzazione basata sui costi

Per fare una ottimizzazione basata sui costi è necessario un modulo Nel **data profile** sono presenti:

- cardinalità delle tuple;
- numero di byte delle tuple;
- numero di byte degli attributi di una tupla;
- numero di valori distinti;
- valori minimi e massimi di un attributo;

Se la query è compile and go in output si avrà solamente le modalità di accesso, altrimenti se la query è compilata insieme al piano di esecuzione vengono passate l'insieme di dipendenze, grazie a questo insieme si decideranno se la query dovrà essere modificata per soddisfare i cambiamenti che avvengono al db col passare del tempo.

13.5 Operatori di accesso

La rappresentazione interna dell'ottimizzatore è una query tree. Quando si effettua la lettura delle tabelle si valutano i predicati di selezione, il predicato viene applicato in fase di lettura, anche attraverso utilizzo di index. Anche il join è un'operazione molto complessa, nei dbms esistono diversi tipi di join:

- **nested loop**: ci sono due tabelle sbilanciate, una grande ed una piccola, per ogni tuple della outer table (grande) viene effettuata una scansione delle inner table (piccola), un eventuale indice creato sulla inner table potrebbe velocizzare l'operazione di join;
- **merge scan join**: tiene conto dell'ordinamento delle tabelle rispetto all'attributo di join, se le tabelle non sono ordinate allora viene fatto un sort;
- **hash join**: viene applicata una funzione di hash rispetto agli attributi della condizione di join, all'interno delle tabella vengono creati dei bucket, all'interno i dati vengono ordinati e poi viene fatto il join tra i bucket;

- **bitmapped join:** è una specie di index definito sui dati (infatti questo join è dipendente dai dati, non come le altre tipologie di join), viene create una bitmap, ogni riga con la rowid della tabella originale e gli attributi che soddisfano la query hanno un 1, mentre gli attributi che non la soddisfano hanno uno zero;

L'operazioni di group by possono essere velocizzata con:

- **sort based** prima con un ordinamento e poi con una selezione;
- **hash based**: la tabella viene bucketizzata;
- **materialized views**: con query rewriting;

13.6 Piani di esecuzione

A differenza dell'obiettivo della query esistono diversi tipi di esecuzione, se i dati servono il prima possibile si trovano dei metodi da iniziare a mandarli appena sono disponibili, mentre nella applicazione di tipo batch ciò che conta è solo il tempo complessivo;

```

1 select snam, s.id
2 from exam e, student s
3 where s.sid = e.sid and score >= 27
4 order by sname;

```

Esempio di piano di esecuzione.

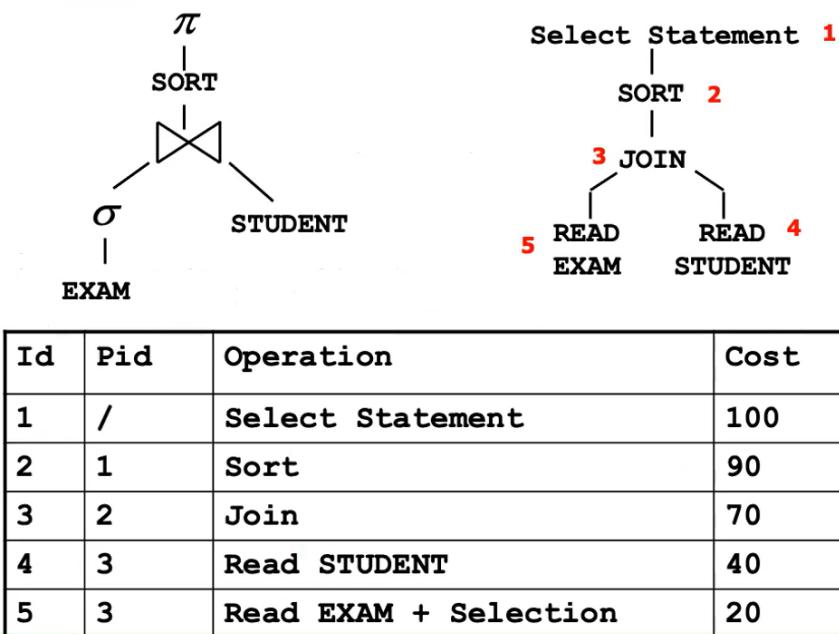


Figure 12: Esecuzione1

Gli accessi alle tabelle possono essere diretti o attraverso gli index, se gli index sono ricoprenti non c'è bisogno di accedere alla tabella.

- **full table scan:** lettura sequenziale della tabella, processando in contemporanea i predicati della where, in oracle esiste una funzione che permette di leggere più pagine di una tabella per velocizzare le operazioni di IO, infatti quando il dbms legge un qualcosa viene sempre letta l'unità minima (pagina), per sapere la distribuzione dei dati nella pagina esiste l'**index clustering factor**, ovvero quanto i dati sono vicini nei blocchi, se l'ICF è alto allora potrebbe essere ottimale leggere la tabella con un index, se è basso allora conviene leggere al tabella;
- ...

Esistono anche dei metodi di lettura relative agli index:

- **index unique scans:** viene utilizzato quando l'attributo su cui è stato creato l'indice è UNIQUE, l'indice per ogni valore della chiave ritorna un solo valore;
- **index range scan:** dalla lettura dell'indice si possono ottenere più rowid, ordinati in modo ascendente;
- **index full scan:** si legge tutto l'indice, questo si fa quando non si ha un predicato ben preciso, il vantaggio è che i dati vengono restituiti ordinati rispetto alla chiave dell'indice;
- **fast full index scan:** viene utilizzato quando l'indice è coprente (non si accede alla tabella), ma si perde l'ordinamento della chiave dell'indice;
- **rowid:** riceve in input il rowid presente nell'index corrispondente a quella della riga della tabella a cui sta puntando;

Il data dictionary sono delle tabelle con statistiche sui dati presenti nelle tabelle del db, oracle per immagazzinare informazioni sulle colonne utilizza gli **istogrammi**. Gli istogrammi sono:

- **height-balanced:** si ha la distribuzione dei valori assunti dall'attributo, i valori vengono distribuiti in intervalli di uguale lunghezza. L'istogramma divide gli intervalli in bucket, il loro valore corrisponde al valore più grande all'interno del bucket;

```
SELECT column_name, num_distinct, num_buckets, histogram
  FROM USER_TAB_COL_STATISTICS
 WHERE table_name = 'INVENTORIES' AND column_name = 'QUANTITY_ON_HAND';

COLUMN_NAME          NUM_DISTINCT  NUM_BUCKETS HISTOGRAM
-----              -----          -----
QUANTITY_ON_HAND      237           10 HEIGHT BALANCED
```

Figure 13: Tipo Di Istogramma

- **frequency**: viene indicata la frequenza dei diversi valori, sempre tra i diversi bucket;

Un'altra opzione può essere la specifica per il miglior **throughput** (applicazioni batch) ed il miglior **response time** (applicazioni real time).

13.7 Regole per valutazione di ottimizzazione

Theorem 13.2 – Regole ottimizzazione

REGOLE EURISTICHE:

- tabella piccola:
 - piccola se le tuple sono $\leq 10^3$, no indice;
 - medio/grande $> 10^3 - 10^4$, valutare indice;
- valutazione predicato selettivo:
 - selettività elevata $\leq \frac{1}{10}$, valutazione indice;
 - selettività medio/bassa $> \frac{1}{10}$, non si valuta l'indice per l'attributo;

PROCEDIMENTO:

- analise delle tabelle:
 - statistiche;
 - cardinalità tabella;
 - min-max attributo (hp distribuzione uniforme);
 - numero di valori distinti (hp distribuzione uniforme);
 - cardinalità sulla selezione;
- ispezionare istruzione sql;
- definizione query tree;
- valutazione delle cardinalità:
 - foglie;
 - nodi intermedi;
 - nodo finale;
- definizione del piano di esecuzione (assegnare un metodo ad ogni nodo);

- valutazion di possibili strutture fisiche accessorie per migliorare il piano di esecuzione:
 - indicare la selettività;
 - aiuto nella group by — sort — join;
 - indice coprente oppure no;
 - primario (clustered) / secondario;
 - tipo di indice;
- valutare anticipo group by;

Example 13.1

- Sono date le relazioni seguenti (le chiavi primarie sono sottolineate):
 - **MULTA(NumSerie, Data, Ora, TipoInfrazione, CodFiscale)**
- Si ipotizzino le seguenti cardinalità per le tabelle:
 - $\text{card}(\text{MULTA}) \approx 10^5$ tuple
 - $\text{MIN}(\text{MULTA.Data}) = 1/1/2005$ and $\text{MAX}(\text{MULTA.Data}) = 31/12/2005$
 - $\text{card}(\sigma_{\text{TipoInfrazione}='Tip010'} \text{ MULTA}) \approx 10^3$ tuple
 - Valori distinti di TipoInfrazione 100

Figure 14: Ex Multa

```

1 select data, count(*)
2 from multa
3 where data >= 1/10/2005 and data <= 30/11/2005
4 group by data;

```

60 tuple; select statement

$$\pi_{data, count(*)}$$

60 tuple; hash gb

$$gb_{data}$$

selettività 2/12, card = $2 \cdot 10^4$

$$\sigma_{data \geq 1/10/2005 \& data \leq 30/11/2005}$$

10^5 tuple, full table scan + filter

multa

Indice su data:

- predicato poco selettivo ($\frac{1}{6}$);
- aiuta la gb (dati ordinati rispetto a data);
- indice coprente;
- indice secondario;

```
1 create index myindex1 on multa(data);
```

SOLUZIONE 1:

- select statement;
- gb hash;
- fast full index scan on myindex1

SOLUZIONE 2:

- select statement;
- gb no sort;
- full index scan on myindex1 or index range scan on myindex1;

Example 13.2

```
1 select data, count(*)
2 from multa
3 where data > = 1/10/2005 and data < = 30/11/2005
4     and tipoInfrazione = 'tip010'
5 group by data;
```

QUERY TREE:

- $\pi_{data, count(*)}$
 - 60
 - select statement;
- gb data;

- 60
- gb sort
- σ_{data} ;
- $\frac{1}{6}10^3$
- $\sigma_{tipoinfrazione}$;
- 10^3
- multa
- 10^5
- full table scan + filter

Indice su data (*non conveniente*):

- bassa selettività;
- poco aiuto a gb (pochi record);
- richiede accesso alla tabella per attributo `tipoInfrazione`;

Indice su tipo infrazione:

- selettività molto buona $\frac{1}{100}$;
- non aiuta gb (però i dati sono pochi);
- accesso alla tabella (si ma solo 10^3 tuple);

SOLUZIONE 1 (indice su tipoInfrazione):

```
1 create index myindex2 on multa(tipoInfrazione);
```

PIANO DI ESECUZIONE:

- select statement;
- gb sort;
- access by rowid;
- index range scan on myindex2;

SOLUZIONE 2 (indice coprente):

```
1 create index myindex3 on multa(tipoInfrazione, data);
```

`tipoInfrazione` va per primo perchè è l'attributo più selettivo.

PIANO DI ESECUZIONE:

- select statement;
- gb sort;
- fast full index scan on myindex3(i data sono 10^5 , si devono ordinare solo 100 record);

Example 13.3

- Sono date le relazioni seguenti (le chiavi primarie sono sottolineate):
 - `STUDENTI(Matricola, Nome, Cognome, DataDiNascita)`
 - `ESAMI(Matricola, CodC, Data, Voto)`
- Si ipotizzino le seguenti cardinalità per le tabelle:
 - $\text{card}(\text{STUDENTI}) \approx 10^4$ tuple
 - $\text{card}(\text{ESAMI}) \approx 3 * 10^5$ tuple
 - $\text{MIN}(\text{ESAMI.Voto}) = 1$
 - $\text{MAX}(\text{ESAMI.Voto}) = 30$
 - **Fattore riduzione $\text{AVG}(\text{Voto}) \geq 26$ pari a 1/50**

Figure 15: Ex Studenti

```

1 select e.matricaola, name, avg(voto)
2 from esami e, studenti s
3 where e.matricola = s.matricola and voto >= 18
4 group by e.matricola, nome
5 having avg(voto) >= 26
6 order by e.matricola;

```

QUERY TREE:

$$\pi_{\text{matricola}, \text{name}, \text{avg}(\text{voto})} \text{ sort } \sigma_{\text{avg}(\text{voto}) \geq 26} \text{ gb}_{\text{matricola}, \text{name}} (\sigma_{\text{voto} \geq 18}(\text{esami}) \bowtie_{\text{matricola}} \text{studenti})$$

13.8 Concurrency Control

Grazie al concurrency control si cerca di massimizzare il throughput del numero di transazioni. I sistemi della gestione concorrente sono gestite da uno scheduler, che

riceve operazioni di lettura e scrittura. Esistono una serie di problemi di concorrenza, detti **anomalie**:

- **lost update:** (bot = begin of transaction)

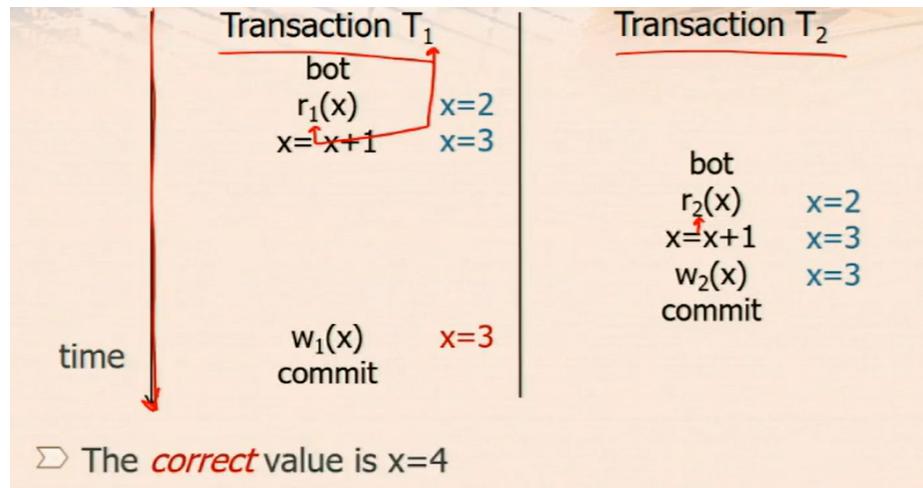


Figure 16: Lost Update

- **dirty read:**

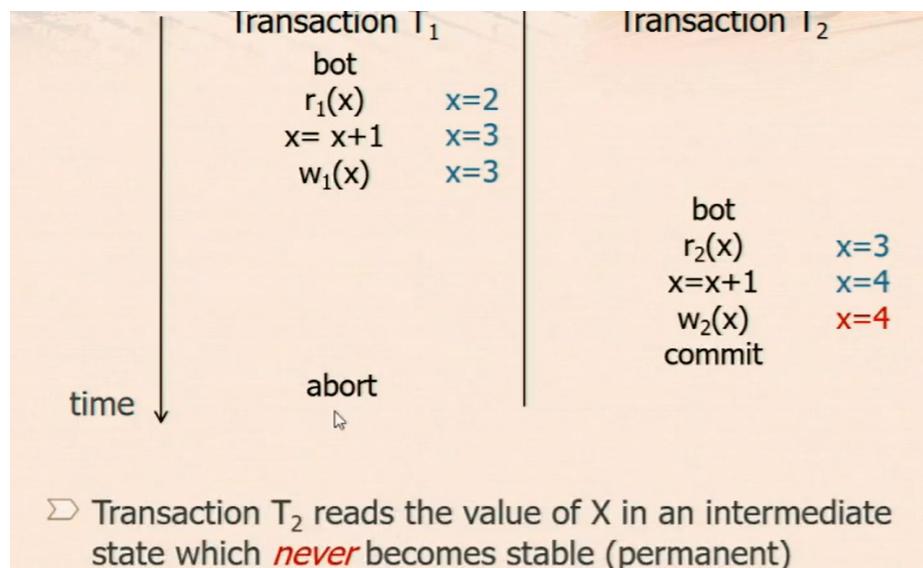


Figure 17: Dirty Read

- **inconsistent read:**

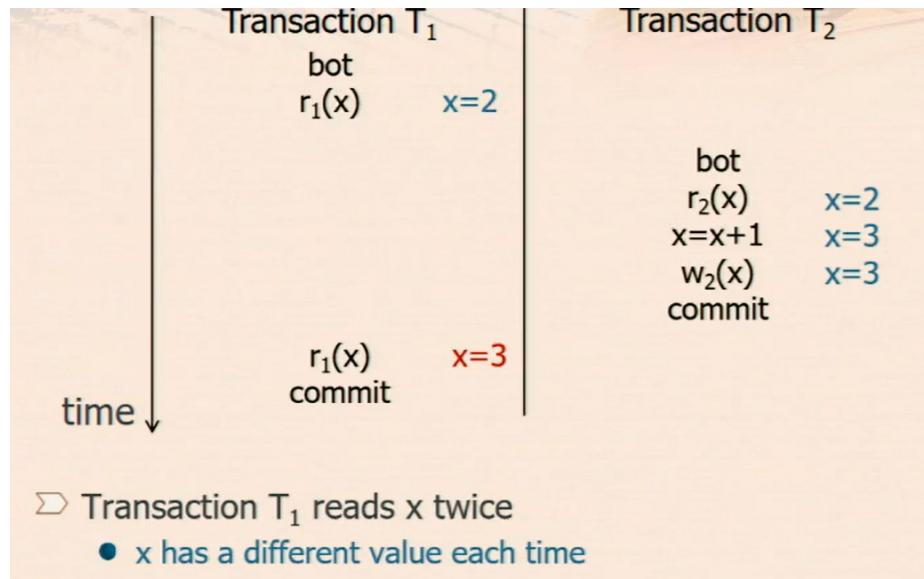


Figure 18: Inconsisten Read

- ghost update (a):

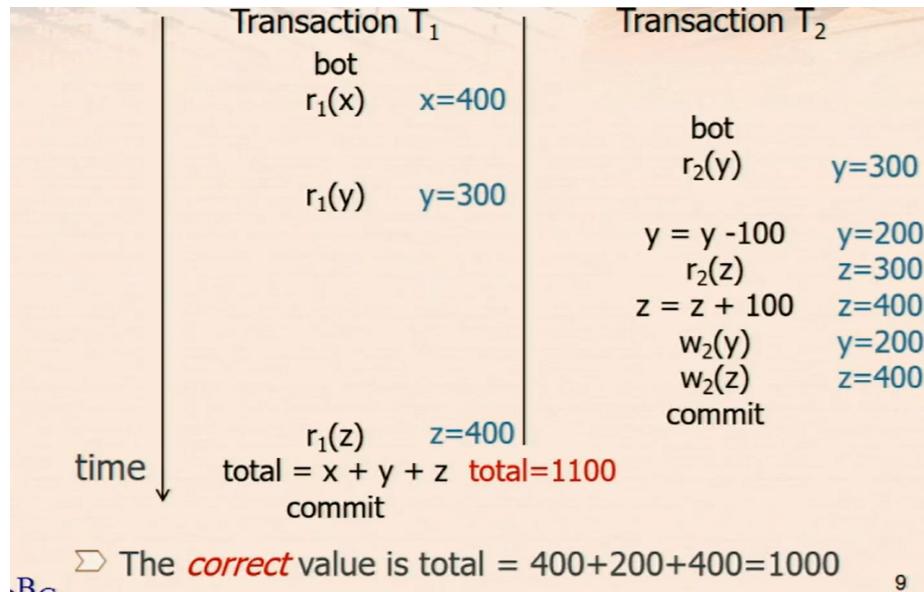


Figure 19: Ghost Read

- **ghost update (b):** come di tipo (a) però viene usato un costrutto aggregato in lettura;

Lo scheduler deve decidere se la schedulazione che arriva potrebbe causare dei problemi, per il momento rimuoviamo il problema del dirty read (l'ipotesi è che tutte le

transazioni vanno a boun fine).

Si parla di **schedule seriale**, quando le transazioni vengono eseguite fino alla fine, senza mescolare operazioni di transazioni diverse. Si può dimostrare che quando uno schedule con delle operazione intercalate è equivalente ad uno schedule seriale, allora si dice che questo schedule è **serializzabile**. Per avere uno scheduler efficiente non si cerca ricostruire in modo perfetto uno scheduling ma il problema viene diviso in classi di equivalenza. Esistono diverse classi di equivalenza:

- **view equivalence**
- **conflict equivalence**
- **2 phase locking**

13.8.1 View Equivalence

In questo caso vengono definiti due insiemi, **reads-from**, si ha una lettura $r_i(x)$ preceduta da $w_j(x)$, e nessun'altra scrittura in mezzo, ed un insieme **final write**, sono le ultime scritture che avvengono sui dati, dopo non vengono più scritte; si dice che due schedule sono **equivalenti** se hanno gli stessi set, se lo schedule seriale è equivalente allo schedule intercalato allora si dice che è **view serializable**. Questa classe di equivalenze riconosce il lost update, l'inconsistency read ed il ghost update (a). Questo problema è di classe NP, il motivo è che si devono provare tutti gli ordinamenti, roccogliendo una classe molto ampia di casistiche.

13.8.2 Conflict Equivalence

Per questa tipologia di equivalenza vanno definite le azioni conflittuali **appartengono a transazioni diverse e operano sulla stesso dato**, le azioni sono:

- read-write/write-read;
- write-write;

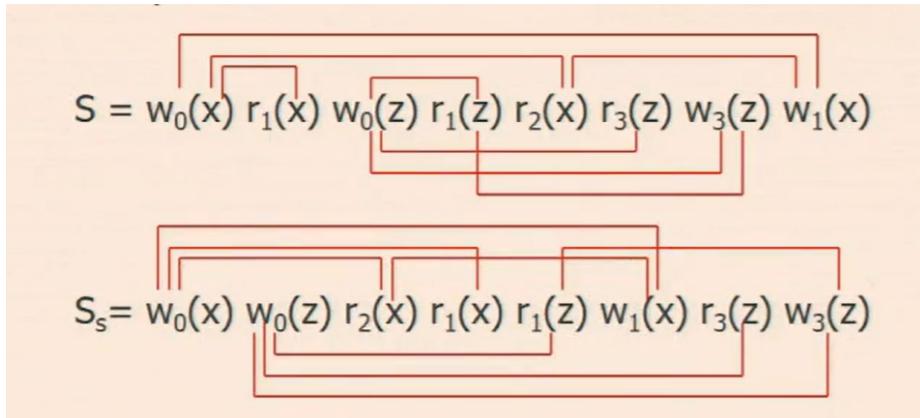


Figure 20: Conflict Equivalence

Per migliorare questo metodo si utilizza un **grafo dei conflitti**, ogni nodo rappresenta una transazione, si mette un arco orientato per ogni conflitto che si trova, se il grafo dei conflitti è aciclico allora lo schedule **conflict serializable**, altrimenti no.

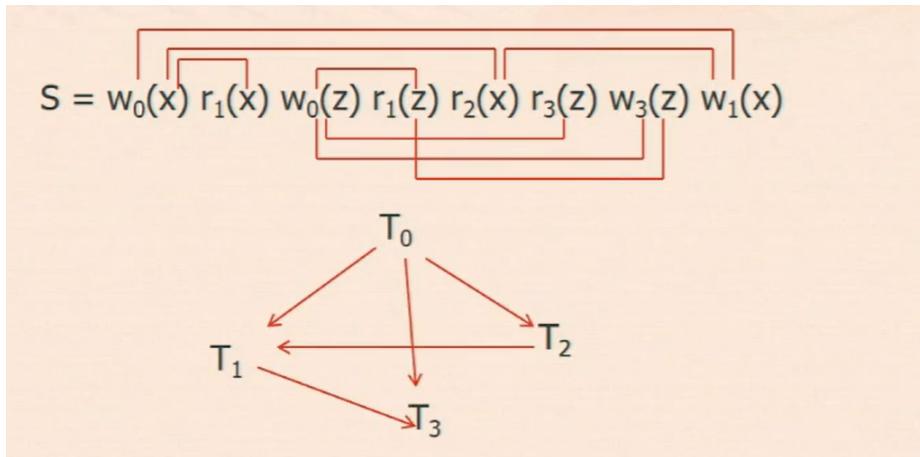


Figure 21: Grafo Dei Conflitti

13.8.3 2 Phase Locking

Qusto metodo è molto utilizzata, si utilizza un **lock**, questo lock viene usato per le letture e per le scritture. I lock sono condivisi dalle transazioni. Lo scheduler diventa in questa situazione un **lock manager** decindendo se togliere o dare dei lock in base allo stato del sistema, una volta che la transazione riceve il lock allora può accedere alla pagina, altrimenti aspetta il rilascio del lock in possesso di un'altra transazione. Il lock manager guarda una tabella dei lock per decidere quale lock dare e la tabella dei conflitti, dove vengono salvate le richieste.

Il lock manager controlla il valore dei lock per decidere cosa fare quando una transazione ne richiede uno. Nei db si utilizzano i **2 phase locking**, esiste una prima

fase detta **growing phase**, dove si acquisiscono i lock, ed una fase detta **shrinking phase**, questo si inizia a rilasciare i lock non è più possibile acquisirne di nuovi fino allo svuotarsi completo.

Su utilizziamo il **strict 2 phase locking**, ovvero abilitare il drop dei lock solo dopo il commit o l'abort di un transazione, si risolve anche il problema del **dirty read**.

Le primitivi di lock sono:

- R-lock(T, x, ErrorCode, TimeOut);
- W-lock(T, x, ErrorCode, TimeOut);
- UnLock(T, x);

Un transazione chiede un lock, il db controlla la tabella dei conflitti, se è disponibile allora il lock gli viene assegnato, se la richiesta non può essere soddisfatta, allora la transazione viene messa in una coda, sarà ripescata quando la risorsa sarà disponibile.

13.8.4 Locking Gerarchico

Si può decidere la granularità del locking:

- l'intera tabella;
- gruppi di tuple (frammenti);
- l'intera pagina;
- la singola tupla;
- il singolo attributo;

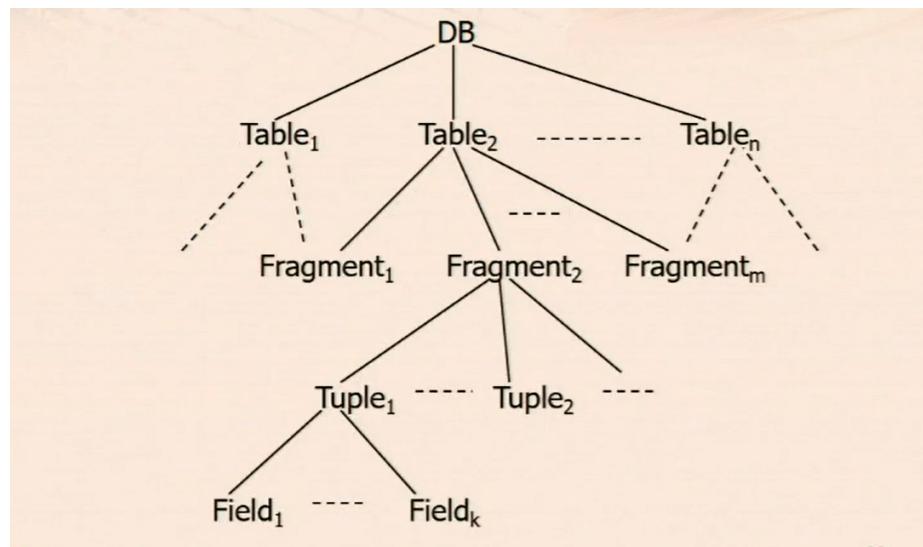


Figure 22: Gerarchia Dei Lock

Per creare questo tipo di lock vengono prese le primitivi dei lock con in aggiunta una **intenzione**.

- Shared Lock (SL) = Read Lock;
- Exclusive Lock (XL) = Write Lock;

I lock vengono assegnati a partire dalla radice dell'albero, mentre vengono rilasciati dalle foglie, se si vuole richiedere un SL (Shared Lock) o un ISL (Intention SL) su un nodo, una transazione devo possedere un ISL o un IXL (Intention Exclusive Lock) del nodo padre.

	Resource State				
Request	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

66

Figure 23: Compatibility Matrix

13.8.5 Predicate Locking

Il 2 phase locking non riesce a bloccare il ghost update (b), il predicate locking risolve questo problema **mettendo un lock sulle tuple che soddisfano un predicato**. Solitamente è presente un indice per quel predicato, ed il problema del predicate locking bloccando proprio l'indice, in questo modo non viene bloccata l'intera tabella.

13.8.6 Locking in SQL

In sql si possono definire i tipi di transazione:

- read-write (default);
- read only;
- livelli di isolamento;

I livelli di isolamento sono:

- **serializable**: include il predicate locking;
- **repeatable read**: strict 2pl;
- **read committed**: non più 2pl, i lock in scrittura vengono rilasciati dopo il commit;
- **read uncommitted**: non 2pl, si legge senza acquisire il lock;

La sintassi è:

```

1 set transaction
2 [isolation level <isolation_level>]
3 [read only]
4 [read write]
```

13.8.7 Deadlock

Per risolvere con certezza il problema del deadlock è l'uso di un timeout, alla termine di questo timeout la transazione viene uccisa. Un modo di prevenire il deadlock è il **2pl pessimistico**, ovvero richiedere a priori tutti i lock di cui la transazione ha bisogno, un altro metodo è l'**analisi del timestamp**, quando due transazioni richiede un lock si dà la precedenza alla transazioni più giovani. Si può anche utilizzare un **grafo delle attese** (se è ciclico c'è un deadlock), solitamente viene utilizzato in pochi casi di db distribuiti, il motivo è che questa tecnica è molto costosa.

13.9 Reliability Manager

La gestione dall'affidabilità garantisce due proprietà, l'**atomicità** e la **durabilità**. Il manager deve gestire delle letture e scritture ulteriori per garantire la correttezza delle operazioni fatte sul db, queste informazioni sono contenute nei **file di log**, che si trovano nella **memoria stabile**. La memoria stabile si tratta di un memoria che non può subire guasti (non realistico), questo si ottiene con della ridondanza, supponiamo che i guasti del db non hanno effetto sulla memoria stabile.

I log sono dei file sequenziali che registrano le attività delle transazioni, scritti in modo efficiente. Esistono dei delimitatori delle transazioni:

- **begin** B(T);
- **commit** C(T);
- **abort** A(T);

Le modificazioni sono (O = object, AS = after state, BS = before state):

- insert I(T,O,AS);
- delete D(T,O,BS);
- update U(T,O,BS,AS);

Fare **UNDO** di un oggetto, vuol dire rimuovere quell'oggetto dalla base dati, il **REDO** è rifare l'azione sull'oggetto. Su queste operazioni deve valere la proprietà di **idempotenza**:

$$\text{undo}(\text{undo}(action)) = \text{undo}(action)$$

Il gestore richiede il **checkpoint**, il cui unico obiettivo è quello di velocizzare l'operazione di recovery, ovvero il punto in cui viene scritto il db.

Il **DUMP** crea una copia fisica dell'intero db, quando si termina l'operazione si scrive nei log, e dal quel punto è possibile fare un recovery in caso di guasti.

Esistono dei protocolli per la gestione del log:

- **Write Ahead Log (WAL)**, è un protocollo che prima di scrivere sul db tutta l'informazione viene prima scritta nel log il BS, e poi viene fatta l'operazione.
- **Commit Precedence**, prima di fare commit si scrive prima valore dell'AS e poi viene fatto il commit.

Per questi due protocolli il BS e l'AS vengono scritti insieme. Il log viene scritto in modo **sincrono** quando un pagina viene scaricata su disco (force) o quando viene fatto un commit, in modo **asincrono** quando si fa un dump del db. Se nei log non si trova il record del commit (in caso di guasto), la transazione deve essere ripetuta.

A fronte di guasto, che possono essere di **sistema** (sistema operativo) o **media** (supporti fisici). Esiste il modello del **fail-stop**, quando si cerca di fare del recovery, viene continuato a fare fino a quando non si è sicuri di avere dei dati consistenti.

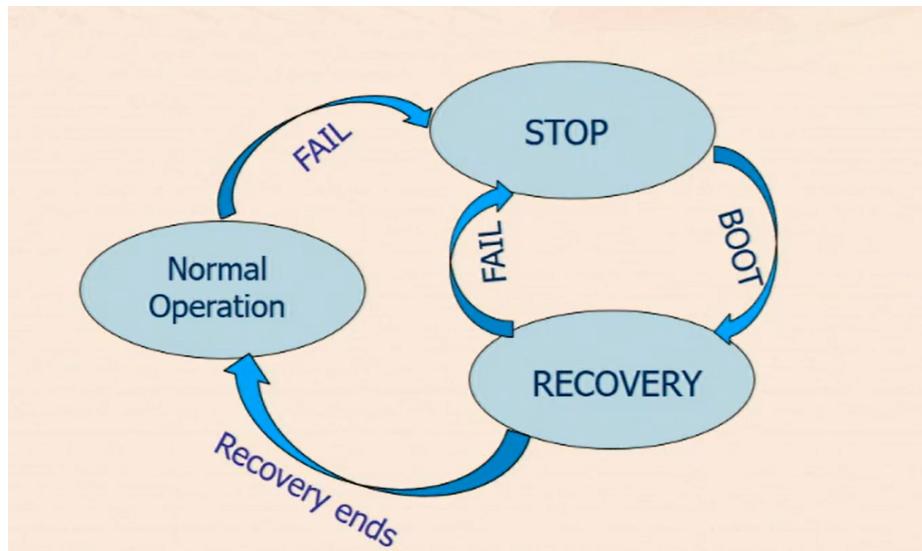


Figure 24: Fail Stop

Esistono due tipi di restart:

- **warm restart**: system failure;
- **cold restart**: media failure;

13.9.1 Warm restart

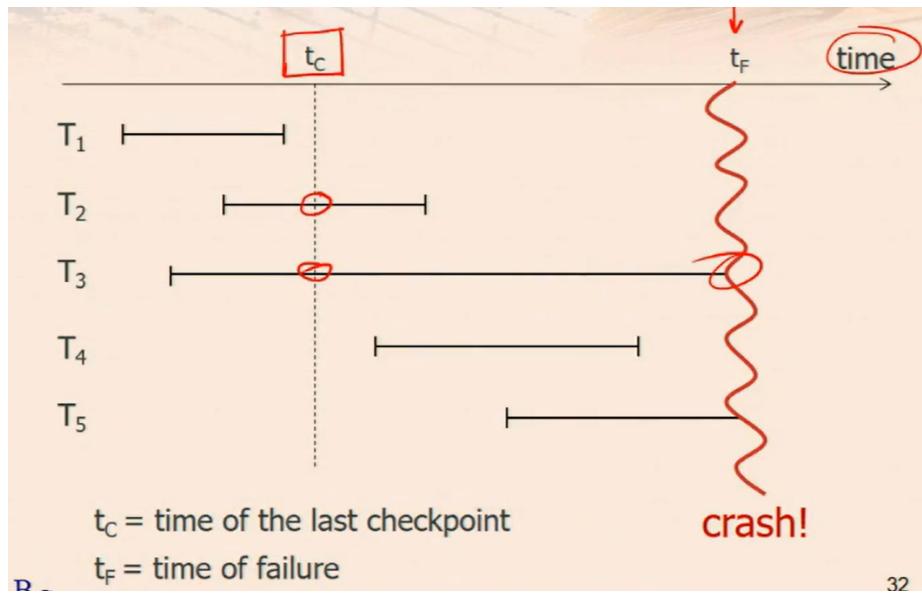


Figure 25: Warm Restart

In questo caso:

- 1: non bisogno di operazioni di recovery;
- 2, 4: hanno bisogno di redo;
- 3, 5: hanno bisogno di undo;

L'algoritmo di ripristino funziona nel segutne modo:

1. legge il log al contrario e cerca il checkpoint più recente;
2. inizia a cercare le transazioni da cui fare undo e redo;
3. crea due liste per undo e redo e mette nella lista di undo tutte le transazioni trovate che si trovano nel checkpoint;
4. legge il log dal checkpoint fino all'ultimo log e per ogni commit che trova sposta la transazione dalla lista di undo alla lista di redo, e per ogni begin che trova sposta la lista nella lista degli undo;
5. si parte dall'ultima entra e percorrendo il log si fanno tutte le azioni di undo;
6. si parte dall'operazione più vecchia di tutte le transazioni presenti nella lista dei redo e andando in avanti si fanno tutte le operazioni di redo;

13.9.2 Cold Restart

Una parte del db è andato perduto per un guasto fisico, di parte dall'ultimo record di dump e si fanno tutte le operazioni transazionionali presenti nel log e al termine si fa un warm restart per ripristinare lo stato consistente dei dati.

14 DBMS Distribuiti

L'obiettivo dei db distribuiti è quello di distribuire dati e computazione su macchine diverse. I vari tipi di architettura possono essere:

- client/server;
- database distribuito, ogni db è indipendente, le proprietà acid devono essere valide anche con i dati distribuiti;
- db replicati, detti server di replicazione;
- architetture parallele;

Nello scenario dei db distribuiti si ha la situazione in cui un client potrebbe accedere a più nodi, ed ogni nodo opera in maniera indipendente. Distribuire i dati permette di avere una localizzazione, maggior disponibilità di dati e maggiore scalabilità, oltre a risolvere il problema di single point of failure.

I dati possono essere frammentati tra i diversi db, può avvenire a livello di tabella come proiezione, selezione, o un mix. Quando si applica una selezione per ricostruire la tabella basta un'operazione di unione. Se si frammenta a modo verticale la chiave primaria deve essere necessariamente aggiunta, per ricostruire la tabella basta fare un join. Quando si decide di frammentare la relazione, la tabella originale non esisterà più ma sarà possibile ricostruirla opportunamente attraverso degli schema, ognuno di questi frammenti sarà gestito in modo indipendente da ogni db su cui si trova. Avere dei dati frammentati permette di avere una ridondanza dei dati, avendo un costo maggiore per operazioni di aggiornamento.

Esistono vari livelli di trasparenza per i frammenti quando si scrivono delle query sql per sapere da dove prendere i dati:

- **fragmentation transparency**: la query viene scritta come se la tabella su un unico db;
- **allocation transparency**: la query viene scritta essendo a conoscenza dei frammenti, senza sapere però su quale server si trovano;
- **language transparency**: la query viene scritta richiedendo i dati da un server specifico;

Le transazioni possono essere classificate.

- **remote request**: solo lettura;
- **remote transaction**: eseguire un comando qualunque su un singolo server;
- **distributed transaction**: si esegue qualsiasi comando sql, ogni comando è riferito ad un singolo nodo, in questo caso la gestione delle transazioni deve essere più robusto (2 phase commit);

- **distributed request:** ogni comando può far riferimento a dati distribuiti su server diversi, mi garantisce la trasparenza di frammentazione;

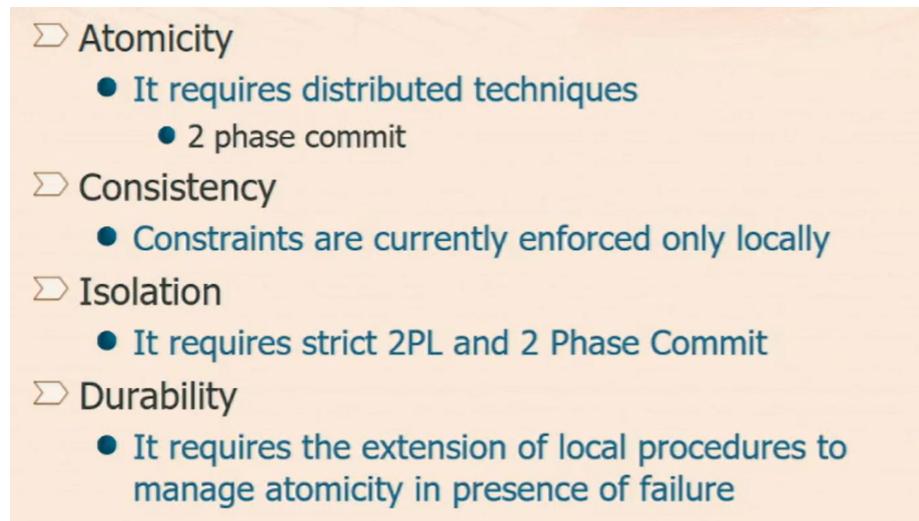


Figure 26: Acid Properties In Distributed Dbms

...

15 NoSQL

Le caratteristiche principali sono dei NoSql sono:

- nessuna possibilità di fare join;
- nessuno schema;
- scalabilità orizzontale (aggiungere molte cpu in parallelo);

Nei db non-relazionali non esistono tabelle ma documenti, o key-value, graph based, columnar storage.

I tipi di db nosql sono:

- **key-value pair:** sono veloci ed efficienti, tutti i dati vengono messi in cache;
- **column oriented:** simili ai datawarehouse;
- **graph db:** usati quando dei dati hanno delle relazioni tra di loro;
- **documento:** ogni documento possiede degli attributi chiave valore, i documenti possono essere nestati tra di loro;

15.1 CouchDB

CouchDB (cluster of unreliable commodity hardware). I punti principali dell'architettura di couchdb sono:

- possibilità di fare **mapreduce**: ogni nodo può fare in modo distribuito del mapping e del reducing su uno stream di dati;
- **replicazione** dei dati nei vari nodi;
- **distribuzione** del sistema;
- interrogazioni attraverso un API HTTP;

15.1.1 MapReduce

Esempio di mapReduce:

```

1 // Map function
2 function(doc) {
3     key = doc.matricola
4     value = [doc.mark, doc.cfu]
5     emit(key, value);
6 }
7
8 // Reduce function
9 function(key, values) {
10    S = sum([ X*Y for X,Y in values])
11    N = sum([ Y for X,Y in values])
12    AVG = S/N
13    return AVG;

```

15.1.2 Replication

Un approccio alla replicazione è il **master-slave**, dove un unico server è il master, che prende tutte le scritture e gli update, tutti gli altri nodi sono degli slave, da cui è possibile solo leggere i dati, i cambiamenti possono essere propagati in maniera sincrona (simile al 2PC), oppure l'asynchronous replication dove il master fa un commit locale, ogni slave fa il fetch indipendenti dei nuovi dati.

15.1.3 Distribuzione

Le caratteristiche di un sistema sono:

- **Consistency**;
- **Availability**;
- **Partizionamento**;

Da qui nasce il **teorema CAP**.

Theorem 15.1 – CAP

In ogni istante è possibile ottenere solo due caratteristiche contemporaneamente.

- CA: si ricade nel db singolo;
- CP: quando si fa il 2PC;
- AP: non ci importa della global consistency;

Le combinazioni possono essere intercambiate in tempi diversi in modo continuo. Infatti nel mondo non relazionale esiste la caratteristica **BASE**, (al contrario di ACID):

- Basically Available;
- Soft state;
- Eventually consistent;

Per la risoluzione dei conflitti si possono utilizzare:

- revision: come con il version control, quando si fetcha una revision si prende quella con il numero più basso e con l'hash ordinato;
-

Nel mondo non relazionale non esistono le transazioni, più operazioni infatti non possono essere rese atomiche, però le operazioni su un singolo documento sono transazionali, quindi un modo di procedere è quello di modellare il documento come una transazione.

15.2 MongoDB

In MongoDB:

- table = collection;
- record = document;
- column = field;

Esempio di query su mongoDB:

```

1 db.<collection name>.find( {<conditions>} , {<fields of interest>}) ;
2
3 // mongoDB
4 db.people.find();
5
6 // sql
7 select *
8 from people;

```

Filtraggio:

```

1 db.people.find({age:55});
2
3 db.people.find({}, {user_id: 1, status: 1});
4
5 db.people.find({age: {$gt: 25, $lte: 50} });
6
7 db.people.find({$or: [{status: "A"}, {age: 55}] });
8
9 db.people.find({ status: {$in: ["A", "B"]} });

```

In mongoDB per recuperare altri documenti si possono usare l'**objectId** ed il **DBRef**.

Si possono anche usare degli operatori aggreganti con il metodo `.aggregate()`.

```

1 db.people.aggregate([
2   { '$group': {
3     '_id: null,
4     mytotal: { '$sum: "$age" },
5     mycount: { '$sum: 1 }
6   }
7 ]]);

```

Per utilizzare le coordinate spaziali si utilizza GeoJSON:

```

1 {
2   _id: "...",
3   location: {
4     type: "Point",
5     coordinates: [
6       45.324, // lat
7       52.123 // lng
8     ]
9   }
10 }

```

Poi si crea un indice sul campo:

```

1 db.collection.createIndex({location: "2dsphere"});
2
3 db.collection.find({location: {
4   '$near: {
5     type: "Point",
6     coordinates: [34.353, 4.535]
7   },

```

```
8      \$maxDistance: 435,  
9      \$minDistance: 43  
10     }  
11  });
```

I risultati vengono restituti in ordine dal più vicino al più lontano.