PER ALTRI APPUNTI CONSULTARE IL SITO:

https://luigi-v.github.io/Appunti_Universita/

Per altri appunti consultare il sito: *https://luigi-v.github.io/Appunti_Universita/*

By *Vollono Luigi*

BASI DI DATI 2 - DOMANDE ORALI

1. DIPENDENZE FUNZIONALI E NORMALIZZAZIONE

- ***Linee guida e misure informali di bontà del database + cosa sono le tuple spurie:***

Si ha il bisogno di misurare formalmente il perché un raggruppamento di attributi in uno schema di relazione possa essere meglio di un altro, misurando la "bontà" (qualità) del progetto. Esistono 4 ***misure informali***:

1. ***Semantica degli attributi***, ogni volta che si raggruppano degli attributi per formare uno schema di relazione, si suppone che ad essi sia associato un certo significato. Ogni relazione può essere interpretata come un insieme di fatti o asserzioni.
   LINEA GUIDA 1: Si progetti ogni schema di relazione in modo tale che sia semplice spiegarne il significato. Non si uniscano attributi provenienti da più tipi di entità e tipi di associazione in un'unica relazione. Se uno schema di relazione corrisponde a un solo tipo di entità o a un solo tipo di associazione, il suo significato tende ad essere chiaro.
2. ***Riduzione dei valori ridondanti nelle tuple***, uno scopo della progettazione di schemi è quello di ridurre al minimo lo spazio di memoria occupato dalle relazioni di base. Il raggruppamento di attributi in schemi di relazione ha un effetto significativo sullo spazio di memoria. Un altro problema che si presenta sono le *anomalie di aggiornamento*, che possono essere classificate in:
   - *Anomalia di inserimento*, per inserire una nuova tupla occorre inserire i valori degli attributi in tutti i campi della relazione, altrimenti valori nulli.
   - *Anomalia di cancellazione*, se si cancella l'ultima tupla in una relazione, l'informazione che si riverisce ad un'altra relazione non sarà più presente.
   - *Anomalia di modifica*, se si cambia un attributo in una relazione occorre aggiornare le tuple di tutte le relazioni collegate ad esso.

   LINEA GUIDA 2: Si progettino gli schemi di relazione di base in modo che nelle relazioni non siano presenti anomalie. Se sono presenti delle anomalie, le si rilevi chiaramente e ci si assicuri che i programmi che aggiornano la base di dati operino correttamente.
3. ***Riduzione del numero di valori nulli nelle tuple***, è possibile che vengano raggruppati numerosi attributi a formare una grossa relazione, se molti attributi non riguardano tutte le tuple della relazione, quelle tuple saranno nulle. Ciò può dar luogo ad uno spreco di memoria e problemi di comprensione del significato. Inoltre, i valori nulli possono avere più interpretazioni.
   LINEA GUIDA 3: Per quanto possibile, si eviti di porre in una relazione di base attributi i cui valori possono essere frequentemente null. Se i valori null sono inevitabili, ci si assicuri che essi si presentino solo in casi eccezionali e che non riguardino una maggioranza di tuple nella relazione.
4. ***Impossibilità di generare tuple spurie***, se viene separata una relazione in relazioni più piccole, e si tenta un'operazione di JOIN NATURALE sulle due relazioni, il risultato del join può produrre molte più tuple rispetto all'originaria popolazione. Tuple aggiuntive sono dette ***tuple spurie*** perché rappresentano un'informazione spuria o sbagliata che non è valida.
   LINEA GUIDA 4: Si progettino schemi di relazione in modo tale che essi possano essere riuniti, tramite JOIN, con condizioni di uguaglianza su attributi che sono o chiavi primarie o chiavi esterne in modo da garantire che non vengano generate tuple spurie. Non si abbiano relazioni che contengono attributi di accoppiamento diversi dalle combinazioni chiave esterna – chiave primaria. Se relazioni di questo tipo sono inevitabili, non si effettui su di esse un'operazione di join sulla base di questi attributi, perché il join può produrre tuple spurie.

- ***Cos'è una dipendenza funzionale:***

Una ***dipendenza funzionale*** è un vincolo tra due insiemi di attributi della base di dati. Formalmente, una dipendenza funzionale (DF), indicata con X→Y, tra due insiemi di attributi X e Y che siano sottoinsiemi di R specifica un vincolo sulle tuple che possono formare uno stato di relazione r di R. Il vincolo è che per ogni coppia di tuple t_1 e t_2 in r per le quali è $t_1[X]=t_2[X]$, si deve avere anche $t_1[Y]=t_2[Y]$. Ciò significa che i valori della componente Y di una tupla in r dipendono da, o sono determinati da, i valori della componente X.

- ***Regole di inferenza di Armstrong (le regole di inferenza di Armstrong sono assiomi o teoremi?):***

Tipicamente non possono essere dedotte tutte le DF semanticamente, alcune possono essere inferite o dedotte dalle DF in F, ed esistono 6 regole:

- ***RI1 (regola riflessiva)***: Afferma che un insieme di attributi determina sempre sé stesso o uno qualsiasi dei suoi sottoinsiemi;
- ***RI2 (regola di arricchimento)***: Sostiene che aggiungendo lo stesso insieme di attributi alla parte sinistra e destra di una dipendenza si ottiene un'altra dipendenza valida;
- ***RI3 (regola transitiva)***: Le dipendenze funzionali sono transitive;
- ***RI4 (regola di decomposizione/proiezione)***: Sostiene che si può rimuovere attributi dalla parte destra di una dipendenza, decomponendo la DF;
- ***RI5 (regola di unione/additiva)***: Consente di fare l'opposto della RI4, è possibile combinare un insieme di dipendenze;
- ***RI6 (regola pseudo transitiva)***.

Le regole di Armstrong sono ***corrette***, ovvero che qualsiasi DF che viene prodotta con le regole è una DF a sua volta, e ***complete***, ovvero che tutte le possibili DF che si possono trovare sono determinabili utilizzando le regole.

- ***Forme normali:***

1NF	La relazione non deve avere attributi non-atomici o relazioni nidificate.	Formare nuove relazioni per ogni attributo non-atomico o relazione nidificata.
2NF	Per relazioni in cui la PK contiene più attributi, nessun attributo non-chiave deve essere funzionalmente dipendente da una parte della PK.	Decomporre e preparare una nuova relazione per ogni chiave parziale coi suoi attributi dipendenti. Assicurarsi di mantenere una relazione con la PK originale e tutti gli attributi funzionalmente dipendenti in modo completo da essa.
3NF	La relazione non deve avere un attributo non-chiave determinato funzionalmente da un altro attributo non-chiave (o da un insieme di attributi non-chiave). Cioè non deve esserci nessuna dipendenza transitiva di un attributo non-chiave dalla PK.	Decomporre e preparare una relazione che comprenda gli attributi non-chiave che determinano funzionalmente altri attributi non-chiave.

- ***Forma normale di boyce-codd (Nota: è sempre ottenibile ma a un costo, ovvero perdere delle dipendenze):***

La forma normale di Boyce e Codd (BCNF) è una forma più restrittiva della 3NF, infatti ogni relazione in BCNF è anche in 3NF, ma non viceversa.
Uno schema è in ***BCNF*** se per ogni DF su R, X->A dove X è superchiave (insieme di attributi contenente la chiave). Se esiste una DF X->Y che viola la proprietà allora si decompone la relazione che la contiene in uno schema dove Y viene tolta e si aggiunge una nuova relazione dove si ha XUY.

- ***Differenza tra 3nf generale e bcnf:***

BCNF è una forma normale in cui per ogni dipendenza funzionale non banali, la parte sinistra è superchiave, mentre 3NF è una forma normale in cui la relazione è in 2NF e ogni attributo non primo (non appartenente ad una superchiave) è non dipendente in modo *transitivo* da ogni chiave primaria.

2. ALGORITMI PER LA PROGETTAZIONE DI BASE DI DATI RELAZIONALI

- ***Copertura minimale:***

Un insieme F di dipendenze funzionali è ***minimale*** se soddisfa le seguenti condizioni:

1. Ogni dipendenza presente in F ha come parte destra un solo attributo.
2. Non è mai possibile sostituire una dipendenza X→A di F con una dipendenza Y→A, dove Y è un sottoinsieme proprio di X, e avere ancora un insieme di dipendenze equivalente a F;
3. Non è mai possibile rimuovere una dipendenza da F e avere ancora un insieme di dipendenze equivalente a F.

Algoritmo 2:

1. Porre G:=F;
2. Rimpiazzare ogni dipendenza funzionale X→ $\{A_1, A_2, \ldots, A_n\}$ in G, con n dipendenze funzionali $X \to A_1$, $X \to A_2$, …, $X \to A_n$;
3. Per ogni dipendenza funzionale X→A in G
   per ogni attributo B che è un elemento di X
   se { {G - {X → A} } ∪ { (X – {B}) → A} } è equivalente a G allora sostituire X → A con (X – {B}) → A in G;
4. Per ogni dipendenza funzionale rimanente X → A in G
   se { G – {X → A} } è equivalente a G allora rimuovere X → A da G;

- ***Algoritmi di sintesi relazionale:***

L'algoritmo crea, a partire da un insieme di dipendenze funzionali F, una decomposizione D di una relazione universale R, che conserva le dipendenze e tale che ogni relazione generata sia in 3NF. Esso garantisce solo la proprietà di conservazione delle dipendenze, non la proprietà di join senza perdita.

Algoritmo 1.1 (***algoritmo di sintesi relazionale***):

Input: Una relazione universale R e un insieme di dipendenze funzionali F sugli attributi di R.

1. Trovare una copertura minimale G di F.
2. Per ogni parte sinistra X di una dipendenza funzionale che appare in G, creare uno schema di relazione $\{X \cup A_1 \cup A_2 \cup \ldots \cup A_m\}$ in D dove $X \to A_1$, $X \to A_2$, …,$X \to A_m$ sono le sole dipendenze in G aventi X come parte sinistra.
3. Mettere in uno schema di relazione singolo tutti gli attributi rimanenti, per garantire la proprietà di ***conservazione delle dipendenze***.

Proposizione 1A: Ogni schema di relazione creato dall'Algoritmo 1 è in 3NF.

- ***Algoritmo lossless-join, dp e 3nf:***

Una semplice variazione dell'Algoritmo 1.1, porta a una decomposizione D di R con le seguenti proprietà:

- conserva le dipendenze,
- gode della proprietà di join senza perdita,
- è tale che ogni schema di relazione risultante nella decomposizione sia in 3NF.

Algoritmo 1.4:

Input: Una relazione universale R e un insieme di dipendenze funzionali F sugli attributi di R.

1. Trovare una copertura minimale G per F;
2. Per ogni parte sinistra X di una FD in G, creare uno schema di relazione in D con attributi $\{XUA_1UA_2U\ldots UA_m\}$ dove $X \to A_1$, $X \to A_2$, …, $X \to A_m$ sono le sole dipendenze in G aventi X come parte sinistra.
3. Se nessuno degli schemi di relazione in D contiene una chiave di R, creare un altro schema di relazione che contiene attributi che formano una chiave per R.

- ***Dipendenze multivalore (MVD):***

In molti casi le relazioni hanno vincoli che non possono essere specificati sotto forma di dipendenze funzionali. Le ***dipendenze multivalore*** sono una conseguenza della prima forma normale (1NF), che impedisce a un attributo in una tupla di conoscere un insieme di valori. Se vi sono due o più attributi multivalore indipendenti nello stesso schema di relazione, si ha il problema di dover ripetere ogni valore di uno degli attributi per ciascun valore dell'altro attributo. Questo vincolo viene espresso mediante una dipendenza multivalore. Quindi, dato un particolare valore di X, l'insieme di valori di Y è determinato completamente solo da X e non dipende dai valori dei restanti attributi in Z di R.

Una MVD $X \twoheadrightarrow Y$ in R è detta ***MVD banale*** (***trivial***) se: Y è un sottoinsieme di X, oppure X U Y = R.

- ***4nf:***

La quarta forma normale ***4NF*** viene violata quando una relazione ha dipendenze multivalore non volute, e quindi può essere usata per identificare e decomporre tali relazioni. Uno schema di relazione R è in 4NF rispetto a un insieme di dipendenze F (che include le dipendenze funzionali e multivalore) se, per ogni dipendenza multivalore non banale $X \twoheadrightarrow Y$ in F^+, X è una superchiave di R.

NOTA sulle MVD non-banali: Relazioni contenenti MVD non-banali tendono ad essere relazioni "tutta chiave" (la chiave è formata da tutti gli attributi).

- ***4nf perché quando progettiamo uno schema ci fermiamo a 3nf o bcnf?***

Perché si potrebbe perdere efficienza in quanto abbiamo sempre più relazioni e di conseguenza quando dobbiamo fare le query ci saranno sempre più operazioni di join da fare (molto dispendiose). Inoltre, non sempre è possibile conservare tutte le dipendenze, quindi la 4nf porta a perdita dei dati.
La 4NF è poco usata nelle applicazioni odierne, in quanto la 3NF e la BCNF già forniscono il giusto compromesso tra semplicità e qualità dei risultati.

- ***Problema 1nf che non è 4nf (in che caso dalla 1nf abbiamo problemi con la 4nf: portando fuori le mvd ma mettendole insieme non sono in 4nf):***

Perché la 1NF non permette attributi multivalore che siano essi stessi composti. Queste relazioni sono dette ***relazioni nidificate*** perché ogni tupla può avere al suo interno una relazione.

4. MEMORIZZAZIONE DI RECORD ED ORGANIZZAZIONE DEI FILE

- ***Hashing:***

Un tipo di organizzazione primaria di file è basato sull'hash, che fornisce un accesso rapido ai record sotto certe condizioni di ricerca. La condizione di ricerca deve essere una condizione di uguaglianza su un campo singolo, detto ***campo hash*** del file, che è anche un campo chiave (chiave hash). L'idea è quella di fornire una funzione hash (di randomizzazione), che è applicata al valore del campo hash di un record e fornisce l'indirizzo del blocco di disco in cui è memorizzato il record. ***Hash interno*** per i file, l'hash è implementato con una tabella hash costruita usando un vettore di record di dimensioni M, e che il campo di valori possibili per l'indice del vettore vada da 0 a M-1, si ha pertanto M slot i cui indirizzi corrispondono agli indici del vettore. Si sceglierà una funzione hash che trasforma il valore del campo hash in un intero compreso tra 0 e M-1.

Una *collisione* è quando il valore del campo hash di un record che sta per essere inserito in uno slot, quest'ultimo è già occupato. In questa situazione occorre inserire il nuovo record in un'altra posizione, questo processo è detto risoluzione delle collisioni:

- *Indirizzamento aperto*: procedendo dalla posizione occupata, si verifica in ordine le posizioni successive fino a che non si trova una posizione vuota;
- *Concatenamento*: Si mantiene una lista concatenata di record di overflow per ogni indirizzo hash. Una collisione viene risolta ponendo il nuovo record in una locazione di overflow inutilizzata, concatenandola agli altri record già presenti in quello slot.
- *Hash multiplo*: viene applicata una seconda funzione hash se la prima ha come risultato una collisione. Se risulta un'altra collisione, il programma usa l'indirizzamento aperto o applica una terza funzione hash e poi, se necessario, usa l'indirizzamento aperto.

L'hash per file su disco è detto ***hash esterno***. Lo spazio di indirizzi obiettivo è fatto di *bucket*, che è un blocco di disco o un cluster di blocchi contigui. La funzione hash mappa una chiave in un numero relativo per il bucket. Una tabella contenuta nell'header del file converte il numero del bucket nel corrispondente indirizzo di blocco di disco. Con i bucket il problema delle collisioni è meno grave perché possono essere inviati nello stesso bucket senza causare problemi. Ma se il bucket è pieno, viene utilizzata una variazione del concatenamento in cui in ogni bucket si mantiene un puntatore a una lista concatenata di record di overflow per il bucket.

5. STRUTTURA DI INDICI PER I FILE

- ***Campo chiave e campo di ordinamento:***

Esistono molti tipi di indici ordinati, come l'indice primario che è specificato sul ***campo chiave di ordinamento*** di un file ordinato di record. Il campo chiave di ordinamento è utilizzato per ordinare fisicamente i record dei file su disco e tutti i record hanno un valore univoco per quel campo. Se il ***campo di ordinamento*** non è un ***campo chiave***, cioè se molti record nel file possono avere lo stesso valore del campo di ordinamento, può essere usato un altro tipo di indice, chiamato indice di cluster. Si noti che un file può avere al massimo un campo di ordinamento fisico, quindi può avere al massimo un indice primario oppure un indice di cluster, ma non entrambi. Su qualsiasi campo non di ordinamento di un file può essere specificato un terzo tipo di indice, detto indice secondario. Un file può avere molti indici secondari oltre al suo metodo di accesso primario.

- ***indice denso e indice sparso:***

Un indice denso contiene una voce per ogni valore della chiave di ricerca (e quindi ogni record) nel file di dati, mentre un indice sparso (o non denso) contiene voci solo per alcuni valori di ricerca.

- ***Cosa e quali sono gli Indici + indici primari, cosa sono a cosa servono e come sono strutturati + Indice multilivello (perché indice primario su livelli successivi?):***

Gli ***indici*** sono strutture ausiliarie di accesso, essi velocizzano la ricerca dei record, fornendo percorsi di accesso secondari, e non influenzano la posizione fisica dei record sul disco. Permettono un accesso efficace sulla base di campi d'indicizzazione che vengono utilizzati per costruire l'indice. Esistono diversi indici:

- ***Indici ordinati***, simile all'indice di un libro, definiti su un unico campo, chiamato campo di indicizzazione. L'indice memorizza ogni valore del campo di indicizzazione corredandolo di un elenco di puntatori a tutti i blocchi del disco che contengono record con quel valore del campo.
- ***Indici primari***, esiste una *voce* (o record dell'indice). Ogni voce è composta da due campi che contengono il valore del campo della chiave primaria del primo record del blocco e un puntatore al corrispondente blocco su disco. Il numero complessivo di voci dell'indice è uguale al numero di blocchi su disco nel file di dati ordinato. Il primo record in ciascun blocco del file di dati è chiamato record àncora del blocco. L'indice primario è un indice *non denso*, poiché include una voce per ogni blocco del file di dati piuttosto che per ogni valore di ricerca o per ogni record.
- ***Indice di cluster***, se i record di un file sono ordinati fisicamente rispetto a un campo che non è chiave, cioè che non ha un valore distinto per ciascun record, quel campo è chiamato campo di raggruppamento. L'indice di cluster velocizza il recupero dei record che hanno lo stesso valore del campo di raggruppamento. Questo indice differisce da un indice primario, che richiede che il campo di ordinamento del file di dati abbia un valore distinto per ogni record. Anche l'indice di cluster è un file ordinato con due campi, il primo è dello stesso tipo del campo di raggruppamento del file di dati, mentre il secondo è un puntatore a un blocco. Un indice di cluster è un altro esempio di indice *non denso*.
- ***Indice secondario***, è un file ordinato con due campi, il primo è dello stesso tipo di dati di un campo del file di dati, che non viene utilizzato per effettuare l'ordinamento del file di dati, chiamato campo di indicizzazione, il secondo campo è un puntatore a record. Vi possono essere più indici secondari (e quindi campi di indicizzazione) per lo stesso file. Vi è una voce dell'indice per *ciascun record* del file di dati, la quale contiene il valore della *chiave secondaria* del record e un puntatore al blocco in cui il record è memorizzato. Questo indice è *denso*.
- ***Indice multilivello***, considera l'indice di primo livello (o livello base) di un indice multilivello, come un file ordinato con un valore distinto. A questo punto è possibile creare un indice primario per l'indice di primo livello, questo è detto indice di secondo livello. Poiché il secondo livello è un indice primario, si possono utilizzare i punti àncora dei blocchi in modo che il secondo livello contenga una voce per ciascun blocco del primo livello. Vengono creati più livelli successivi perché, mentre per gli altri indici la ricerca viene ridotta di un fattore pari a 2, negli indici multilivello viene ridotta ad ogni passo la parte dell'indice di un cosiddetto *fattore di blocco* dell'indice, che è maggiore di 2, riducendo lo spazio di ricerca molto più velocemente.

6. GESTIONE DELLE TRANSAZIONI

- ***Gestione delle transazioni:***

I ***sistemi di gestione delle transazioni*** sono sistemi con grandi basi di dati e centinaia di utenti che eseguono transazioni contemporaneamente. Più utenti possono accedere al database simultaneamente grazie al concetto di *multiprogrammazione*, che consente ad un computer di elaborare più transazioni alla volta. Quando si ha una sola CPU, si elabora un processo alla volta, infatti, l'illusione di avere più programmi che vengono eseguiti contemporaneamente è fornita dai sistemi operativi multiprogrammati, che eseguono alcune istruzioni da un programma, per poi sospenderlo ed eseguire altre istruzioni da altri programmi e così via. Quando un programma viene riattivato, esso riparte dal punto in cui era stato sospeso. Su sistemi monoprocessore, l'esecuzione concorrente dei programmi è quindi intervallata (*interleaved*), questo ha luogo quando un programma effettua operazioni di I/O che lasciano la CPU inattiva. Su sistemi *multiprocessore* l'esecuzione dei programmi avviene realmente *in parallelo*.

- ***Letture fantasma nelle implementazioni sql:***

Le ***letture fantasma*** si verificano quando:

1. La Transazione A legge tutte le righe che soddisfano una clausola WHERE su una query SQL.
2. La Transazione B inserisce una riga aggiuntiva che soddisfa la clausola WHERE.
3. La Transazione A rivaluta la condizione WHERE e raccoglie la riga aggiuntiva.

- ***Problemi per cui serve il controllo della concorrenza (es aggiornamento perso etc):***

Il controllo della concorrenza riguarda principalmente i comandi di accesso al database in una transazione. Transazioni inviate da più utenti, che possono accedere e aggiornare gli elementi del database, vengono eseguite in modo concorrente. Se l'esecuzione concorrente non è controllata, si possono avere problemi di *database inconsistente* (due dati che rappresentano la stessa informazione hanno valori diversi). Possibili ***problemi*** sono:

- ***Problema dell'aggiornamento perso***, supponendo che due transazioni siano avviate contemporaneamente e che le loro operazioni siano interleaved (interfogliate), se la seconda transazione legge il valore di una variabile prima che la prima transazione salvi il suo valore modificato, risulterà che l'aggiornamento della prima transazione venga perso;
- ***Problema dell'aggiornamento temporaneo*** *(o* ***lettura sporca****)*, una transazione aggiorna un elemento ma poi essa fallisce, non riuscendo a salvare tale aggiornamento, ma l'elemento aggiornato è letto da un'altra transazione prima che esso sia riportato al suo valore originario;
- ***Problema della totalizzazione scorretta***, se una transazione sta calcolando una funzione di aggregazione su un certo insieme di record, mentre altre transazioni stanno aggiornando alcuni di quei record, la funzione può calcolare alcuni valori prima dell'aggiornamento ed altro dopo. Quindi il risultato dell'aggregazione risulterà sbagliato, in quanto legge inizialmente valori non aggiornati;
- ***Problema delle letture non ripetibili***, tale problema avviene se una transazione legge due volte lo stesso item, ma tra le due letture un'altra transazione ne ha modificato il valore, leggendola una seconda volta si ottengono nuovi dati.

- ***Ciclo di vita delle transazioni:***

Una ***transazione*** è un'unità atomica di lavoro che, o è completata nella sua interezza o è integralmente annullata. Per motivi di recovery, il sistema deve tenere traccia dell'inizio, fine o abort di ogni transazione. Il manager di recovery tiene quindi traccia delle seguenti operazioni:

- ***BEGIN_TRANSACTION***: marca l'inizio dell'esecuzione della transazione;
- ***READ*** o ***WRITE***: specifica operazioni di lettura o scrittura sul database, eseguite come parte di una transazione;
- ***END_TRANSACTION***: specifica che le operazioni di READ e WRITE sono finite e marca il limite di fine esecuzione della transazione;
- ***COMMIT_TRANSACTION***: segnala la fine con successo della transazione, in modo che qualsiasi cambiamento può essere reso permanente, senza possibilità di annullarlo;
- ***ROLL-BACK*** (o ***ABORT***): segnala che la transazione è terminata senza successo e tutti i cambiamenti devono essere annullati.

- ***Transazioni acid + quali proprietà acid sono garantite dal programmatore e quali dal dbms:***

Le transazioni dovrebbero possedere alcune proprietà (dette ***ACID properties***, dalle loro iniziali):

- ***Atomicità***: rappresenta il fatto che vengono resi visibili tutti gli effetti di una transazione o la transazione non deve avere alcun effetto sul DB (responsabilità del DBMS);
- ***Consistency preserving***: quando il sistema rileva che una transazione sta violando uno dei vincoli di integrità, esso interviene per annullare la transazione o per correggere la violazione del vincolo, facendo passare il DB da uno stato consistente ad un altro (resp. programmatori);
- ***Isolamento***: richiede che l'esecuzione di una transazione sia indipendente dalla contemporanea esecuzione di altre transazioni, rendendo invisibili i vari aggiornamenti finché non è committed (responsabilità del DBMS);
- ***Durability***: la persistenza richiede che l'effetto di una transazione che ha eseguito il commit correttamente non venga più perso (resp. DBMS).

- ***Schedule:***

Uno ***schedule*** è un ordinamento delle operazioni delle transazioni processate in modo interleaved e le operazioni di una transazione devono apparire nello stesso ordine anche nello schedule. Due operazioni in uno schedule sono in *conflitto* se:

1. Appartengono a differenti transazioni;
2. Accedono allo stesso elemento X;
3. Almeno una delle due operazioni è una write_item(X).

È importante caratterizzare i tipi di schedule in base alla possibilità di effettuare il recovery. Infatti, si vuol garantire che per una transazione committed non è mai necessario il roll-back. I tipi di schedule sono:

- ***Schedule completo***, non contiene transazioni attive perché sono tutte committed o aborted;
- ***Schedule stretto***, le transazioni non possono né leggere né scrivere un elemento X finché l'ultima transazione che ha scritto X non è completata (commit o abort);
- ***Schedule seriale***, per ogni transazione nello schedule, tutte le operazioni delle transazioni sono eseguite senza interleaving.

Uno schedule è detto *cascadeless* (evitare il roll-back in cascata) se ogni transazione nello schedule legge elementi scritti solo da transazioni committed, in alternativa, si possono avere *roll-back in cascata* se una transazione non committed legge un dato scritto da una transazione fallita.

- ***Cos'è la serializzabilità:***

Uno schedule contenente transazioni è ***serializzabile*** se è "equivalente" a qualche schedule seriale delle stesse transazioni. Due schedule sono detti *result equivalent* (*equivalenti*) se producono lo stesso stato finale del database, ma non è una definizione accettabile, perché la produzione dello stesso stato può essere accidentale. Una definizione più appropriata è quella di *conflict equivalent*, cioè due schedule sono conflict equivalent se l'ordine di ogni coppia di operazioni in conflitto è lo stesso in entrambi gli schedule.

Uno schedule è ***conflict serializable*** se è conflict equivalent a qualche schedule seriale. È possibile determinare, per mezzo di un semplice algoritmo, la conflict serializzabilità di uno schedule. Molti metodi per il controllo della concorrenza non testano la serializzabilità, piuttosto utilizzano protocolli che garantiscono che uno schedule sarà serializzabile. Per quanto riguarda l'algoritmo, esso cerca solo le operazioni di read_item e write_item, per costruire un grafo di precedenza (o grafo di serializzazione).

7. TECNICHE PER IL CONTROLLO DELLA CONCORRENZA

- ***Tecniche per il controllo della concorrenza:***

Con l'esecuzione di transazioni concorrenti è necessario evitare che esse interferiscano fra di loro garantendo l'isolamento. Per favorire ciò si utilizzano delle tecniche di gestione delle transazioni, per garantire che il database consistente, tali tecniche garantiscono la serializzabilità degli schedule e sono:

- ***Tecniche di locking***: i data item sono bloccati per prevenire che transazioni multiple accedano allo stesso item concorrentemente.
- ***Timestamp***: un identificatore unico per ogni transazione, generato dal sistema. Un protocollo può usare l'ordinamento dei timestamp per assicurare la serializzabilità.

- ***Tecniche di locking + Lock shared/esclusivi:***

Le ***tecniche di locking*** si basano sul concetto di "blocco" (lock) di un item. Un lock è una variabile associata ad un data item nel database, e descrive lo stato di quell'elemento rispetto alle possibili operazioni applicabili ad esso. Possono essere utilizzati diversi tipi di lock:

- ***Lock binario***, associato a ciascun elemento del DB, può assumere due valori, se lock(X) = 1 le operazioni del database non possono accedere all'elemento, altrimenti lock(X) = 0 si può accedere all'elemento quando richiesto. Un lock binario rafforza la *mutua esclusione* di un data item. Le operazioni di *lock* e *unlock* devono essere implementate come unità indivisibili (*sezioni critiche*), nel senso che non è consentito alcun interleaving dall'avvio fino al termine dell'operazione di lock/unlock o all'intervento della transazione di una coda di attesa;
- ***Lock shared/esclusivi***, il lock binario è troppo restrittivo, poiché l'accesso ad un data item è consentito ad una sola transazione per volta, è possibile consentire l'accesso in sola lettura a più transazioni contemporaneamente. Se una transazione deve scrivere un data item, deve avere un accesso esclusivo su quel dato. Per questo motivo si utilizza un *multiple mode lock*, cioè un lock che può avere più stati. Ciascuna delle tre operazioni, Read_lock(X), Write_lock(X), Unlock_item(X), deve essere considerata indivisibile, cioè nessun interleaving deve essere consentito dall'inizio dell'operazione fino al completamento o all'inserimento della transazione in una coda di attesa per quell'elemento.

- ***Conversioni di lock (regole per aumentare un read lock a un write lock / regole per diminuire un write lock a un read lock):***

Una transazione può invocare un Read_Lock(X) e poi successivamente ***incrementare*** il lock, invocando un Write_Lock(X). Tale conversione è possibile solo se T è l'unica transazione che ha un Read_Lock su X, altrimenti deve aspettare. È possibile anche ***decrementare*** un lock, se una transazione T invoca una Write_Lock(X) e successivamente una Read_Lock(X).

- ***Protocolli per il controllo della concorrenza + 2pl (perché protocollo 2pl? per la serializzabilità) + Serializzabilità 2pl + 2pl deadlock free:***

Lock binari e multiple-mode non garantiscono la serializzabilità degli schedule e quindi occorre un protocollo per stabilire il posizionamento delle operazioni di lock/unlock in ogni transazione. Una transazione T segue il ***protocollo Two-Phase Locking*** (***2PL***) se tutte le operazioni di locking (Read_lock o Write_Lock) precedono la prima operazione di Unlock nella transazione. Una transazione del genere può essere divisa in due fasi:

1. *Expanding phase* (*espansione*), possono essere acquisiti nuovi lock su elementi ma nessuno può essere rilasciato;
2. *Shrinking phase* (*contrazione*), i lock esistenti possono essere rilasciati ma non possono essere acquisti nuovi lock.

È dimostrabile che se ogni transazione in uno schedule segue il ***protocollo 2PL***, allora lo schedule è ***serializzabile***. 2PL può però limitare la concorrenza in uno schedule, poiché alcuni elementi possono essere bloccati più del necessario, finché la transazione necessita di effettuare letture e scritture.

Il protocollo 2PL appena visto è detto ***2PL di base***. Una variazione del 2PL è nota come ***2PL conservativo*** (o ***statico***). Esso richiede che una transazione, prima di iniziare, blocchi tutti gli elementi a cui accede, pre-dichiarando i propri read_set (insieme di tutti i data item che saranno letti dalla transazione) e write_set (insieme di tutti i data item che saranno scritti dalla transazione). Se qualche data item dei due insiemi non può essere bloccato, la transazione resta in attesa finché tutti gli elementi necessari non divengono disponibili. Il 2PL conservativo è un ***protocollo deadlock-free***.
La variazione più diffusa del protocollo 2PL è il ***2PL stretto***, che garantisce schedule stretti, in cui le transazioni non possono né scrivere né leggere un elemento X finché l'ultima transazione che ha scritto X non termina (con commit o abort), esso non risulta essere deadlock-free.

- ***Deadlock e starvation in gestione transazioni (problematiche da risolvere in ambiente concorrente):***

Il protocollo di lock a due fasi garantisce la serializzabilità, ma non consente tutti i possibili schedule serializzabili, cioè alcuni schedule serializzabili vengono vietati da protocollo, perché possono causare:

- ***Deadlock***, quando due o più transazioni aspettano qualche item bloccato da altre transazioni T' in un insieme. Ogni transazione T' è in una coda di attesa e aspetta che un elemento sia rilasciato da un'altra transazione in attesa. Per prevenire il deadlock, occorre usare il *deadlock prevention protocol*, usato nel 2PL conservativo;
- ***Starvation***, se una transazione non può procedere per un tempo indefinito mentre altre transazioni nel sistema continuano normalmente. La causa è nello schema di waiting non sicuro che dà la precedenza ad alcune transazioni invece di altre. Un possibile schema di waiting sicuro (safe) usa una coda *first-come-first-serve* (*FCFS*) dove le transazioni bloccano gli elementi rispettando l'ordine con cui hanno richiesto il lock.

- ***Granularità + in che modo la scelta della granularità ci determina le possibilità della concorrenza?***

La dimensione del data item è detta ***granularità***, può essere un singolo campo di un record, così come un intero blocco di un disco, e può essere:

- *Fine*: riferita a data item di piccole dimensioni come un campo di un record.
- *Grossa*: riferita a data item di dimensioni maggiori come file o database intero.

La granularità influenza le prestazioni nel controllo della concorrenza e del recovery. Maggiore è il livello di granularità, minore è la concorrenza permessa. Se la dimensione di un data item è un blocco del disco, una transazione che necessita di leggere un record in quel blocco effettuerà un lock dell'intero blocco. Altre transazioni, interessate a record diversi ma ugualmente in quel blocco, resteranno inutilmente in attesa.

Per contro, a un data item di dimensioni inferiori corrisponde un numero maggiore di item nel database, quindi ci sarà una quantità superiore di lock ed il lock manager introdurrò un overhead nel sistema a causa delle molte operazioni che dovrà gestire.

8. TECNICHE DI RECOVERY

- ***Strategie di recovery:***

Se viene sottomessa una transazione, possono esserci solo due possibilità, o tutte le operazioni della transazione sono completate ed il loro effetto è registrato permanentemente nel database, oppure la transazione non ha nessun effetto né sul database né sulle altre transazioni (in caso di failure).

Effettuare un *recovery* (o recupero) da una transazione fallita significa *ripristinare* il database al più recente stato consistente appena prima del failure. Per fare ciò, il sistema deve tener traccia dei cambiamenti causati dall'esecuzione di transazioni sui dati. Tali informazioni sono memorizzate nel *system log*. Il log è strutturato come una lista di record, dove in ogni record memorizza un ID univoco della transazione e le operazioni effettuate.

Esistono varie ***strategie di recovery***, le tipiche:

1. Se il danno è notevole, a causa di un ***failure catastrofico*** (crash di un disco), si ripristina una precedente copia di back-up da una memoria di archivio e si ricostruisce lo stato più vicino a quello corrente riapplicando (REDO) tutte le transazioni committed salvate nel log fino al momento del failure.
2. Se, a seguito di un ***failure non catastrofico***, il database non è danneggiato fisicamente, ma è solo in uno stato inconsistente, si effettua l'UNDO delle operazioni che hanno causato l'inconsistenza.

- ***Tecniche di recovery***:

Concettualmente si possono distinguere due tecniche principali per il ***recovery*** *da failure non catastrofici*:

- ***Tecniche ad aggiornamento differito*** (***deferred update***) – Algoritmo NO-UNDO/REDO:
  I dati non sono fisicamente aggiornati fino all'esecuzione del commit di una transazione. Le modifiche effettuate da una transazione sono memorizzate in un suo spazio di lavoro locale (buffer). Durante il commit, gli aggiornamenti sono salvati persistentemente prima nel log e poi nel database. Se la transazione fallisce, non è mai necessario l'UNDO, dato che gli aggiornamenti vengono fatti prima nel buffer, può però essere necessario il REDO di alcune operazioni.
- ***Tecniche ad aggiornamento immediato*** (***immediate update***) – Algoritmo UNDO/REDO:
  Il database può essere aggiornato fisicamente prima che la transazione effettui il commit. Le modifiche sono registrate prima nel log (con un force-writing) e poi sul database, permettendo comunque il recovery. Se una transazione fallisce dopo aver effettuato dei cambiamenti, ma prima del commit, l'effetto delle sue operazioni nel database deve essere annullato, occorre quindi effettuare il rollback della transazione (UNDO), può però essere necessario il REDO di alcune operazioni.

Per quanto riguarda il ***recovery*** *da failure catastrofici*, la principale tecnica è quella del back-up di database. Periodicamente il database e il system log sono ricopiati su un device di memoria economico. Il system log è back-upped più di frequente, dato che la sua dimensione è notevolmente minore rispetto all'intero database. In caso di failure catastrofico, tutto il database viene ricaricato su dischi e, seguendo il system log, gli effetti delle transazioni committed vengono ripristinati.

9. BASE DI DATI DISTRIBUITI ED ARCHITETTURA CLIENT-SERVER

- ***Database distribuiti:***

Un ***database distribuito*** è un singolo DB logico che è sparso fisicamente attraverso computer in località differenti e connessi attraverso una rete di comunicazione dati. Invece, un *sistema per la gestione di basi di dati distribuite* (*DDBMS*) è un sistema software che gestisce un DB distribuito rendendo la distribuzione *trasparente* all'utente. La rete deve quindi consentire ad un utente in una data località di essere in grado di accedere (e aggiornare) i dati che si trovano in una località differente dalla sua. Un database distribuito è caratterizzato dall'eterogeneità dell'hardware e dei sistemi operativi di ogni nodo. Un database distribuito richiede DDBMS multipli, funzionanti ognuno su di un sito remoto (nodo). Gli ambienti di database distribuiti si differenziano in base al grado di cooperazione dei DDBMS, e alla presenza di un *sito master* che coordina le richieste.
Quando un sito fallisce, gli altri possono continuare ad operare, solo i dati del sito fallito sono inaccessibili. La replicazione dei dati aumenta ancora di più l'*affidabilità* e la *disponibilità*. In un sistema distribuito è più facile espandere il sistema in termini di aggiunta di nuovi dati, accrescere il numero di siti o aggiungere nuovi processori (miglioramento delle prestazioni). Infine, ogni sito ha un database di taglia più piccola e, query e transazioni sono processate più rapidamente.

- ***Livello di trasparenza:***

I vantaggi dei Database Distribuiti, partono dalla gestione dei dati distribuiti a diversi ***livelli di trasparenza***, ovvero sono nascosti i dettagli riguardo la posizione di ogni data item presente nel sistema.

- ***Trasparenza di distribuzione*** o ***di rete***, dove vi è libertà dell'utente dai dettagli della rete. Essa si suddivide in:
  - ***Trasparenza di locazione***: un comando utilizzato è indipendente dalla località dei dati e dalla località di emissione del comando.
  - ***Trasparenza di naming***: la specifica di un nome di un oggetto implica che una volta che un nome è stato fornito, gli oggetti possono essere referenziati usando quel nome, senza dover fornire dettagli addizionali.
- ***Trasparenza di replicazione***: dove delle copie dei dati possono essere mantenute in siti multipli per una migliore disponibilità, prestazioni ed affidabilità. L'utente non percepisce l'esistenza di tali copie.
- ***Trasparenza di frammentazione***, sono possibili due tipi di frammentazione:
  - ***Frammentazione orizzontale*** che distribuisce una relazione attraverso insieme di tuple.
  - ***Frammentazione verticale*** che distribuisce una relazione in sotto relazioni formate da un sottoinsieme delle colonne della relazione originale.

Una query globale dell'utente deve essere trasformata in una serie di *frammenti di query*. La trasparenza di frammentazione consente all'utente di non percepire l'esistenza di tali frammenti.

- ***Tecniche di frammentazione sui ddb (frammentazione verticale -> outer join) + Esempio frammentazione orizzontale e verticale:***

Il design dei DDB prevede l'applicazione di tecniche di ***Frammentazione*** dei dati, deve essere deciso quale sito deve memorizzare le diverse porzioni del database. Una relazione può essere memorizzata per intero in un sito, oppure essere divisa in unità più piccole distribuite. Uno schema di frammentazione di un database è la definizione di un insieme di frammenti che include tutte le tuple e gli attributi del database. Esso, inoltre, consente la ricostruzione dell'intero database applicando una sequenza di operazioni di OUTER JOIN e UNION.

- Un ***frammento orizzontale*** di una relazione è un sottoinsieme delle tuple (righe della tabella) della relazione. Le tuple vengono assegnate ad un determinato frammento orizzontale in base al valore di uno o più attributi. Ad esempio, considerato il database "Company", esso può essere distribuito su siti diversi definendo frammenti orizzontali in base al valore del numero del DIPARTIMENTO. Ogni frammento contiene le tuple degli impiegati che lavorano per un particolare dipartimento. La *segmentazione orizzontale derivata*, applica la partizione di una relazione primaria anche a relazioni secondarie, collegate alla prima con una chiave esterna. Ad esempio, dal partizionamento di DIPARTIMENTO deriva il partizionamento di IMPIEGATO e PROGETTO.
- La ***frammentazione verticale*** divide una relazione "verticalmente" per colonne. Un frammento verticale di una relazione mantiene solo determinati attributi di una relazione. Ad esempio, IMPIEGATO può essere suddiviso in due frammenti, uno contenente le informazioni personali {NOME, DATA_DI_NASCITA e INDIRIZZO}, e l'altro contenente {SSN, SALARIO e NUMERO_DIPARTIMENTO}. Questa frammentazione, però, non consente di ricostruire la tupla IMPIEGATO originale, infatti non ci sono attributi in comune fra i due frammenti. La soluzione è includere la chiave primaria o una candidata in ogni frammento verticale. Per ricostruire la relazione originaria dai frammenti verticali è necessario fare l'OUTER JOIN dei frammenti.
- La ***frammentazione mista***, è possibile combinare la frammentazione orizzontale e verticale. Ad esempio, si combinano le due frammentazioni viste per la relazione IMPIEGATO. La relazione originaria può essere ricostruita applicando OUTER JOIN e UNION nell'ordine appropriato.

- ***Concorrenza nei ddb:***

Nell'ottica della gestione delle transazioni il controllo della ***concorrenza*** ed il commit sono gestiti dal sito primario. La tecnica del *lock a due fasi* (*2PL*) è usata per bloccare e rilasciare i data item. Se tutte le transazioni in tutti i siti seguono la politica delle due fasi allora viene garantita la serializzabilità.

- Vantaggi: i data item sono bloccati (locked) solamente in un sito ma possono essere usati da qualsiasi altro sito.
- Svantaggi: tutta la gestione delle transazioni passa per il sito primario che potrebbe essere sovraccaricato. Nel caso in cui il sito primario fallisce, l'intero sistema è inaccessibile.

Per assistere il recovery, un sito di backup viene designato come copia di backup del sito primario. Nel caso di fallimento del sito primario, il sito di backup funziona da sito principale. Esiste anche un controllo della ***concorrenza basato sul voting*** non esiste la copia primaria del coordinatore:

1. Spedire una richiesta di lock ai siti che hanno i data item.
2. Se la maggioranza dei siti concedono il lock allora la transazione richiedente ottiene il data item.
3. Le informazioni di lock (concesse o negate) sono spedite a tutti questi siti.
4. Per evitare un tempo di attesa inaccettabile, viene definito un periodo di time-out. Se la transazione richiedente non riceve alcuna informazione di voto allora la transazione viene abortita.

- ***Controllo della concorrenza e recovery in ambito distribuito:***

I database distribuiti incontrano un numero di ***controlli di concorrenza e problemi di recovery*** che non sono presenti nei database centralizzati, come:

- Trattamento di *copie multiple di dati*: Il controllo della consistenza deve mantenere una consistenza globale. Allo stesso modo il meccanismo di recovery deve recuperare tutte le copie e conservare la consistenza dopo il recovery.
- Fallimenti di *singoli siti*: La disponibilità del database non deve essere influenzata dai guasti di uno o due siti e lo schema di recovery li deve recuperare prima che siano resi disponibili.
- Guasto dei *collegamenti di comunicazione*: Tale guasto può portare ad una partizione della rete che può influenzare la disponibilità del database anche se tutti i siti sono in esecuzione.
- *Commit distribuito*: Una transazione può essere frammentata ed essere eseguita su un numero di siti. Questo richiede un approccio basato sul commit a due fasi per il commit della transazione.
- *Deadlock distribuito*: Poiché le transazioni sono processate su siti multipli, due o più siti possono essere coinvolti in un deadlock. Di conseguenza devono essere considerate le tecniche per il trattamento dei deadlock.
- Controllo della concorrenza distribuito basato su una *copia designata* dei dati: Si designa una particolare copia di ogni dato (copia designata). Tutte le richieste di lock ed unlock vengono inviate solo al sito che la contiene.
- Tecnica del *sito primario*: Un singolo sito è designato come sito primario, il quale fa da coordinatore per la gestione delle transazioni.

12. DATABASE NoSQL

- ***db nosql:***

I ***DB NoSQL*** sono appositamente realizzati per far fronte ad un gran numero di utenza e alla rapida crescita dell'ammontare di dati. La chiave a monte dei NoSQL è l'accesso globale in real-time. NoSQL non è uno specifico linguaggio, ma è il termine che raggruppa un insieme di tecnologie per la persistenza dei dati che funzionano in modo sostanzialmente diverso dai DB relazionali, quindi non rispettano una o più caratteristiche dei RDBMS. Infatti, alcuni DB NoSQL garantiscono solo alcune proprietà ACID, esse vengono rilassate per fornire performance migliori. Le principali caratteristiche dei NoSQL è che sono *schemaless* e consentono di memorizzare attributi on the fly, anche senza averli definiti a priori, questo per consentire la memorizzazione di dati fortemente dinamici, la scalabilità che consente di memorizzare e gestire una grande quantità di informazioni e velocità di risposta alle query. I NoSQL si basano sul *modello BASE*:

- *Basically Available*: garantire la disponibilità dei dati anche in presenza di fallimenti multipli. L'obiettivo è raggiunto attraverso un approccio fortemente distribuito;
- *Soft State*: abbandonano il requisito della consistenza dei modelli ACID quasi completamente. La consistenza è un problema dello sviluppatore e non deve essere gestita dal database.
- *Eventually Consistent*: l'unico requisito riguardante la consistenza è garantire che, ad un certo momento, nel futuro, i dati possano convergere ad uno stato consistente.

- ***cap theorem:***

Il ***Brewer's Cap Theorem*** dice che un sistema distribuito è in grado di supportare solamente due tra le seguenti caratteristiche:

- ***Consistency***: tutti i nodi vedono lo stesso dato nello stesso tempo;
- ***Availability***: ogni operazione deve sempre ricevere una risposta;
- ***Partition tolerance***: capacità di un sistema di essere tollerante ad una aggiunta o rimozione di un nodo nel sistema distribuito o alla disponibilità di un componente singolo.

13. DATA WAREHOUSE

- ***Operatori olap (Nota: anche il pivoting):***

Un Data Warehouse è una base di dati per il supporto alle decisioni, che è mantenuta separata dalle basi di dati operative, nella quale vengono raccolte tutte le informazioni di ausilio all'analisi. ***OLAP*** è la principale modalità di fruizione delle informazioni contenute in un DW. Consente agli utenti di esplorare interattivamente i dati sulla base del *modello multidimensionale*, quest'ultimo viene utilizzato per la rappresentazione e l'interrogazione dei dati nei DW. Gli OLAP sono operazioni che si applicano a cubi multidimensionali e restituiscono nuovi cubi, non necessariamente con lo stesso numero di dimensioni. Esistono diversi operatori tra cui:

- ***Slicing***, si fissa un valore per almeno uno degli attributi dimensionali e si escludono dall'analisi tutti quegli eventi che non presentano tale valore;
- ***Dacing***, si stabilisce per almeno una delle dimensioni di analisi un sottoinsieme di valori possibili per tale attributo e di escludere quei fatti che non sono associati a nessuno di tali valori;
- ***Roll-Up***, consiste in un'aggregazione dei dati di un cubo seguita dall'applicazione di una funzione aggregativa (in genere la somma);
- ***Drill-Down***, operazione inversa del roll-up, consiste cioè di aggiungere dettaglio a un cubo disaggregandolo lungo una o più dimensioni;
- ***Drill-Across***, consiste nello stabilire un confronto tra due o più cubi correlati in modo da ottenere una visualizzazione comparata di due diverse misure e per il calcolo di misure derivate dai dati presenti sui cubi.
- ***Drill-Through***, consiste nel passaggio dai dati aggregati multi dimensionalmente del DW ai dati operazionali presenti nelle sorgenti;
- ***Pivoting***, consiste nel ruotare gli assi di visualizzazione del cubo dei fatti mantenendo invariato il livello di aggregazione ed il numero di dimensioni: incrementa leggibilità delle stesse informazioni.

- ***Eventi primari e secondari:***

Un ***evento primario*** è una particolare occorrenza di un fatto, individuata da una ennupla caratterizzata da un valore per ciascuna dimensione. Ad esempio, un evento primario potrebbe essere il “05/04/01” nel negozio “DiTutto” è stata venduta una quantità 3 con incassi 12 del prodotto “Brillo”.

Un ***evento secondario*** è, dato un insieme di attributi dimensionali (pattern), ciascuna ennupla di questi valori che aggrega tutti gli eventi primari corrispondenti.

Le *gerarchie* definiscono il modo in cui gli eventi primari possono essere aggregati e selezionati significativamente per il processo decisionale.