BASI DATI:

Sistemi Informativi e Sistemi Informatici

Nello svolgimento di ogni attività, la **disponibilità di informazioni** e la **capacità di gestirle efficacemente** sono essenziali, sia a livello individuale sia nelle organizzazioni di ogni dimensione. Ogni organizzazione, infatti, dispone di un **sistema informativo** che organizza e gestisce le informazioni necessarie per perseguire i propri scopi. È importante notare che l'esistenza di un sistema informativo è solo **in parte legata alla sua automatizzazione**: *i sistemi informativi esistono da molto prima dell'era dei calcolatori elettronici* (ad esempio, archivi bancari o anagrafici).

Quando parliamo della parte automatizzata di un sistema informativo, usiamo il termine **sistema informatico**. Negli ultimi decenni, la diffusione capillare dell'informatica ha fatto sì che molti sistemi informativi siano anche sistemi informatici, ma non bisogna confondere i due concetti. Nelle attività umane più semplici, le informazioni possono essere gestite anche senza rappresentazioni complesse, utilizzando **lingua scritta, numeri o disegni**. In altri casi, le informazioni non vengono nemmeno formalizzate ma *restano memorizzate a livello mentale*. Con la crescita della complessità delle attività, è diventato necessario **organizzare e codificare** meglio le informazioni.

Nei **sistemi informatici**, la **rappresentazione e codifica** delle informazioni diventa cruciale. Le informazioni vengono tradotte in **dati**, che devono essere interpretati per fornire significato. **I dati da soli non hanno alcun significato**; *è solo attraverso la loro interpretazione e correlazione che essi diventano informazioni*. Le informazioni, a loro volta, *arricchiscono la nostra conoscenza del mondo*. A livello pratico, possiamo considerare i **dati** come **simboli** (numeri, stringhe, ecc.) che richiedono **elaborazione** per acquisire significato, diventando così **informazioni** utili.

Un esempio semplice chiarisce questa distinzione: la stringa "Ferrari" e il numero "8", scritti su un foglio di carta, sono **dati**. Da soli, non significano nulla. Tuttavia, *nel contesto di un ristorante durante una notte di Capodanno*, questi dati possono rappresentare un'ordinazione di una bottiglia di spumante della marca Ferrari, da addebitare alla camera numero 8. Questo esempio dimostra come, *una volta aggiunte informazioni contestuali*, i dati si trasformino in **informazioni**.

Introducendo il concetto di **base di dati**, possiamo definire questa come una **collezione di dati** utilizzata per rappresentare le informazioni necessarie per un sistema informativo. Nei sistemi informativi complessi, una **base di dati** diventa uno strumento fondamentale per gestire grandi quantità di dati in modo efficiente. Inoltre, *i dati tendono a essere molto più stabili nel tempo rispetto alle procedure che operano su di essi*. Per esempio, i **dati bancari** sono rimasti invariati per decenni, mentre le procedure che li gestiscono cambiano più frequentemente. Quando una procedura viene sostituita, la nuova "eredita" i **dati** della vecchia, spesso con trasformazioni.

Questa stabilità dei **dati** porta alla conclusione che essi rappresentano una **risorsa fondamentale** per l'organizzazione: *un vero e proprio patrimonio da sfruttare e proteggere*.

Basi di Dati: Un Pilastro dell'Informazione

Fin dagli albori dell'informatica, l'attenzione verso la **gestione dei dati** ha sempre rappresentato un aspetto centrale. Tuttavia, solo a partire dalla fine degli anni Sessanta sono stati sviluppati **sistemi software specifici per la gestione dei dati**, e, ancora oggi, alcune applicazioni non ne fanno uso. In assenza di un **software dedicato**, la gestione dei dati è stata tradizionalmente affidata a **linguaggi di programmazione** come C e Fortran o, in tempi più recenti, a **linguaggi a oggetti** come C++ e Java. Esistono ancora applicazioni scritte in COBOL, un linguaggio degli anni Sessanta ormai superato.

L'approccio **convenzionale** alla gestione dei dati si basa su **archivi o file** per memorizzare i dati in modo persistente su memoria di massa. Sebbene un file permetta di memorizzare e cercare dati, esso offre solo **semplici meccanismi di accesso** e condivisione. In questo modello, ogni programma gestisce i propri **file "privati"**, il che comporta **duplicazioni** dei dati e **possibili incoerenze** quando dati replicati non vengono aggiornati in modo sincrono.

Un esempio concreto di questa problematica si può osservare nel contesto universitario, dove diverse entità gestiscono informazioni sui docenti. L'**ufficio del personale** mantiene i dati relativi alla carriera accademica, le **presidenze delle facoltà** gestiscono gli incarichi di insegnamento, e l'**ufficio stipendi** utilizza tali informazioni per calcolare le retribuzioni. Se ognuno di questi uffici gestisse le proprie informazioni separatamente, si creerebbero **incoerenze** tra le varie copie dei dati.

Le **basi di dati** sono state sviluppate per **risolvere questi problemi**, permettendo la gestione integrata delle informazioni e riducendo **ridondanza** e **incoerenze**. Un **sistema di gestione di basi di dati** (DBMS) è un **software**

progettato per gestire **collezioni di dati** di grandi dimensioni, condivise tra più utenti, e garantire che siano **persistenti**, **affidabili** e **private**.

Le principali caratteristiche di un **DBMS** sono:

- **Grandi dimensioni**: Le basi di dati possono raggiungere dimensioni enormi, misurate in centinaia o migliaia di terabyte. I DBMS devono essere in grado di gestire queste collezioni di dati, che spesso superano la capacità della memoria centrale disponibile.
- **Condivisione dei dati**: I dati devono essere **accessibili** a più utenti o applicazioni in modo sicuro, riducendo la ridondanza e prevenendo **inconsistenze**. I DBMS includono un **controllo di concorrenza** per garantire che le operazioni simultanee non causino conflitti.
- **Persistenza**: I dati rimangono **disponibili** anche al termine dell'esecuzione di un programma. A differenza dei dati temporanei gestiti in memoria centrale, i dati in un DBMS sono **permanenti**.
- **Affidabilità**: I DBMS offrono meccanismi di **salvataggio e ripristino** per proteggere i dati da errori hardware o software, garantendo che i dati cruciali, come quelli finanziari, siano **preservati nel tempo**.
- **Privatezza**: I DBMS regolano l'accesso ai dati attraverso **meccanismi di autorizzazione**, permettendo solo ad utenti autorizzati di eseguire determinate operazioni sui dati.
- **Efficienza ed efficacia**: Un DBMS deve essere **efficiente** nell'utilizzo delle risorse di sistema e **efficace** nel rendere produttive le attività degli utenti.

Sebbene sia possibile gestire dati **persistenti** e **di grandi dimensioni** utilizzando file semplici, i **DBMS offrono funzionalità superiori**, come l'accesso condiviso e la gestione centralizzata. Ad esempio, in un'università, un sistema ideale prevedrebbe una **singola base di dati** condivisa tra tutti gli uffici per gestire le informazioni sui docenti, i corsi, le facoltà e gli aspetti amministrativi. Tuttavia, a volte è necessario mantenere basi di dati separate, come nel caso delle informazioni riservate sui docenti rispetto a quelle pubbliche sui corsi, che possono essere accessibili tramite il sito web dell'ateneo.

Modelli dei dati

Un **modello dei dati** è un insieme di concetti utilizzati per **organizzare** i dati di interesse e descriverne la **struttura** in modo che possa essere compresa e gestita da un elaboratore. Ogni modello dei dati fornisce dei **meccanismi di strutturazione**, che sono analoghi ai costruttori di tipo dei linguaggi di programmazione. Questi meccanismi permettono di definire **nuovi tipi** basati su tipi **predefiniti** (elementari) e costruttori di tipo.

Ad esempio, nel linguaggio di programmazione **C**, è possibile definire nuovi tipi attraverso costruttori come **struct**, **union**, **enum**, e **pointer** (*). Il **modello relazionale**, introdotto formalmente agli inizi degli anni Settanta e diffusosi durante il decennio successivo, rappresenta il modello dei dati più utilizzato e viene considerato il modello di **riferimento** nel campo della gestione dei dati.

Il **modello relazionale dei dati** consente di definire tipi attraverso il **costruttore relazione**, che organizza i dati in **insiemi di record a struttura fissa**. Una relazione è comunemente rappresentata come una **tabella**: le **righe** rappresentano specifici record, mentre le **colonne** corrispondono ai campi di ciascun record. È importante notare che l'**ordine** delle righe e delle colonne in una tabella non ha rilevanza nel contesto del modello relazionale.

Schemi e istanze

Nelle **basi di dati**, si distingue tra una parte **sostanzialmente invariante nel tempo**, detta **schema della base di dati**, e una parte **variabile nel tempo**, chiamata **istanza o stato della base di dati**. Lo **schema** rappresenta le **caratteristiche strutturali dei dati**, mentre l'**istanza** si riferisce ai **valori effettivi** che tali dati assumono in un dato momento.

Ad esempio, nella relazione **DOCENZA**, lo schema è costituito dalla struttura fissa che include due **attributi**: **Corso** e **NomeDocente**. Lo schema della relazione si può rappresentare così:
DOCENZA(Corso, NomeDocente).
Le **righe** della tabella, invece, sono soggette a cambiamenti nel tempo, riflettendo l'**offerta corrente di corsi** e i **docenti associati**. Ad esempio, in un dato momento, l'istanza potrebbe essere:

Basi di dati – Rossi

Reti – Neri

Linguaggi – Verdi

In termini tecnici, lo **schema** rappresenta la componente **intensionale** della base di dati, mentre l'**istanza** rappresenta la componente **estensionale**.

Architettura a tre livelli delle basi di dati

La nozione di modello e di schema può essere ulteriormente articolata attraverso una proposta di **architettura a tre livelli** per i DBMS. I tre livelli sono:

- **Schema esterno**: descrive una porzione della base di dati, specifica per l'utente o un gruppo di utenti. Lo schema esterno può avere un'organizzazione diversa rispetto a quella dello schema logico. In effetti, è possibile avere diversi schemi esterni per uno stesso schema logico.
- **Schema logico**: rappresenta una descrizione **complessiva** della base di dati secondo il **modello logico** scelto (ad esempio, relazionale o a oggetti).
- **Schema interno**: descrive la rappresentazione **fisica** dei dati, come vengono **memorizzati** su disco, per esempio tramite file sequenziali o hash.

In pratica, nei sistemi moderni, il **livello esterno** può essere gestito tramite l'uso di **viste** (**views**), ovvero relazioni derivate da altre tabelle. Queste permettono di adattare l'**accesso ai dati** in base alle esigenze degli utenti. Ad esempio, uno studente di **Ingegneria Elettronica** potrebbe vedere solo i corsi del suo manifesto, tramite una **vista** sulla base di dati più ampia, come mostrato nella relazione **ELETTRONICA**.

Inoltre, i meccanismi di **autorizzazione** consentono di regolare l'**accesso degli utenti** alle informazioni della base di dati, garantendo così sia la sicurezza che la **privatezza** dei dati sensibili.

Modelli logici nei sistemi di basi dati

Il **modello relazionale** si fonda su due concetti principali: **relazione** e **tabella**. Questi concetti, pur avendo origini diverse, sono strettamente connessi. La **relazione** proviene dalla **teoria degli insiemi** nella matematica, mentre la **tabella** è un concetto intuitivo e di uso comune. Questa combinazione ha contribuito al successo del modello relazionale, in quanto offre sia una base formale rigorosa sia una rappresentazione comprensibile e utilizzabile facilmente dagli utenti finali.

Indipendenza dei dati

Il modello relazionale è stato progettato per garantire l'**indipendenza dei dati**, separando il **livello fisico** da quello **logico**. Gli utenti e i programmatori operano esclusivamente al livello logico, senza la necessità di conoscere come i dati vengono effettivamente memorizzati fisicamente. Questa separazione rende il modello versatile e adatto per molte applicazioni, poiché eventuali modifiche alla struttura fisica dei dati non richiedono cambiamenti nelle applicazioni che li utilizzano.

In confronto, i modelli di basi di dati **precedenti** (come quello **reticolare** e quello **gerarchico**) incorporavano riferimenti espliciti alle strutture fisiche attraverso puntatori e l'ordine fisico dei dati. Questo vincolo rendeva più difficile la gestione e l'evoluzione delle basi di dati.

Tre accezioni del termine "relazione"

Il termine **relazione**, nell'ambito delle basi di dati, ha tre diverse interpretazioni:

- **Relazione matematica**: deriva dalla teoria degli insiemi e si riferisce a una collezione di n-tuples o coppie ordinate di elementi.
- **Relazione nel modello relazionale**: utilizza la nozione matematica come base, ma presenta alcune differenze, come vedremo più avanti. Una tabella rappresenta una relazione in questo contesto.
- **Relazione nel modello Entità-Relazione (Entity-Relationship)**: questo concetto, tradotto dall'inglese "relationship", è usato per rappresentare i **legami tra entità** del mondo reale nel modello concettuale.

Relazioni e Tabelle

Il **prodotto cartesiano** è un concetto fondamentale nella matematica, in particolare nella teoria degli insiemi, ed è alla base del **modello relazionale** nelle basi di dati. Dati due insiemi D_1 e D_2, il loro prodotto cartesiano, indicato come $D_1 \times D_2$, è l'insieme di tutte le **coppie ordinate** (u_1, u_2), dove u_1 appartiene a D_1 e u_2 appartiene a D_2.

Esempio di prodotto cartesiano:

Se consideriamo gli insiemi:

$$A = \{1,2,4\}$$

$$B = \{a, b\}$$

Il prodotto cartesiano $A \times B$ è l'insieme di tutte le possibili coppie (a, b) in cui il primo elemento proviene da A e il secondo da B, cioè:

$$A \times B = \{(1, a), (1, b), (2, a), (2, b), (4, a), (4, b)\}$$

Questo esempio genera sei coppie ordinate, poiché A contiene 3 elementi e B ne contiene 2, quindi $3 \times 2 = 6$.

Relazione matematica:

Una **relazione matematica** sugli insiemi D_1 e D_2 è semplicemente un **sottoinsieme del prodotto cartesiano** $D_1 \times D_2$. Ad esempio, una relazione su A e B potrebbe essere:

$$\{(1, a), (1, b), (4, b)\}$$

Questa relazione contiene solo alcune coppie del prodotto cartesiano, quindi è un sottoinsieme.

Rappresentazione tabellare:

Le relazioni matematiche possono essere rappresentate in forma di **tabelle**, dove:

- Le colonne della tabella corrispondono ai domini D_1, D_2, ecc.
- Le righe corrispondono alle n-uple o coppie ordinate della relazione.

Nell'esempio precedente, il prodotto cartesiano $A \times B$ e la relazione specifica possono essere rappresentati come tabelle con righe e colonne.

Generalizzazione a n insiemi:

Il concetto di prodotto cartesiano può essere generalizzato a più di due insiemi. Dati n insiemi $D_1, D_2, \ldots, D_n$, il loro prodotto cartesiano è indicato come $D_1 \times D_2 \times \ldots \times D_n$ ed è costituito dalle **n-uple** $(v_1, v_2, \ldots, v_n)$, dove ciascun v_i appartiene a D_i. Una **relazione matematica** su questi insiemi è un sottoinsieme del prodotto cartesiano.

Il numero n delle componenti di una n-upla (e quindi del prodotto cartesiano) è chiamato **grado** della relazione. Il numero di n-uple in una relazione è la sua **cardinalità**.

Applicazione pratica nelle basi di dati:

Le relazioni matematiche trovano una rappresentazione pratica nel modello relazionale delle basi di dati. Ad esempio, una tabella che registra i risultati di partite di calcio può essere rappresentata come una relazione. Se i domini sono:

`Squadra1:Stringa`

`Squadra2:Stringa`

`GolSquadra1:Intero`

`GolSquadra2:Intero`

La relazione risulterebbe essere un sottoinsieme del prodotto cartesiano di questi domini:

$$Stringa \times Stringa \times Intero \times Intero$$

Ogni riga della tabella rappresenta una partita e i rispettivi risultati.

In sintesi, il concetto di **prodotto cartesiano** e di **relazione** formano la base del **modello relazionale** utilizzato nelle basi di dati, dove ogni tabella rappresenta una relazione matematica che può essere rappresentata e manipolata in modo chiaro e intuitivo.

Relazioni con attributi

Le osservazioni sulle relazioni e le loro rappresentazioni tabellari nel contesto delle basi di dati evidenziano diversi aspetti importanti. Ogni **n-upla** in una relazione stabilisce un legame tra i suoi elementi. Ad esempio, una n-upla $(u_1, u_2, \dots, u_n)$ collega i valori dei domini $D_1, D_2, \dots, D_n$ in un certo ordine. Nella relazione che rappresenta i risultati di partite di calcio, una riga come $("Juventus", "Lazio", 3,1)$ stabilisce un collegamento tra i nomi delle squadre e i rispettivi punteggi.

È importante notare che una relazione è un insieme di n-uple, e come tale presenta alcune proprietà. Innanzitutto, non esiste alcun ordinamento tra le n-uple; l'ordine degli elementi in un insieme non è rilevante. Anche se in una rappresentazione tabellare le righe sono visualizzate in un certo ordine, questo è "accidentale". Cambiando l'ordine delle righe, la relazione rappresentata rimane invariata. Inoltre, le n-uple di una relazione devono essere distinte; non possono esserci n-uple duplicate, poiché gli elementi di un insieme devono essere unici. Pertanto, ogni riga di una tabella deve essere unica.

Allo stesso tempo, **l'ordinamento interno** delle n-uple è cruciale. Sebbene l'ordine delle righe in una relazione sia irrilevante, all'interno di ciascuna n-upla l'ordine dei valori è importante. Ogni valore è associato a un dominio specifico e la posizione del valore è fondamentale per interpretarlo correttamente. Se, ad esempio, si scambiasse la posizione dei domini che rappresentano i gol segnati dalle squadre, il significato della relazione cambierebbe drasticamente.

Il fatto che i valori siano interpretati in base alla loro posizione rende il concetto di relazione matematica meno flessibile quando si tratta di gestire dati complessi. In molti contesti informatici, si preferisce l'uso di notazioni non posizionali, come nei record, dove i campi sono identificati da nomi (attributi), piuttosto che da posizioni. Questa preferenza deriva dal fatto che i nomi sono più esplicativi e permettono una gestione più semplice e comprensibile dei dati.

Quando usiamo le relazioni per organizzare i dati nelle basi di dati, possiamo pensare a ciascuna relazione come a un insieme di record omogenei, definiti sugli stessi campi (o attributi). Ciò rende la struttura della relazione più vicina a quella di un record piuttosto che a una semplice n-upla posizionale. Una relazione che registra i risultati delle partite potrebbe avere attributi come **SquadraDiCasa**, **SquadraOspitata**, **RetiCasa**, e **RetiOspitata**.

Associando a ciascun dominio un nome simbolico (attributo), si può migliorare la chiarezza della relazione. Invece di riferirsi ai domini in base alla loro posizione, possiamo riferirci agli attributi. Per esempio, nella tabella delle partite, gli attributi possono essere usati come intestazioni delle colonne, rendendo la relazione più leggibile e interpretabile.

Una volta introdotti gli attributi, l'ordine delle colonne (e degli attributi stessi) diventa irrilevante. Non è più necessario parlare di primo dominio, secondo dominio e così via; è sufficiente fare riferimento agli attributi. Questo consente una maggiore flessibilità nella rappresentazione dei dati. Inoltre, è possibile utilizzare una notazione utile per riferirsi agli attributi, come $t[A]$, dove t è una tupla e A è un attributo. Questa notazione ci permette di accedere al valore della tupla per un dato attributo.

Infine, possiamo formalizzare il concetto di tupla: una tupla su un insieme di attributi X è una funzione t che associa a ciascun attributo $A \in X$ un valore appartenente al dominio associato a A. Con questa nuova definizione, una relazione non è più vista come un insieme di n-uple posizionali, ma come un insieme di tuple che associano i valori ai nomi degli attributi. Questa definizione elimina la necessità di interpretare i dati in base alla loro posizione, rendendo la gestione dei dati più flessibile e intuitiva.

Relazione e basi di dati

Come già osservato, una relazione può essere utilizzata per organizzare dati rilevanti nell'ambito di un'applicazione di interesse. Tuttavia, di solito non è sufficiente una singola relazione; una base di dati è generalmente costituita da più relazioni, le cui tuple contengono valori comuni, necessari per stabilire corrispondenze.

Per esemplificare, consideriamo una base di dati in cui:

- **La prima relazione** contiene informazioni relative a un insieme di studenti, con numero di matricola, cognome, nome e data di nascita.
- **La seconda relazione** raccoglie informazioni relative agli esami, specificando il numero di matricola dello studente, il codice del corso e il voto. Questa relazione fa riferimento ai dati contenuti nelle altre due: agli studenti, attraverso i numeri di matricola, e ai corsi, tramite i relativi codici.
- **La terza relazione** include informazioni su alcuni corsi, comprendenti codice, titolo e docente.

La base di dati presenta una delle caratteristiche fondamentali del modello relazionale, spesso descritta come "basata su valori": i riferimenti tra dati in relazioni diverse sono rappresentati mediante valori dei domini che compaiono nelle

tuple. Gli altri modelli logici, come quello reticolare e gerarchico (sviluppati prima del modello relazionale ma ancora in uso), stabiliscono le corrispondenze in modo esplicito attraverso puntatori e sono perciò detti modelli "basati su record e puntatori". Sebbene non approfondiremo questi modelli, è utile evidenziare la loro caratteristica principale: i dati sono collegati attraverso puntatori.

Rispetto a un modello basato su record e puntatori, il modello relazionale presenta diversi vantaggi:

Richiede di rappresentare solo ciò che è rilevante dal punto di vista dell'applicazione (e quindi dell'utente); i puntatori sono un elemento aggiuntivo, legato a aspetti realizzativi. Nei modelli con puntatori, il programmatore si riferisce a dati che non sono significativi per l'applicazione.

La rappresentazione logica dei dati, costituita esclusivamente dai valori, non fa alcun riferimento alla rappresentazione fisica, che può cambiare nel tempo. Il modello relazionale consente quindi di ottenere l'indipendenza fisica dei dati; poiché tutte le informazioni sono contenute nei valori, è relativamente semplice trasferire i dati da un contesto a un altro (ad esempio, trasferire una base di dati da un computer a un altro). In presenza di puntatori, l'operazione risulta più complessa, poiché i puntatori hanno un significato locale al singolo sistema, il che non sempre è immediato da esportare.

È interessante notare che anche in una base di dati relazionale, a livello fisico, i dati possono essere rappresentati attraverso l'uso di puntatori. Tuttavia, nei modelli relazionali, i puntatori non sono visibili a livello logico. Inoltre, nei sistemi di basi di dati a oggetti, che rappresentano una delle direzioni evolutive delle basi di dati, vengono introdotti identificatori di oggetto, i quali, sebbene a un livello di astrazione più alto, presentano alcune caratteristiche dei puntatori.

Possiamo ora riassumere le definizioni relative al modello relazionale, distinguendo il livello degli schemi da quello delle istanze:

Schema di relazione: costituito da un simbolo R (nome della relazione) e da un insieme di (nomi di) attributi $X = \{A_1, A_2, \dots, A_n\}$, di solito indicato come $R(X)$. A ciascun attributo è associato un dominio, come visto in precedenza.

Schema di base di dati: un insieme di schemi di relazione con nomi diversi:

$$R = \{R_1(X_1), R_2(X_2), \dots, R_n(X_n)\}$$

I nomi di relazione servono principalmente a distinguere le varie relazioni all'interno della base di dati.

Istanza di relazione: un insieme r di tuple su X.

Istanza di base di dati: un insieme di relazioni $r = \{r_1, r_2, \dots, r_n\}$ su uno schema

$$R = \{R_1(X_1), R_2(X_2), \dots, R_n(X_n)\}$$

dove ogni r_i è una relazione sullo schema $R_i(X_i)$.

Per chiarire, lo schema della base di dati può essere definito come segue (con opportune definizioni per i domini):

$$R = \{STUDENTI(Matricola, Cognome, Nome), ESAMI(Voto, Corso), CORSI(Codice, Titolo, Docente)\}$$

Per approfondire ulteriormente i concetti fondamentali del modello relazionale, esaminiamo un paio di esempi. In primo luogo, notiamo che secondo la definizione sono ammissibili relazioni su un solo attributo. Ciò può avere senso, in particolare, in basi di dati composte da più relazioni, in cui la relazione su singolo attributo contiene valori che appaiono come valori di un attributo di un'altra relazione. Ad esempio, in una base di dati che include la relazione STUDENTI, si può utilizzare un'altra relazione sul solo attributo Matricola per indicare gli studenti lavoratori (attraverso i rispettivi numeri di matricola, che devono apparire nella relazione STUDENTI).

Informazione incompleta e valori nulli

La struttura del modello relazionale, come discusso, è molto semplice e potente, ma presenta anche una certa rigidità. Le informazioni devono essere rappresentate tramite tuple di dati omogenee, e ogni relazione può contenere solo tuple conformi al suo schema. Questo può risultare problematico quando i dati disponibili non corrispondono esattamente al formato previsto.

Ad esempio, in una relazione con lo schema:

PERSONE(Cognome, Nome, Indirizzo, Telefono)

il valore dell'attributo Telefono potrebbe non essere disponibile per tutte le tuple. Utilizzare un valore del dominio per rappresentare l'assenza di informazioni, come ad esempio lo zero per i numeri di telefono, può generare confusione e non è una pratica corretta. Ciò richiede che esista un valore mai utilizzato per indicare l'assenza, e non sempre è possibile trovare un valore del dominio adeguato. Ad esempio, in un attributo per le date di nascita, non ci sono valori non utilizzati in un tipo Data correttamente definito.

Inoltre, l'uso di valori del dominio può mascherare la distinzione tra valori "veri" e valori fittizi, rendendo necessario che i programmi accedano alla base di dati con attenzione, distinguendo correttamente i valori.

Per rappresentare la non disponibilità di valori in modo più chiaro, si introduce il concetto di valore nullo. Un valore nullo denota l'assenza di informazione ed è un valore speciale distinto dai valori del dominio. Nelle rappresentazioni tabellari, il valore nullo viene indicato con il simbolo NULL.

Consideriamo i seguenti esempi di valori nulli:

Città A: ha un valore nullo per l'indirizzo della prefettura, indicando che il valore è sconosciuto.

Città B: non ha una prefettura, quindi l'attributo relativo è inesistente; il valore nullo rappresenta l'inesistenza del valore.

Città C: è una nuova provincia e non si conosce né se esista né quale sia l'indirizzo della prefettura; il valore nullo indica l'incertezza riguardo all'esistenza dell'informazione.

Nei sistemi di basi di dati relazionali, i valori nulli sono gestiti in modo semplice, senza ipotesi particolari, collocandoli nella categoria di valori "senza informazione".

Riflettendo ulteriormente sui valori nulli, consideriamo una base di dati con lo stesso schema delle PERSONE. Un valore nullo per la data di nascita potrebbe essere accettabile, mentre un valore nullo per il numero di matricola o per il codice di un corso comporterebbe problematiche maggiori, poiché questi valori sono essenziali per stabilire correlazioni tra tuple di relazioni diverse.

La presenza di valori nulli in una relazione, come nella relazione degli esami, può rendere inutilizzabili le informazioni. Inoltre, un numero eccessivo di valori nulli in una relazione può sollevare dubbi sull'identità delle tuple stesse. È quindi cruciale controllare la presenza di valori nulli nelle relazioni, specificando che sono ammessi solo su alcuni attributi e non su altri. Alla fine, vedremo che è possibile definire criteri per individuare quali attributi non dovrebbero contenere valori nulli.

Vincoli di Integrità

In una base di dati, è fondamentale evitare situazioni problematiche come quelle descritte in precedenza. A tal fine, è stato introdotto il concetto di **vincolo di integrità**, che rappresenta una proprietà che deve essere soddisfatta dalle istanze per **garantire informazioni corrette per l'applicazione**. Ogni vincolo può essere considerato come un predicato che associa a ogni istanza un valore di vero o falso. Se il predicato assume il valore vero, si afferma che l'istanza soddisfa il vincolo. In generale, a uno schema di base di dati è associato un insieme di vincoli e si considerano corrette (o lecite, o ammissibili) le istanze che soddisfano tutti i vincoli. Per ciascuno dei casi precedentemente discussi, potrebbe essere introdotto un vincolo per vietare la situazione indesiderata.

I vincoli possono essere classificati in base agli elementi della base di dati che ne sono coinvolti. Distinguiamo due categorie, di cui la prima presenta alcuni casi particolari:

Vincoli intrarelazionali: questi vincoli sono definiti rispetto a singole relazioni della base di dati. I primi tre casi precedentemente discussi corrispondono a vincoli intrarelazionali. Talvolta, il coinvolgimento riguarda le tuple (o addirittura i valori) separatamente l'una dall'altra:

- **Vincolo di tupla**: è un vincolo che può essere valutato su ciascuna tupla indipendentemente dalle altre. I vincoli relativi ai primi due casi rientrano in questa categoria.
- **Vincolo su valori o vincolo di dominio**: è un caso ancora più specifico, in cui si impone una restrizione sul dominio di un attributo. Ad esempio, nel caso in cui sono ammessi solo valori dell'attributo Voto compresi fra 18 e 30.

Vincoli interrelazionali: questi vincoli coinvolgono più relazioni. Ad esempio, nella situazione indesiderata del quarto caso, è possibile vietare la situazione richiedendo che un numero di matricola compaia nella relazione ESAMI solo se è presente nella relazione STUDENTI.

Vincoli di tupla

Come abbiamo discusso, i vincoli di tupla esprimono condizioni sui valori di ciascuna tupla, indipendentemente dalle altre. Una sintassi utile per esprimere vincoli di questo tipo consente di definire espressioni booleane, che utilizzano connettivi logici come AND, OR e NOT, con atomi che confrontano (con operatori di uguaglianza, disuguaglianza e ordinamento) valori di attributo o espressioni aritmetiche su valori di attributo.

I vincoli violati identificati possono essere descritti come segue:

$$(Voto > 18)\ AND\ (Voto < 30)$$

$$NOT\ (Lode = 'lode')\ OR\ (Voto = 30)$$

In particolare, il secondo vincolo stabilisce che la lode è ammissibile solo se il voto è pari a 30, affermando che o non c'è la lode, oppure il voto è pari a 30. Il primo vincolo è, infatti, un vincolo di dominio, in quanto coinvolge un solo attributo.

La definizione di vincolo di tupla può includere anche espressioni più complesse, purché siano definite sui valori delle singole tuple. Ad esempio, su una relazione con lo schema:

$$\boldsymbol{PAGAMENTI(Data, Importo, Ritenute, Netto)}$$

è possibile definire il vincolo che richiede, come naturale, che il netto sia pari alla differenza tra l'importo originario e le ritenute, nel modo seguente:

$$Netto = Importo - Ritenute$$

Questo tipo di vincolo garantisce che le relazioni contabili siano mantenute in modo corretto, evitando errori nei dati inseriti.

Chiavi

In questo paragrafo discutiamo i vincoli di chiave, che sono fondamentali nel modello relazionale; senza di essi, il modello stesso risulterebbe privo di significato. Cominciamo con un esempio pratico. Nella relazione in esame, i valori dell'attributo Matricola sono tutti univoci; ciò significa che la matricola identifica univocamente gli studenti. Questo concetto è stato introdotto nelle università per consentire un riferimento chiaro e non ambiguo agli studenti. Analogamente, i dati anagrafici, come Cognome, Nome e Nascita, possono anch'essi identificare univocamente le persone.

Intuitivamente, una chiave è un insieme di attributi usato per identificare univocamente le tuple di una relazione. La definizione formale di chiave può essere espressa in due passi:

Un insieme K di attributi è considerato una **superchiave** di una relazione r se non esistono due tuple distinte t_1 e t_2 in r tali che $t_1[K] = t_2[K]$.

K è una **chiave** di r se è una **superchiave minimale**, ovvero non esiste un'altra superchiave K' di r che sia un sottoinsieme proprio di K.

Prendiamo in considerazione i seguenti esempi:

L'insieme {Matricola} è una superchiave e, poiché non contiene sottoinsiemi che siano anch'essi superchiavi, è anche una chiave.

L'insieme {Cognome, Nome, Nascita} è anch'esso una superchiave e, non avendo sottoinsiemi che siano superchiavi, è un'altra chiave.

L'insieme {Matricola, Corso} è una superchiave, ma non è una chiave minimale, poiché {Matricola} è già una superchiave.

L'insieme {Nome, Corso} non è una superchiave, poiché esistono tuple che condividono gli stessi valori su entrambi gli attributi.

Sebbene {Cognome, Corso} possa essere una chiave in questo specifico caso, non possiamo affermare che sia così in generale, poiché potrebbero esserci studenti con lo stesso cognome iscritti allo stesso corso.

Quando definiamo uno schema, associamo a esso vincoli di integrità che si applicano a tutte le istanze valide di quello schema. In particolare, per lo schema:

STUDENTI(Matricola, Cognome, Nome, Nascita, Corso)

si devono considerare come chiavi gli insiemi:

{Matricola}

{Cognome, Nome, Nascita}

Entrambe le relazioni di esempio soddisfano questi vincoli.

Le riflessioni sul concetto di chiave ci portano a evidenziare che ogni relazione e ogni schema di relazione devono avere almeno una chiave. Poiché ogni relazione è un insieme, è garantito che per ogni relazione r(X), l'insieme X di tutti gli attributi sia una superchiave. Se tale insieme non è chiave, possiamo ricercare sottoinsiemi fino a trovare una superchiave minimale.

Questo assicura che ogni relazione abbia una chiave e, di conseguenza, che esista almeno una chiave per ogni schema. La presenza di chiavi garantisce che i valori siano univocamente identificabili e facilita le corrispondenze tra dati di relazioni diverse. Ad esempio, nella relazione ESAMI, gli studenti sono identificati tramite i numeri di matricola, che fungono da chiave nella relazione STUDENTI, e i corsi sono identificati tramite i relativi codici, che rappresentano la chiave nella relazione CORSI.

Chiavi e valori Nulli

La presenza di valori nulli nelle relazioni può compromettere l'identificazione univoca delle tuple. Quando ci sono valori nulli nelle chiavi, diventa difficile distinguere tra tuple e stabilire riferimenti tra relazioni diverse. Ad esempio, se una tupla ha valori nulli per gli attributi che compongono la chiave, non può essere identificata e non possiamo sapere se un nuovo record si riferisce alla stessa entità.

Per affrontare questo problema, è fondamentale limitare i valori nulli nelle chiavi. In particolare, si vieta la presenza di valori nulli nella chiave primaria, garantendo così che ogni tupla possa essere identificata univocamente. In altre chiavi, invece, i valori nulli possono essere ammessi, a meno di esigenze specifiche. Gli attributi della chiave primaria sono spesso evidenziati per indicarne l'importanza.

In situazioni in cui non ci sono attributi sempre disponibili e identificativi, è possibile introdurre un attributo aggiuntivo, come un codice generato, per garantire l'univocità delle tuple. Questo approccio è comune in vari contesti, in cui codici identificativi (come numero di matricola o codice fiscale) sono utilizzati per assicurare un'identificazione chiara e per facilitare i riferimenti tra le entità.

Vincoli di integrità referenziale

I vincoli di integrità referenziale sono fondamentali per garantire la coerenza e l'integrità dei dati all'interno di un sistema di gestione di basi di dati relazionali. Questi vincoli stabiliscono e mantengono relazioni significative tra diverse tabelle, impedendo la creazione di collegamenti inconsistenti e migliorando la comprensibilità del sistema nel suo complesso.

Importanza dei Vincoli di Integrità Referenziale

- **Coerenza dei Dati**: I vincoli di integrità referenziale assicurano che ogni riferimento a un valore in un'altra tabella sia valido. Ad esempio, se una tabella delle infrazioni fa riferimento a un agente tramite un numero di matricola, questo numero deve esistere nella tabella degli agenti. Ciò evita situazioni in cui un'infrazione si riferisce a un agente inesistente, riducendo il rischio di confusione e errori nei rapporti.
- **Prevenzione dei Dati Orfani**: Questi vincoli impediscono la creazione di record "orfani", ossia record in una tabella che non possono essere collegati a record in un'altra tabella. Ad esempio, se un agente viene rimosso

dalla tabella degli agenti, tutte le infrazioni associate devono essere anch'esse rimosse o aggiornate. Senza i vincoli, si rischierebbe di avere infrazioni che non possono più essere collegate a nessun agente.

- **Chiarezza e Comprensibilità**: I vincoli di integrità referenziale forniscono una struttura chiara e comprensibile per i dati. Quando i riferimenti sono ben definiti, è più facile per gli sviluppatori e gli utenti capire come le tabelle si collegano tra loro, facilitando la manutenzione e l'aggiornamento della base di dati.
- **Operazioni di Manutenzione**: Durante la manutenzione dei dati, come l'aggiornamento o l'eliminazione di record, i vincoli di integrità referenziale forniscono linee guida su come procedere. Se un agente viene trasferito o rimosso, il sistema può impedire automaticamente l'eliminazione se ci sono infrazioni associate, avvisando l'utente della presenza di riferimenti attivi.

Definizione e Implementazione dei Vincoli

Un vincolo di integrità referenziale viene definito tra un insieme di attributi in una tabella e una chiave primaria in un'altra. Affinché questo vincolo sia soddisfatto, ogni valore dell'insieme di attributi deve corrispondere a un valore della chiave primaria nell'altra tabella.

Chiave Primaria e Chiave Esterna: La chiave primaria identifica in modo univoco ogni record in una tabella, mentre la chiave esterna stabilisce un legame con la chiave primaria di un'altra tabella. È fondamentale che i valori nella chiave esterna corrispondano a quelli nella chiave primaria per mantenere l'integrità referenziale.

Ordine degli Attributi: Quando le chiavi primarie sono composte da più attributi, è essenziale rispettare l'ordine di questi attributi. Se una chiave primaria è composta da "Prov" e "Numero", i valori di riferimento devono seguire questo ordine per garantire che la corrispondenza sia valida.

Modello Entità - Relazione (ER)

Nell'ambito della progettazione dei database, il modello entità-relazione (o modello entità-associazione; più comunemente modello E-R) è un modello teorico per la rappresentazione concettuale e grafica dei dati a un alto livello di astrazione, formalizzato da Peter Chen nel 1976.

Il modello entità-relazione viene spesso utilizzato nella prima fase della progettazione di una base di dati, in cui è necessario tradurre le informazioni risultanti dall'analisi di un determinato dominio in uno schema concettuale, detto diagramma entità-relazione (o diagramma E-R).

Nella progettazione ingegneristica delle basi di dati si distinguono tre livelli indipendenti e consecutivi di progettazione: progettazione concettuale, progettazione logica e progettazione fisica. Il modello E-R è la tecnica principale per la fase di progettazione concettuale, mentre il modello relazionale è utilizzato per la progettazione logica. Solo nella fase di progettazione fisica si prendono in considerazione software e hardware applicativi, proprietari e non, presenti sul mercato.

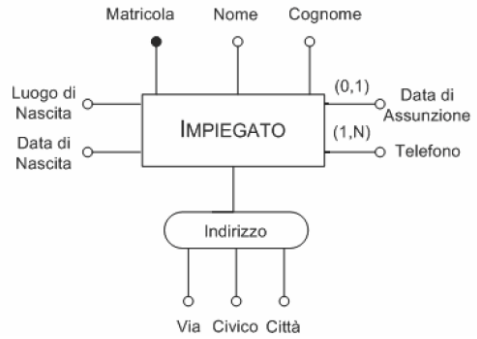
Generalità

Il modello E-R si basa su un insieme di concetti molto vicini alla realtà di interesse: quindi facilmente intuibili dai progettisti (e in genere considerati sufficientemente comprensibili e significativi anche per i non-tecnici), ma non implementabili sugli elaboratori. Infatti, pur essendo orientato alla progettazione di basi di dati, il modello prescinde dai criteri specifici di organizzazione fisica dei dati persistenti nei sistemi informatici. Esistono tecniche per la traduzione dei concetti ad alto livello (meglio comprensibili per gli umani) in concetti di più basso livello tipici dei vari modelli logici (ad esempio il modello relazionale) implementati nei diversi DBMS esistenti.

Il modello E-R ha rappresentato per lungo tempo (e forse ancora oggi) uno degli approcci più solidi per la modellazione di domini applicativi in ambito informatico; per questo motivo, è stato spesso usato anche al di fuori del contesto della progettazione di database, ed è stato utilizzato come modello di riferimento per numerose altre notazioni per la modellazione. Al modello E-R era ispirata, tra l'altro, la notazione OMT poi confluita in UML.

Tramite una superchiave identificativa (campi: ID_codice padre, ID_codice figlio), lo schema Entità-Associazione rappresenta un grafo ad albero su un numero di livelli a piacere (in particolare anche una distinta base), assai diffusa nel mondo informatico.

Costrutti principali del modello



Entità

Le entità rappresentano classi di oggetti, fatti, cose o persone che condividono proprietà comuni e hanno un'esistenza autonoma nell'applicazione di interesse. Un'occorrenza di un'entità è un'istanza della classe rappresentata dall'entità. Un'occorrenza di entità ha un'esistenza indipendente dalle sue proprietà, il che differenzia il modello E-R dal modello relazionale, dove non è possibile rappresentare un oggetto senza conoscere alcune sue proprietà. In uno schema, ogni entità ha un nome univoco e viene rappresentata graficamente con un rettangolo contenente il nome dell'entità.

Associazione

Le associazioni (o relazioni) rappresentano un legame tra due o più entità. Il numero di entità coinvolte è definito dal grado dell'associazione, con una prevalenza di associazioni di grado due nei buoni schemi E-R. È possibile creare associazioni tra un'entità e se stessa (associazione ad anello) e collegare le stesse entità con più associazioni. Graficamente, l'associazione è rappresentata da un rombo contenente il nome dell'associazione, che può essere un verbo o un sostantivo.

Attributi

Le entità e le associazioni possono essere descritte tramite attributi, che rappresentano le proprietà condivise dagli oggetti della stessa classe entità o associazione. Gli attributi riflettono il livello di dettaglio con cui si vogliono rappresentare le informazioni. Per ogni classe di entità o associazione, si definisce una chiave, ovvero un insieme minimale di attributi che identificano univocamente un'istanza. Gli attributi sono rappresentati graficamente da ellissi, e la chiave primaria viene indicata sottolineando o cerchiando il nome dell'attributo.

Tipi di Attributi di un Entità

attributi semplici: sono descritti per mezzo di linee terminate da cerchi e da nomi;

identificatori delle entità: sono usati come “strumento” per l’identificazione univoca delle occorrenze dell’entità, vengono indicati con cerchietto pieno;

attributi composti: sono identificati tramite i loro attributi componenti;

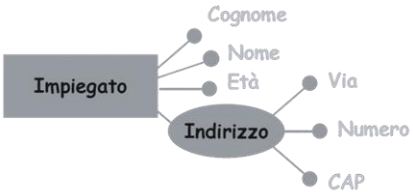
attributi multivalore: sono costituiti da un insieme di valori ed indicati tramite due numeri che esprimono la cardinalità dell’attributo;

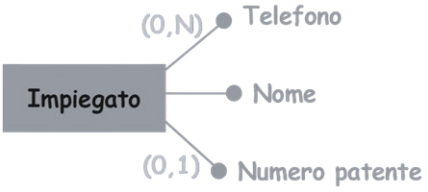
attributi opzionali: cioè con cardinalità(0, 1); sono usati quando non è necessario specificare un valore dell’attributo in quanto la sua assenza non compromette la significatività del concetto.

Attributo Composto

È l'insieme di attributi di una medesima entità o relazione che presentano affinità nel loro significato o uso.

Esempio: Via, Numero civico e CAP formano un Indirizzo





Cardinalità delle associazioni
Le cardinalità delle associazioni indicano, per ciascuna entità che partecipa a un'associazione, quante volte un'occorrenza di una di queste entità può essere legata a occorrenze delle altre entità coinvolte. Specificano il minimo e il massimo di occorrenze per ciascuna entità nell'associazione.

Cardinalità degli attributi
È possibile definire vincoli di cardinalità anche sugli attributi per indicare opzionalità o attributi multivalore. Se non viene specificato alcun vincolo, la cardinalità dell'attributo è generalmente (1,1). Ad esempio:

(0,1) NumeroPatente: un impiegato può avere una patente o nessuna.

(0,n) NumeroTelefono: un impiegato può avere molti numeri di telefono o nessuno.

(1,n) TitoloStudio: un impiegato deve avere almeno un titolo di studio, ma può averne molti.

Identificatori delle entità

Gli identificatori sono un sottoinsieme di attributi che identificano univocamente ogni occorrenza di un'entità. Ad esempio, l'attributo CodiceFiscale identifica univocamente ogni cittadino nell'entità CittadinoItaliano, in quanto non possono esistere due cittadini con lo stesso codice fiscale.

Identificatori esterni e interni

Gli **identificatori interni** sono attributi che appartengono all'entità stessa e che consentono di identificarla univocamente. Questi sono gli attributi che compongono la **chiave primaria** di un'entità.

Esempi:

Se consideriamo l'entità **Studente**, un identificatore interno potrebbe essere il **numero di matricola**: ogni studente ha un numero di matricola unico che lo distingue da tutti gli altri studenti.

Per l'entità **Prodotto**, potrebbe essere il **codice prodotto**, univoco per ogni prodotto.

Gli **identificatori esterni** sono attributi che fanno riferimento a un'altra entità e vengono utilizzati per rappresentare le **relazioni** tra diverse entità. Questi attributi fungono da **chiavi esterne (foreign key)** e sono utilizzati per stabilire legami tra entità diverse.

Esempi:

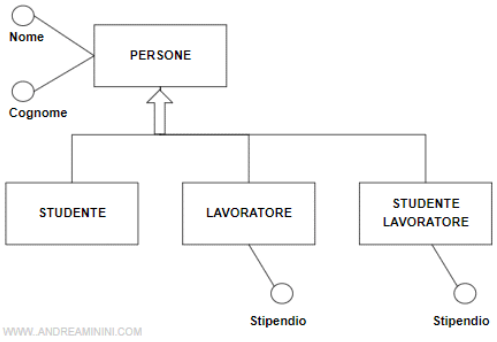
Se abbiamo un'entità **Ordine** e un'entità **Cliente**, un identificatore esterno nell'entità **Ordine** potrebbe essere il **codice cliente**: questa è una chiave esterna che fa riferimento al cliente che ha effettuato l'ordine.

Se consideriamo un database per un'azienda, l'entità **Dipendente** potrebbe avere un attributo **codice dipartimento**, che è un identificatore esterno e rappresenta il legame con l'entità **Dipartimento**.

11.4 Generalizzazioni
Le generalizzazioni rappresentano legami logici tra due o più entità, distinguendo:

Entità padre: una sola.

Entità figlie: una o più, che rappresentano "casi particolari" dell'entità padre.



Le entità figlie ereditano tutti gli attributi dell'entità padre, ma possono avere anche attributi propri che le distinguono. Ad esempio, l'entità padre "Persona" ha attributi come codice fiscale, cognome, nome e età, mentre le entità figlie "Uomo" e "Donna" possono avere attributi specifici.
Le generalizzazioni possono essere:

- **Totali**: se l'unione dei sottoinsiemi delle entità figlie costituisce l'insieme dell'entità padre (es. la generalizzazione da "Persona" a "Uomo" o "Donna").
- **Parziali**: se l'unione dei sottoinsiemi delle entità figlie non identifica l'insieme padre (es. "MezzoDiLocomozione" con "Bicicletta" e "Automobile").

Inoltre, le generalizzazioni possono essere:

- **Esclusive**: se l'intersezione tra i sottoinsiemi delle entità figlie è vuota.
- **Sovrapposte**: se l'intersezione non è vuota (es. un "Lavoratore" può essere sia "Impiegato" che "Studente").

Progettazione Logica del modello ER

La progettazione logica rappresenta un passaggio cruciale nel processo di sviluppo di un database, poiché consente di tradurre lo schema concettuale, solitamente rappresentato attraverso un modello Entità-Relazione (ER), in uno schema logico di dati. Questo schema logico si fonda su un modello specifico di rappresentazione dei dati, come il modello relazionale, e fornisce una struttura che è concreta e indipendente dai dettagli fisici dell'implementazione.

Traduzione dello Schema Concettuale

La fase di traduzione è fondamentale perché permette di trasformare le entità e le relazioni definite nel modello concettuale in tabelle, colonne e relazioni nel modello logico. Tuttavia, non tutte le caratteristiche del modello ER possono essere tradotte direttamente in un modello logico. Per esempio, gli attributi composti, che possono contenere più valori all'interno di un singolo attributo, e gli attributi multivalore, che consentono di memorizzare più istanze di un attributo, non possono essere direttamente rappresentati nel modello relazionale. Pertanto, è necessario un processo di semplificazione dello schema concettuale.

Semplificazione dello Schema Concettuale

La semplificazione è necessaria per ottenere uno schema che possa essere facilmente tradotto e ottimizzato. Durante questo processo, gli attributi composti e multivalore devono essere trasformati in attributi semplici. Ad esempio, un attributo "Indirizzo" che contiene "Via", "Città" e "CAP" può essere scomposto in attributi distinti all'interno di una tabella. Allo stesso modo, gli attributi multivalore devono essere gestiti creando tabelle separate e definendo relazioni tra le entità.

Questo processo di semplificazione non solo rende più facile la traduzione, ma contribuisce anche a ottenere uno schema logico più ottimizzato. Un modello ottimizzato garantisce che i dati siano organizzati in modo da ridurre la ridondanza e migliorare l'efficienza delle operazioni di accesso e manipolazione dei dati.

Schema Logico Finale

Una volta completate le fasi di trasformazione e traduzione, il risultato finale è lo schema logico, che nel caso del modello relazionale si traduce in un insieme di tabelle da istanziare. Ogni tabella rappresenta un'entità, e le colonne all'interno delle tabelle rappresentano gli attributi di queste entità. È fondamentale definire le chiavi primarie per ogni

tabella, che serviranno per identificare univocamente ciascuna riga, e le chiavi esterne, che stabiliranno relazioni tra le tabelle.

In aggiunta alla creazione dello schema logico, è essenziale considerare la normalizzazione. Questo processo consiste nell'analizzare le tabelle per eliminare la ridondanza e assicurarsi che i dati siano memorizzati in modo coerente. La normalizzazione si basa su una serie di forme normali, ciascuna delle quali fornisce linee guida per strutturare le tabelle e le relazioni in modo efficiente.

Vincoli di Integrità

Oltre alla struttura delle tabelle, è necessario definire i vincoli di integrità. Questi vincoli garantiscono la coerenza e l'affidabilità dei dati nel database. I vincoli di integrità possono includere vincoli di chiave primaria, vincoli di chiave esterna, vincoli di unicità e vincoli di controllo sui valori degli attributi. È importante definire questi vincoli all'interno dello schema logico, ma anche includerli nei programmi di gestione del database per garantire che le regole siano rispettate durante le operazioni di inserimento, aggiornamento e cancellazione dei dati.

Trasformazione dallo Schema E-R nel modello Relazionale

La trasformazione dello schema concettuale in uno schema logico è una fase cruciale nel processo di progettazione di un database. Essa consiste in vari passaggi, tra cui l'eliminazione di attributi multivalore e composti, la scelta e l'aggiunta di identificatori primari. Queste operazioni mirano a garantire che il modello logico soddisfi i requisiti del modello relazionale e sia ottimizzato per l'accesso ai dati.

Eliminazione di Attributi Multivalore e Composti

Un primo passo nella trasformazione è l'eliminazione di attributi che non possono essere direttamente tradotti nello schema logico. Gli **attributi composti** e **multivalore** non rispettano la **prima forma normale (1NF)**, la quale richiede che ogni attributo contenga solo valori atomici e univoci.

Attributi Compositi: Un attributo composto è un attributo che può essere suddiviso in più attributi semplici. Per esempio, un attributo "Indirizzo" che include "Via", "Città" e "CAP" deve essere scomposto in tre attributi distinti: "Via", "Città", e "CAP". Questo processo è reiterabile, nel senso che ogni attributo composto può essere ulteriormente scomposto in attributi semplici, se necessario.

Attributi Multivalore: Un attributo multivalore è un attributo che può contenere più valori per una singola istanza dell'entità. Per esempio, se un'entità "Studente" ha un attributo "Lingue Parlate" che può includere più lingue, si deve introdurre una nuova entità, ad esempio "Lingua", e creare una relazione uno a molti tra "Studente" e "Lingua". Questa relazione consente di gestire i dati in modo più efficace, evitando la duplicazione e garantendo una rappresentazione chiara delle informazioni.

Scelta e Aggiunta degli Identificatori Primari

Una volta eliminati gli attributi multivalore e composti, il passo successivo è la scelta dell'identificatore principale per ogni entità. L'identificatore principale, o chiave primaria, è essenziale perché su di esso si basano le relazioni tra le entità e perché i sistemi di gestione di database (DBMS) utilizzano le chiavi primarie per creare indici di accesso ai dati.

Criteri di Scelta degli Identificatori Primari:

- **Rispetto del Principio di Unicità:** Un identificatore deve garantire che ogni istanza dell'entità possa essere univocamente identificata. Non possono essere scelti identificatori basati su attributi che ammettono valori nulli, poiché questi non garantiscono l'accesso a tutte le occorrenze dell'entità.
- **Criteri di Efficienza:** Si preferisce utilizzare identificatori composti da un minor numero di attributi. Questo perché i DBMS generano indici più piccoli e gestibili per chiavi primarie con meno attributi, il che porta a un'occupazione ridotta di spazio e a un accesso più efficiente. Inoltre, si prediligono identificatori utilizzati da un maggior numero di operazioni, in modo che possano sfruttare gli indici già creati dal DBMS.

Traduzione dello Schema E-R nel Modello Relazionale

Al termine della fase di trasformazione, il progettista dispone di uno schema Entità-Relazione (E-R) semplificato e adattato alle esigenze del modello logico. Questo schema E-R costituisce la base per la traduzione nel modello relazionale, da cui verrà generato lo schema della base di dati. Questa traduzione è fondamentale perché consente di

creare una rappresentazione strutturata dei dati che può essere implementata efficacemente in un sistema di gestione di database (DBMS).

Risultati della Traduzione

Durante la traduzione nel modello relazionale, si esegue una trasformazione significativa:

- **Entità in Tabelle:** Ogni entità presente nel modello E-R viene tradotta in una tabella nel modello relazionale. Il nome della tabella corrisponde al nome dell'entità, ma viene generalmente utilizzato il plurale per riflettere che la tabella contiene molte istanze di quell'entità.
- **Associazioni in Tabelle:** Le associazioni tra entità, che rappresentano relazioni, vengono anch'esse tradotte in tabelle. Queste tabelle servono a connettere le entità coinvolte e a gestire le informazioni relative alle relazioni tra di esse.
- **Vincoli di Integrità Referenziali:** Tra le tabelle di associazioni e quelle delle entità collegate vengono stabiliti vincoli di integrità referenziale. Questi vincoli garantiscono che i dati rimangano coerenti, assicurando che un identificatore di un'entità possa essere trovato nella tabella collegata. In pratica, le tabelle delle relazioni includono gli identificatori di tutte le entità coinvolte.

Problematiche di Traduzione a Seconda della Cardinalità delle Associazioni

La traduzione delle associazioni presenta diverse problematiche in base alla cardinalità delle relazioni, che possono essere classificate come:

- **Uno a Uno (1:1):** In questo tipo di relazione, ogni istanza di un'entità è collegata a una sola istanza di un'altra entità. In questo caso, si può scegliere di mantenere i dati in una sola tabella oppure in due tabelle separate, a seconda delle esigenze di progettazione.
- **Uno a Molti (1:M):** Qui, un'entità può essere collegata a più istanze di un'altra entità, ma non viceversa. In questo caso, l'identificatore della tabella dell'entità "uno" viene inserito come chiave esterna nella tabella dell'entità "molti".
- **Molti a Molti (M:M):** Le associazioni molti a molti richiedono una gestione più complessa. In questo scenario, si deve creare una nuova tabella di associazione che contenga gli identificatori di entrambe le entità coinvolte, formando così una chiave composta. Questa tabella di associazione può anche contenere attributi specifici della relazione.

Traduzione delle Entità

Quando si traduce un'entità in una tabella, il risultato finale avrà le seguenti caratteristiche:

- **Nome della Tabella:** Il nome della tabella sarà il nome dell’entità al plurale, per rappresentare le molteplici istanze.
- **Attributi:** Gli attributi dell’entità diventeranno le colonne della tabella, mantenendo i nomi originali.
- **Chiave Primaria:** L’identificatore principale dell’entità viene definito come chiave primaria della tabella, garantendo l'unicità di ogni riga.

Traduzione delle Associazioni Molti a Molti

La traduzione di un’associazione molti a molti produce una tabella che ha:

- **Nome della Tabella:** Il nome della tabella sarà il nome dell’associazione al plurale.
- **Attributi:** Includerà gli attributi della relazione, se presenti, e gli identificatori di tutte le entità coinvolte. Questi identificatori formeranno la chiave della relazione. In alcuni casi, possono formare una superchiave, da cui è necessario scegliere un sottoinsieme appropriato per la chiave primaria.
- **Ridenominazione per Leggibilità:** Per migliorare la leggibilità degli schemi, può essere utile ridenominare gli attributi degli identificatori e rendere espliciti i nomi delle chiavi esterne.

Esempio

Ovvero:

IMPIEGATO(Matricola, Cognome, Stipendio)

PROGETTI(Codice, Nome, Budget)

PARTECIPAZIONE(Matricola, Codice, Data Inizio)

Per migliorare la comprensione della tabella PARTECIPAZIONI conviene rinominare gli attributi presi dalle entità ed esplicitare le chiavi esterne:

PARTECIPAZIONI(Impiegato:IMPIEGATI, Progetto:PROGETTI, Data Inizio)

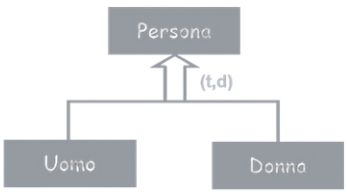
VINCOLI DI INTEGRITÀ REFERENZIALI

TRA	DI	E	DI
Impiegato	PARTECIPAZIONI	Matricola	IMPIEGATI
Progetto	PARTECIPAZIONI	Codice	PROGETTI

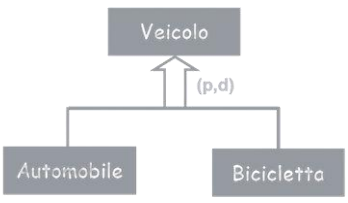
Vincoli delle Gerarchie

Le gerarchie nelle basi di dati possono essere classificate utilizzando due tipi fondamentali di vincoli: i **vincoli di copertura** e i **vincoli di disgiunzione**, che regolano il modo in cui le occorrenze delle sottoclassi si relazionano con quelle della superclasse.

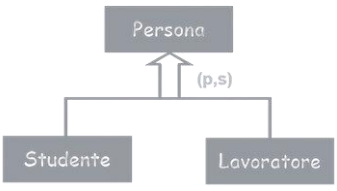
- I **vincoli di copertura** riguardano la distinzione tra generalizzazione totale o parziale. Una generalizzazione è **totale** quando ogni occorrenza della superclasse (classe padre) appartiene almeno a una delle sottoclassi (entità figlie). In altre parole, non esistono istanze che appartengano solo alla superclasse senza essere classificate in una sottoclasse. Al contrario, la generalizzazione è considerata **parziale** quando alcune occorrenze della superclasse non appartengono a nessuna delle sottoclassi, restando quindi escluse dalla suddivisione.
- I **vincoli di disgiunzione**, invece, determinano se una stessa occorrenza della superclasse può appartenere a più sottoclassi contemporaneamente. Se la generalizzazione è **disgiunta**, significa che ogni occorrenza della superclasse può essere parte di una sola sottoclasse; in altre parole, le sottoclassi non possono sovrapporsi. Se la generalizzazione è invece **sovrapposta**, un'istanza della superclasse può appartenere a più di una sottoclasse simultaneamente, permettendo quindi che le sottoclassi condividano alcune occorrenze.



Totale e Disgiunta



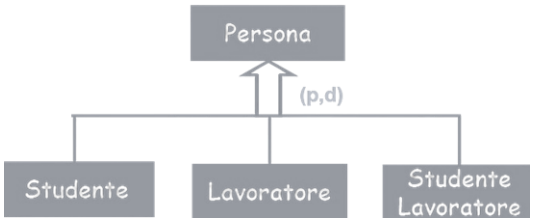
Parziale e Disgiunta



Parziale e Sovrapposta

Da sovrapposta ad esclusiva

Le **generalizzazioni sovrapposte** possono essere trasformate in **generalizzazioni esclusive** introducendo una o più entità figlie che rappresentano le sovrapposizioni tra le sottoclassi. In una generalizzazione sovrapposta, un'istanza della superclasse può appartenere a più sottoclassi contemporaneamente, creando potenziali ambiguità. Per eliminare queste sovrapposizioni e rendere la gerarchia esclusiva, si possono aggiungere nuove entità figlie che catturano esplicitamente queste combinazioni. In questo modo, ogni istanza della superclasse sarà associata solo a una sottoclasse, garantendo che le generalizzazioni siano disgiunte e mantenendo una rappresentazione più chiara e strutturata dei dati.



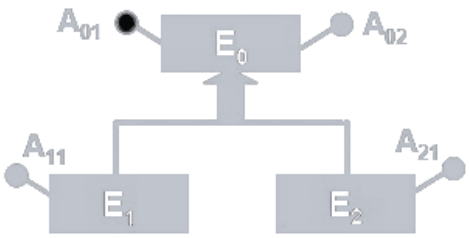
È sufficiente aggiungere l'entità StudenteLavoratore per ottenere una generalizzazione esclusiva da quella con i soli Studente e Lavoratore.

Risoluzione delle Gerarchie

Le **gerarchie di generalizzazione** sono strumenti potenti nella progettazione concettuale dei database, utilizzati per rappresentare in modo chiaro le relazioni e le dipendenze tra le diverse entità presenti nel mondo reale. Tuttavia, queste gerarchie non trovano una corrispondenza diretta nei modelli logici delle basi di dati, poiché non esiste un costrutto logico equivalente che le rappresenti immediatamente. Per implementarle, è necessario trasformare il costrutto concettuale della gerarchia in una combinazione di costrutti di base, come entità e relazioni. Questa trasformazione permette di mantenere la struttura delle informazioni attraverso la creazione di tabelle che rappresentano le diverse sottoclassi, insieme alle relazioni che le collegano alla superclasse, rispettando i vincoli di integrità e garantendo una gestione efficace dei dati.

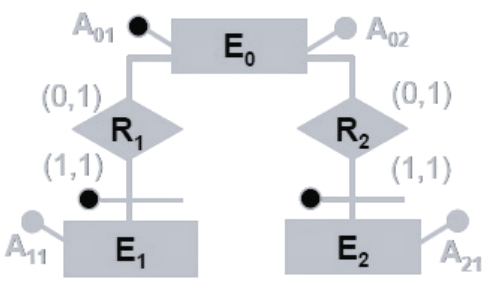
Tre alternative

Il **costrutto di generalizzazione** può essere trasformato in tre modi principali durante la progettazione logica di un database. Il primo approccio è l'**accorpamento nell'entità padre delle entità figlie**, dove la generalizzazione assorbe le specializzazioni, combinando in un'unica tabella tutti gli attributi dell'entità padre e delle sue sottoclassi. Il secondo metodo consiste nell'**accorpamento dell'entità padre nelle entità figlie**, in cui ogni sottoclasse eredita direttamente gli attributi della superclasse, creando tabelle separate per ciascuna entità figlia con tutte le informazioni necessarie.



Infine, la terza opzione è la **sostituzione della generalizzazione con relazioni**, dove la generalizzazione viene tradotta in una serie di relazioni uno a uno che collegano l'entità padre a ciascuna entità figlia, mantenendo così una separazione ma legando strettamente i dati tra padre e figlie. Ogni approccio ha vantaggi e svantaggi in termini di efficienza e semplicità di gestione.

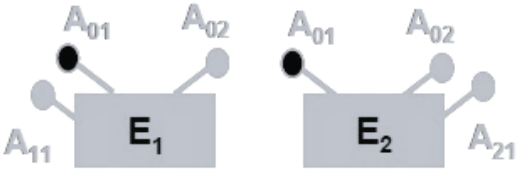
Prima Soluzione



La terza modalità di trasformazione, identificata come **trasformazione c)** del costrutto di generalizzazione, ossia la **sostituzione della generalizzazione con relazioni**, è solitamente applicata in presenza di **generalizzazioni parziali**. Questo approccio è particolarmente vantaggioso quando ci sono operazioni distinte che devono essere eseguite sia sull'entità padre che sulle entità figlie. Uno dei principali benefici di questa trasformazione è il risparmio di memoria, poiché non vengono generati valori nulli e non è necessario l'uso di un attributo di tipo, come un campo **TIPO** per distinguere le sottoclassi.

Tuttavia, comporta un numero maggiore di accessi al database, poiché per operare su una delle entità figlie (ad esempio, E1 o E2), è necessario accedere prima all'entità padre (E0). Nonostante l'aumento degli accessi, questa soluzione offre una rappresentazione chiara e precisa delle relazioni tra la superclasse e le sottoclassi, mantenendo i dati organizzati in maniera efficiente.

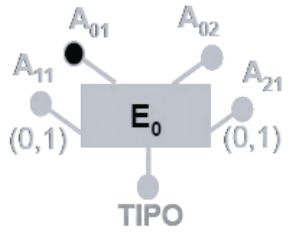
Seconda Soluzione



La trasformazione in cui la **generalizzazione scompare** e rimangono solo le **specializzazioni** implica che tutte le entità figlie ereditano gli attributi dell'entità padre. Questa modalità di trasformazione, identificata come la **trasformazione b)**, può essere applicata solo in presenza di **generalizzazioni totali**. In caso contrario, le occorrenze dell'entità padre (E0) che non appartengono né a E1 né a E2 non sarebbero rappresentate, risultando in una perdita di dati.

Questo approccio è vantaggioso quando si devono gestire operazioni distinte e specializzate su E1 ed E2, in quanto permette di lavorare separatamente sulle sottoclassi. Inoltre, consente un risparmio di memoria, poiché non si creano valori nulli e non è necessario un attributo di tipo, come il campo **TIPO**, per distinguere le sottoclassi. Tuttavia, l'applicabilità limitata alle generalizzazioni totali rende questo metodo adatto solo a specifici contesti.

Terza Soluzione



L'**accorpamento delle specializzazioni nella generalizzazione** prevede che l'entità padre inglobi le entità figlie, assumendone tutti gli attributi. La **trasformazione a)** si applica in situazioni in cui le operazioni non distinguono tra le occorrenze e gli attributi delle varie entità figlie, permettendo di trattare le informazioni come un insieme unico. Questo approccio riduce il numero di accessi al database, poiché tutte le occorrenze e gli attributi, precedentemente distribuiti tra l'entità padre e le entità figlie, sono ora concentrati in un'unica entità.

Tuttavia, questa soluzione introduce la problematica dei **valori nulli**, poiché ci saranno occorrenze dell'entità padre che non trovano corrispondenza in nessuna delle entità figlie (generalizzazione parziale), così come occorrenze di una sottoclasse che non hanno riscontro nell'altra sottoclasse. Inoltre, è necessario introdurre un attributo supplementare per distinguere le occorrenze provenienti dalle diverse entità figlie. Questo comporta una maggiore occupazione di memoria, non solo a causa dei campi con valori nulli, ma anche per la presenza dell'attributo aggiuntivo utilizzato per differenziare le entità. Sebbene questa trasformazione possa semplificare alcune operazioni, risulta meno efficiente in termini di utilizzo dello spazio.

La soluzione mista

Questo approccio, particolarmente adatto per **gerarchie parziali e sovrapposte**, prevede la sostituzione delle sottoclassi con un'unica sottoclasse in associazione 1:1 con la superclasse. In questo scenario, la sottoclasse unica eredita tutti gli attributi delle sottoclassi originarie e sarà identificata esternamente dalla superclasse. Questo implica che la nuova entità figlia sarà l'unica sottoclasse e conterrà:

Tutti gli attributi precedentemente appartenenti alle diverse sottoclassi.

Tutte le associazioni a cui partecipavano le sottoclassi originali.

In aggiunta, viene introdotto un attributo **"Tipo"** che serve a distinguere a quale sottoclasse ciascuna occorrenza appartiene. Questo attributo permette di mantenere la distinzione tra le occorrenze provenienti dalle diverse sottoclassi, pur semplificando la struttura gerarchica in un'unica sottoclasse.

Algebra relazionale

Operazioni insiemistiche [Notazione Classica]

L'algebra relazionale è un insieme di operazioni che permette di manipolare e interrogare i dati all'interno di un **database relazionale**. Queste operazioni lavorano su relazioni (ossia tabelle) e producono come risultato altre relazioni. In particolare, l'algebra relazionale si basa fortemente sulle operazioni insiemistiche, dato che le relazioni possono essere considerate come **insiemi di tuple** (righe di dati).

Le principali operazioni insiemistiche che possiamo applicare a due relazioni sono:

Unione

Intersezione

Differenza

Queste operazioni richiedono che le due relazioni abbiano lo stesso **schema**, cioè che abbiano gli stessi attributi con lo stesso tipo e lo stesso ordine. Vediamo nel dettaglio come funzionano ciascuna di queste operazioni.

Unione di Relazioni

Definizione

L'**unione** tra due relazioni $\boldsymbol{r_1}$ e $\boldsymbol{r_2}$ è un'operazione che restituisce una nuova relazione contenente tutte le tuple presenti.

Notazione $r = r_1 \cup r_2$

Formalizzazione $r = \{t \mid t \in r_1 \vee t \in r_2\}$

Intersezione di Relazioni

Definizione

L'**intersezione** tra due relazioni $\boldsymbol{r_1}$ e $\boldsymbol{r_2}$ è un'operazione che restituisce una nuova relazione contenente solo le tuple presenti in entrambi.

Notazione $r = r_1 \cap r_2$

Formalizzazione $r = \{t \mid t \in r_1 \wedge t \in r_2\}$

Differenza tra Relazioni

Definizione

La **differenza** tra due relazioni $\boldsymbol{r_1}$ e $\boldsymbol{r_2}$ è un'operazione che restituisce una nuova relazione contenente solo le tuple presenti in **una ma non in entrambe**.

Notazione $r = r_1 - r_2$

Formalizzazione $r = \{t \mid t \in r_1 \wedge t \in /r_2\}$

Operazioni di modifica dello stato

Nel contesto del modello relazionale, le operazioni sulle **relazioni** (ossia tabelle) sono essenziali per la gestione e manipolazione dei dati. Le operazioni di base considerate includono **inserimento**, **cancellazione** e **modifica** di tuple (righe) all'interno di una relazione. In questa premessa, ci focalizziamo su come queste operazioni siano definite inizialmente per una singola tupla e, successivamente, si estendano a operazioni che coinvolgono più tuple attraverso query SQL.

Modello di Operazioni su una Singola Tupla

Inizialmente, il modello relazionale considerava solo operazioni che coinvolgono una singola tupla alla volta all'interno di una relazione. Questo approccio è utile per operazioni semplici e di base sulle tabelle, come:

Insert: inserimento di una nuova tupla nella relazione.

Delete: rimozione di una tupla esistente dalla relazione.

Update: modifica dei valori di una tupla già presente nella relazione.

Operazione di Insert

L'operazione di **insert** permette di aggiungere una nuova tupla alla relazione esistente.

Obiettivo: Inserire una nuova riga (tupla) in una tabella.

Sintassi concettuale: $insert(r, t)$

Esempio:

```
INSERT INTO Studenti (Nome, Cognome, Matricola)
VALUES ('Giulia', 'Neri', 1004);
```

Operazione di Delete

L'operazione di **delete** rimuove una tupla specifica dalla relazione, basandosi su una condizione che identifica quale tupla eliminare.

Obiettivo: Rimuovere una riga (tupla) esistente dalla tabella.

Sintassi concettuale: $delete(r, condizione)$

Esempio:

```
DELETE FROM Studenti
WHERE Matricola = 1004;
```

Operazione di Update

L'operazione di **update** consente di modificare i valori di una tupla specifica nella relazione, basandosi su una condizione.

Obiettivo: Modificare i valori di una o più colonne di una riga (tupla) esistente nella tabella.

Sintassi concettuale: $update(r, attributo, valore\_nuovo, condizione)$

Esempio:

```
UPDATE Studenti
SET Cognome = 'Verdi'
WHERE Matricola = 1004;
```

Operazioni Relazionali

Nel modello relazionale, le operazioni classiche che permettono di estrarre e manipolare dati dalle relazioni sono **proiezione**, **selezione** e **join**. Queste operazioni costituiscono la base dell'algebra relazionale, che descrive in maniera formale il modo in cui si manipolano insiemi di tuple (righe) in un database relazionale.

Proiezione

Definizione Procedurale

La **proiezione** è un'operazione che consente di selezionare solo un sottoinsieme di attributi da una relazione, eliminando gli altri. In altre parole, la proiezione crea una nuova relazione che contiene solo alcune colonne della relazione originale.

Notazione

Se r è una relazione e $A_1, A_2, \dots, A_k$ sono attributi di r, la proiezione su questi attributi si indica come:

$$\pi_{A_1, A_2, \dots, A_k}(r)$$

In SQL, l'operazione di proiezione si realizza selezionando specifiche colonne di una tabella con il comando **SELECT**.

```
SELECT A1, A2, ..., Ak
FROM r;
```

Esempio

Supponiamo di avere una tabella **Studenti** con i seguenti attributi: **Nome**, **Cognome**, **Matricola**, **AnnoIscrizione**. Se vogliamo ottenere solo i nomi e i cognomi degli studenti, possiamo eseguire la seguente proiezione:

```
SELECT Nome, Cognome

FROM Studenti;
```

Selezione

Definizione Procedurale

La **selezione** è un'operazione che consente di estrarre tutte le tuple di una relazione che soddisfano una determinata condizione. Si basa su un **predicato** che filtra le righe in base ai valori dei loro attributi.

Notazione

Se r è una relazione e condizione è un predicato logico applicabile alle tuple di r, la selezione si indica come:

$$\sigma_{condizione}(r)$$

In SQL, la selezione si realizza utilizzando il comando **WHERE** per specificare la condizione logica.

```
SELECT *

FROM r

WHERE condizione;
```

Esempio

Supponiamo di avere la tabella **Studenti** e vogliamo selezionare tutti gli studenti che si sono iscritti nell'anno 2023. In SQL, eseguiremo la seguente query:

```
SELECT *

FROM Studenti

WHERE AnnoIscrizione = 2023;
```

Soddisfacimento di una Condizione di Selezione nel Modello Relazionale

Nel modello relazionale, la **selezione** filtra le tuple di una relazione in base a una **condizione**. Le condizioni possono essere atomiche o combinate.

1. Condizione Atomica

Una **condizione atomica** è un confronto tra attributi o tra un attributo e una costante:

$A\theta B$: confronta due attributi con $\theta \in \{=, \neq, <, >, \leq, \geq\}$.

$A = c$: confronta un attributo A con una costante c.

2. Condizione di Selezione

Una **condizione di selezione** può essere atomica o composta tramite operatori logici:

$vc_1 \wedge vc_2$: vera se entrambe le condizioni sono vere (AND).

$vc_1 \vee vc_2$: vera se almeno una condizione è vera (OR).

$\neg vc_1$: vera se la condizione è falsa (NOT).

3. Soddisfacimento di una Condizione

Una tupla t soddisfa una condizione χ se:

Confronto tra attributi: $t[A]\ \theta\ t[B]$.

Confronto con costanti: $t[A]\ \theta\ c$.

Combinazioni complesse si valutano tramite gli operatori logici. Se tutte le condizioni sono soddisfatte, la tupla è selezionata.

Confronto tra Stringhe in SQL: Utilizzo dell'Operatore LIKE

Nell'ambito della gestione dei dati, il confronto tra stringhe è un'operazione fondamentale, specialmente quando si lavora con database relazionali come SQL. Le stringhe, che rappresentano sequenze di caratteri, possono essere confrontate in vari modi, e uno degli operatori più utili in SQL per questo scopo è l'operatore LIKE. Questo operatore consente di eseguire confronti basati su modelli, facilitando la ricerca di stringhe che soddisfano determinati criteri.

Operatore LIKE

L'operatore LIKE viene utilizzato per cercare un pattern specifico all'interno di una colonna di tipo stringa. A differenza dei confronti standard che richiedono una corrispondenza esatta, LIKE consente di utilizzare caratteri jolly per rendere la ricerca più flessibile.

Caratteri Riservati

Due caratteri speciali sono comunemente usati con l'operatore LIKE:

Carattere di percentuale (%): Rappresenta un numero arbitrario di caratteri. Può essere usato all'inizio, alla fine o nel mezzo di una stringa per indicare che non ci sono restrizioni su cosa può apparire in quella posizione.

Esempio: LIKE 'A%' troverà tutte le stringhe che iniziano con 'A'.

Carattere di sottolineatura (_): Rappresenta un singolo carattere. È utile quando si desidera specificare che esattamente un carattere deve apparire in una posizione specifica.

Esempio: LIKE '_e_' troverà tutte le stringhe che hanno 'e' come secondo carattere.

Operatori Aritmetici e di Ordinamento in SQL

Operatori Aritmetici

Gli operatori aritmetici sono utilizzati per effettuare calcoli matematici su colonne di tipo numerico. I principali operatori aritmetici in SQL sono:

Addizione (+): somma due valori.

Sottrazione (-): sottrae un valore da un altro.

Moltiplicazione (*): moltiplica due valori.

Divisione (/): divide un valore per un altro.

Esempio di Utilizzo degli Operatori Aritmetici

Supponiamo di voler visualizzare il nome, il cognome e lo stipendio incrementato di 10 euro per tutti gli impiegati di un'azienda. La query SQL potrebbe essere la seguente:

```
SELECT Nome, Cognome, Stipendio + 10 AS StipendioIncrementato

FROM ANAGRAFICA;
```

Ordinamento dei Risultati

La clausola ORDER BY viene utilizzata per ordinare le righe restituite da una query. Si può ordinare in ordine crescente (ASC) o decrescente (DESC). Per impostazione predefinita, l'ordinamento è crescente.

Esempio di Ordinamento dei Risultati

Per visualizzare i cognomi degli studenti ordinati alfabeticamente e i nomi ordinati in senso decrescente, la query sarà:

```
SELECT Cognome, Nome

FROM ANAGRAFICA

ORDER BY Cognome ASC, Nome DESC;
```

Prodotto Cartesiano

Il prodotto cartesiano è un concetto fondamentale in algebra relazionale, utilizzato per combinare tutte le possibili coppie di tuple provenienti da due relazioni diverse. Questo concetto è alla base di molte operazioni in SQL, dove le tabelle (relazioni) possono essere combinate per generare nuovi insiemi di dati.

Definizione

Data due relazioni r_1 e r_2 definite sugli schemi $R(X_1)$ e $R(X_2)$ con $X_1 \cap X_2 = \emptyset$ (ovvero, non ci sono attributi in comune tra le due relazioni), il prodotto cartesiano restituisce un'istanza di relazione r definita su $R(X_1 \cup X_2)$.

Rappresentazione Formale

La relazione r è definita come:

$$r = r_1 \times r_2 = \{t = \langle t_1, t_2 \rangle \mid t_1 \in r_1 \wedge t_2 \in r_2\}$$

Dove t_1 è una tupla di r_1 e t_2 è una tupla di r_2. Ogni tupla t nel risultato finale r è la concatenazione di una tupla da r_1 e una da r_2.

Dot Notation

La **dot notation** (notazione a punti) è un metodo utile per distinguere gli attributi comuni tra due relazioni, specialmente quando si esegue un prodotto cartesiano tra di esse. In un prodotto cartesiano, ogni riga della prima relazione viene combinata con ogni riga della seconda, il che può portare a conflitti di nome se entrambe le relazioni hanno attributi con lo stesso nome.

Per evitare ambiguità, utilizziamo la dot notation per fare riferimento a ciascun attributo:

r1.id: riferimento all'attributo **id** della relazione **r1**

r2.id: riferimento all'attributo **id** della relazione **r2**

r1.nome: riferimento all'attributo **nome** della relazione **r1**

r2.codice: riferimento all'attributo **codice** della relazione **r2**

Theta Join ed Equi Join

La **THETA JOIN** è un'operazione fondamentale in algebra relazionale che consente di combinare due relazioni in base a una condizione specifica, espressa tramite un operatore di confronto (θ) tra gli attributi delle due relazioni. Questa operazione può includere diversi tipi di confronti, come uguaglianze, maggiore, minore, e così via.

Definizione di THETA JOIN

Data due relazioni:

r1 su $R(X_1)$

r2 su $R(X_2)$

con $X_1 \cap X_2 = \emptyset$ (ovvero, gli attributi di r_1 e r_2 non si sovrappongono), la THETA JOIN può essere rappresentata come:

$$\chi \equiv r_1 \theta r_2$$

dove θ è una condizione che può essere un'uguaglianza o un'altra relazione di confronto.

EQUI JOIN

Se la condizione θ è un'uguaglianza tra i valori di un attributo di $\boldsymbol{r_1}$ e quelli di $\boldsymbol{r_2}$, si parla di **EQUI JOIN**. In questo caso, la condizione di join è espressa come:

$$r_1.attr = r_2.attr$$

In SQL, l'operazione di **JOIN** è fondamentale per combinare righe di due o più tabelle in base a una condizione comune. Ci sono due modi principali per eseguire un JOIN: **in modo implicito** e **in modo esplicito**. Vediamo entrambe le modalità in dettaglio.

1. JOIN Implicito

Il **JOIN implicito** si ottiene applicando la definizione di equi-join come prodotto cartesiano, seguito da una selezione. In questa modalità, le tabelle sono elencate nella clausola **FROM** e la condizione di join è specificata nella clausola **WHERE**.

```
SELECT *

FROM r1, r2

WHERE Condizione;
```

2. JOIN Esplicito

Il **JOIN esplicito**, d'altra parte, richiede una sintassi più chiara e espressiva in cui l'operazione di join è definita direttamente nella query. Utilizza la clausola **JOIN** seguita da **ON** per specificare la condizione di join.

```
SELECT *

FROM r1 JOIN r2 ON Condizione;
```

15.9 Tuple Dondolanti

Il concetto di **"tupla dondolante"** (o **dangling tuple**) si riferisce a quelle righe di una relazione che non trovano corrispondenza nell'altra relazione durante un'operazione di **JOIN**, in particolare quando si utilizza un **EQUI JOIN**. Queste tuple rimangono "sospese" o "dondolanti" perché non hanno un legame diretto con le tuple dell'altra relazione.

Esempio di Tupla Dondolante

ID	Nome	Mansione
1	Mario Rossi	101
2	Anna Bianchi	102
3	Luca Verdi	103

Codice	Descrizione
101	Responsabile
102	Operativo
104	Amministratore

Risultato dell'EQUI JOIN

L'output di questa query sarà:

ID	Nome	Mansione	Codice	Descrizione
1	Mario Rossi	101	101	Responsabile
2	Anna Bianchi	102	102	Operativo

In questo caso, la terza tupla della tabella **ANAGRAFICA_DIPENDENTI** (Luca Verdi) non appare nel risultato dell'operazione di join, perché non esiste una mansione corrispondente (103) nella tabella **MANSIONI**.

Tupla Dondolante

Tupla dondolante: La terza tupla in **ANAGRAFICA_DIPENDENTI** (ID = 3, Nome = Luca Verdi, Mansione = 103) è considerata una tupla dondolante, poiché non ha alcun corrispettivo nella tabella **MANSIONI**. Questo significa che l'impiegato Luca Verdi non ha una mansione associata.

Implicazioni delle Tuple Dondolanti

Integrità dei Dati: Le tuple dondolanti possono indicare problemi di integrità nei dati, in particolare se ci si aspetta che ogni dipendente abbia una mansione valida.

Filtri e Query: È importante tener conto delle tuple dondolanti quando si scrivono query, poiché potrebbero portare a risultati non attesi.

Join Estesi: Se si desidera includere anche le tuple dondolanti nel risultato, si potrebbe utilizzare una **LEFT JOIN**, che garantisce di includere tutte le righe dalla tabella a sinistra (in questo caso, **ANAGRAFICA_DIPENDENTI**), anche se non ci sono corrispondenze nella tabella a destra (**MANSIONI**).

Natural Join

l **JOIN NATURALE** è una forma particolare di **THETA JOIN** in cui le condizioni di uguaglianza vengono applicate automaticamente su tutti gli attributi con lo stesso nome nelle due relazioni. Questo tipo di join è utile quando si desidera combinare le righe di due tabelle in base a colonne che condividono nomi e significato, senza dover specificare manualmente le condizioni di uguaglianza per ogni coppia di attributi.

Definizione di JOIN NATURALE

Data due relazioni **r1** e **r2**, se condividono un attributo comune AcAcAc (ad esempio, un ID, un codice o un nome), il JOIN NATURALE restituisce solo le righe in cui i valori di AcAcAc sono uguali in entrambe le tabelle.

Sintassi

La sintassi per un JOIN NATURALE è:

```
SELECT *

FROM r1 NATURAL JOIN r2;
```

Esempio di JOIN NATURALE

Supponiamo di avere due tabelle:

ID	Nome	Mansione
1	Mario Rossi	101
2	Anna Bianchi	102
3	Luca Verdi	103

Codice	Descrizione
101	Responsabile
102	Operativo
103	Manager

Se desideriamo eseguire un JOIN NATURALE sulle colonne **Mansione** di **ANAGRAFICA_DIPENDENTI** e **Codice** di **MANSIONI**, il risultato del JOIN NATURALE sarà:

```
SELECT *

FROM ANAGRAFICA_DIPENDENTI NATURAL JOIN MANSIONI;
```

Risultato del JOIN NATURALE

L'output della query sarà:

ID	Nome	Mansione	Codice	Descrizione
1	Mario Rossi	101	101	Responsabile
2	Anna Bianchi	102	102	Operativo
3	Luca Verdi	103	103	Manager

Join Esterno

Il **JOIN ESTERNO** è un'operazione fondamentale in SQL che consente di includere tutte le righe di una o entrambe le tabelle coinvolte in una query, anche se non esistono corrispondenze nelle altre tabelle. Questo è particolarmente utile per evitare di perdere dati significativi, specialmente quando si lavora con tuple dondolanti.

Tipi di JOIN ESTERNO

Ci sono tre tipi principali di JOIN ESTERNO:

- **JOIN ESTERNO SINISTRO (LEFT OUTER JOIN)**
- **JOIN ESTERNO DESTRO (RIGHT OUTER JOIN)**
- **JOIN ESTERNO COMPLETO (FULL OUTER JOIN)**

1. JOIN ESTERNO SINISTRO (LEFT OUTER JOIN)

Il **JOIN ESTERNO SINISTRO** restituisce tutte le righe dalla relazione a sinistra (rs) e le righe corrispondenti dalla relazione a destra (rd). Se non ci sono corrispondenze nella relazione a destra, i valori delle colonne provenienti da quest'ultima saranno NULL.

```
SELECT *

FROM rs LEFT OUTER JOIN rd ON condizione;
```

Esempio

Consideriamo di avere due tabelle:

ID	Nome
1	Marco
2	Lucia
3	Giovanni

ID_Ordine	ID_Cliente	Prodotto
1	1	Prodotto A
2	2	Prodotto B

Eseguendo un **LEFT JOIN**:

```
SELECT *

FROM CLIENTI LEFT JOIN ORDINI ON CLIENTI.ID = ORDINI.ID_Cliente;
```

Risultato

ID	Nome	ID_Ordine	ID_Cliente	Prodotto
1	Marco	1	1	Prodotto A
2	Lucia	2	2	Prodotto B
3	Giovanni	NULL	NULL	NULL

In questo caso, Giovanni appare nel risultato con valori NULL per le colonne della tabella **ORDINI**, poiché non ha effettuato alcun ordine.

2. JOIN ESTERNO DESTRO (RIGHT OUTER JOIN)

Il **JOIN ESTERNO DESTRO** restituisce tutte le righe dalla relazione a destra (rd) e le righe corrispondenti dalla relazione a sinistra (rs). Se non ci sono corrispondenze nella relazione a sinistra, i valori delle colonne provenienti da quest'ultima saranno NULL.

```
SELECT *

FROM rs RIGHT OUTER JOIN rd ON condizione;
```

Esempio

Utilizzando le stesse tabelle, eseguendo un **RIGHT JOIN**:

ID	Nome	ID_Ordine	ID_Cliente	Prodotto
1	Marco	1	1	Prodotto A
2	Lucia	2	2	Prodotto B
NULL	NULL	NULL	NULL	NULL

3. JOIN ESTERNO COMPLETO (FULL OUTER JOIN)

Il **JOIN ESTERNO COMPLETO** combina i risultati del **LEFT JOIN** e del **RIGHT JOIN**. Restituisce tutte le righe da entrambe le relazioni, con valori NULL nelle colonne dove non ci sono corrispondenze.

```
SELECT *

FROM rs FULL OUTER JOIN rd ON condizione;
```

ID	Nome	ID_Ordine	ID_Cliente	Prodotto
1	Marco	1	1	Prodotto A
2	Lucia	2	2	Prodotto B
3	Giovanni	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL

Ridenominazione

L'operazione di **ridenominazione** è un concetto fondamentale in teoria delle basi di dati, particolarmente quando si lavora con relazioni in SQL. Essa permette di cambiare i nomi degli attributi di una relazione, mantenendo intatti i dati e la loro struttura.

Definizione di Ridenominazione

Data una relazione **r** definita su uno schema **R(X)**, dove **X** è un insieme di attributi, possiamo definire un nuovo insieme di attributi **Y** avente la stessa cardinalità di **X**.

Condizioni

Cardinalità: L'insieme **Y** deve avere la stessa cardinalità di **X**. Ciò significa che se **X** ha **k** attributi, anche **Y** deve avere **k** attributi.

Dominio: È necessario definire un ordinamento tra gli attributi di **X** e quelli di **Y**. Se indichiamo gli attributi di **X** come $A_1, A_2, \dots, A_k$ e quelli di **Y** come $B_1, B_2, \dots, B_k$, deve valere la proprietà che $dom(A_i) = dom(B_i)$ per ogni i da 1 a k.

Operazione di Ridenominazione

L'operazione di ridenominazione si può formalizzare come segue:

Si parte dalla relazione **r** e si applica una trasformazione per rinominare gli attributi. In notazione formale, se **r** ha attributi $A_1, A_2, \dots, A_k$, e desideriamo rinominarli in $B_1, B_2, \dots, B_k$, l'operazione può essere espressa come:

$$\rho_Y(r) = nuova\ relazione\ con\ attributi\ B_1, B_2, \dots, B_k$$

Esempio di Ridenominazione

Supponiamo di avere la seguente relazione **EMPLOYEES**:

ID	Nome	Mansione
1	Mario	Manager
2	Anna	Developer
3	Luca	Tester

Se desideriamo rinominare gli attributi da **ID, Nome, Mansione** a **Employee_ID, Full_Name, Job_Title**, possiamo applicare l'operazione di ridenominazione:

```
SELECT ID AS Employee_ID, Nome AS Full_Name, Mansione AS Job_Title
FROM EMPLOYEES;
```

Risultato della Ridenominazione

La nuova relazione avrà la seguente forma:

Employee_ID	Full_Name	Job_Title
1	Mario	Manager
2	Anna	Developer
3	Luca	Tester

Alias

Gli alias consentono di creare nomi temporanei per tabelle o colonne, facilitando la leggibilità e la scrittura delle query.

Concetto di Alias

Alias per Tabelle: Permette di fare riferimento a una tabella con un nome diverso, utile in situazioni di auto-join o quando si devono fare confronti all'interno della stessa tabella.

Alias per Colonne: Consente di rinominare le colonne nel risultato della query, migliorando la chiarezza dei risultati.

Esempio di Utilizzo degli Alias

Consideriamo il seguente scenario: vogliamo trovare i nomi e i cognomi dei dipendenti che hanno la stessa mansione di un dipendente specifico (ad esempio, Carlo Rossi), ma escludendo Carlo Rossi stesso dai risultati.

Nome	Cognome	Mansione
Carlo	Rossi	Manager
Anna	Bianchi	Developer
Marco	Verdi	Manager
Luca	Neri	Tester

La query che implementa questo concetto utilizzando alias è la seguente:

```
SELECT T1.Nome, T1.Cognome

FROM ANAGRAFICA_DIPENDENTI T1, ANAGRAFICA_DIPENDENTI T2

WHERE (T1.Mansione = T2.Mansione)

AND (T2.Nome = 'Carlo')

AND (T2.Cognome = 'Rossi')

AND (T1.Nome <> 'Carlo')

AND (T1.Cognome <> 'Rossi');
```

Spiegazione della Query

Alias:

T1: è un alias per la prima istanza della tabella **ANAGRAFICA_DIPENDENTI**. Rappresenta i dipendenti che vogliamo elencare.

T2: è un alias per la seconda istanza della stessa tabella. Rappresenta il dipendente di riferimento (Carlo Rossi) di cui vogliamo trovare i colleghi.

Condizioni:

`(T1.Mansione = T2.Mansione)`: confronta la mansione del dipendente di riferimento (T2) con quella di altri dipendenti (T1).

`(T2.Nome = 'Carlo') AND (T2.Cognome = 'Rossi')`: specifica il dipendente di riferimento.

`(T1.Nome <> 'Carlo') AND (T1.Cognome <> 'Rossi')`: esclude Carlo Rossi dai risultati.

Supponendo che la tabella **ANAGRAFICA_DIPENDENTI** contenga i dati sopra menzionati, il risultato della query sarebbe:

Nome	Cognome
Marco	Verdi

Aggregazione

L'**aggregazione** si riferisce a un insieme di operazioni che combinano valori da più righe in un'unica riga. Queste operazioni sono fondamentali per ottenere informazioni sintetiche dai dati e sono particolarmente utili in contesti di reportistica e analisi dei dati. Alcune delle **funzioni di aggregazione** più comuni sono:

COUNT(): Questa funzione restituisce il numero totale di righe che soddisfano una certa condizione. È utile per determinare quanti elementi esistono in un dataset, come ad esempio il numero totale di vendite.

SUM(): Somma i valori di una colonna numerica, fornendo un totale. Può essere utilizzata, ad esempio, per calcolare il totale delle vendite in un periodo specifico.

AVG(): Calcola la media aritmetica dei valori in una colonna. È utile per avere un'idea del valore medio di una variabile, come il prezzo medio di un prodotto.

MIN() e MAX(): Queste funzioni restituiscono rispettivamente il valore minimo e massimo di una colonna. Possono essere utilizzate, ad esempio, per trovare il prezzo più basso o più alto tra i prodotti in vendita.

Raggruppamento

Il **raggruppamento** consente di aggregare i dati in base ai valori di uno o più attributi. Quando utilizziamo la clausola **GROUP BY**, stiamo dicendo al database di creare dei gruppi di righe che condividono un valore comune in uno o più attributi. Questo è utile quando vogliamo applicare funzioni di aggregazione a ogni gruppo separato.

Esempio di Raggruppamento

Immagina di avere una tabella che registra le vendite di vari prodotti in diversi periodi. Se desideriamo sapere quante vendite sono state fatte per ciascun prodotto, possiamo utilizzare il raggruppamento. Ad esempio, se stiamo analizzando le vendite per ogni prodotto, raggruppando i dati per il nome del prodotto, possiamo sommare le vendite per ottenere il totale per ciascun prodotto.

Clausole di Aggregazione

GROUP BY

La clausola **GROUP BY** è fondamentale per il raggruppamento. Essa permette di specificare gli attributi sui quali vogliamo raggruppare i dati. Tutti gli attributi elencati nella clausola SELECT devono essere presenti anche nella clausola GROUP BY, a meno che non siano utilizzati con una funzione di aggregazione.

HAVING

La clausola **HAVING** è utilizzata per filtrare i risultati dopo che il raggruppamento è stato effettuato. Mentre la clausola **WHERE** viene utilizzata per filtrare righe prima del raggruppamento (sulle singole righe), **HAVING** consente di applicare condizioni sui risultati aggregati. Questo significa che possiamo, ad esempio, filtrare i gruppi che superano un certo valore totale dopo aver eseguito la somma.

Differenze Chiave tra WHERE e HAVING

Caratteristica	WHERE	HAVING
Fase di Esecuzione	Filtra le righe prima del raggruppamento.	Filtra i risultati dopo il raggruppamento.
Utilizzo	Può essere utilizzato con qualsiasi colonna di dati.	Utilizzato per applicare condizioni sui risultati aggregati.
Funzioni di Aggregazione	Non può utilizzare funzioni di aggregazione.	Può utilizzare funzioni di aggregazione.
Scopo	Usato per filtrare righe in base a condizioni specifiche.	Usato per filtrare gruppi di dati risultanti da una query aggregata.
Esempio di Utilizzo	`SELECT * FROM Tabella WHERE colonna1 = valore;`	`SELECT colonna1, COUNT(*) FROM Tabella GROUP BY colonna1 HAVING COUNT(*) > 1;`

Operatori Insiemistici in SQL

In SQL, gli operatori insiemistici consentono di combinare i risultati di due o più query trattandoli come se fossero insiemi matematici. Le principali operazioni insiemistiche sono **UNION**, **INTERSECT** ed **EXCEPT**. Ognuno di questi operatori ha caratteristiche uniche che li rendono utili in diverse situazioni.

UNION

L'operatore **UNION** è utilizzato per unire i risultati di due o più query, producendo un insieme di righe distinte. Questo significa che, se ci sono righe duplicate tra le query, queste verranno eliminate nel risultato finale. Un aspetto importante da considerare quando si utilizza **UNION** è che le colonne delle query devono avere lo stesso numero e tipo di dati compatibili. Ad esempio, se una query seleziona nomi e cognomi, anche l'altra deve avere colonne corrispondenti.

Immagina di avere due tabelle, **STUDENTI_BASIDATI** e **STUDENTI_ANALISI1**, entrambe contenenti dati sugli studenti. Utilizzando **UNION**, puoi ottenere un elenco unico di nomi e cognomi degli studenti presenti in entrambe le tabelle, senza duplicati. Ad esempio:

```
SELECT Nome, Cognome FROM STUDENTI_BASIDATI

UNION
```

```
SELECT Nome, Cognome FROM STUDENTI_ANALISI1;
```

Questo comando restituirebbe un insieme di studenti, assicurandosi che ogni nome e cognome appaiano una sola volta.

INTERSECT

L'operatore **INTERSECT** è utile quando desideri identificare le righe comuni tra due query. Questa operazione restituisce solo le righe che si trovano in entrambe le tabelle, consentendo di vedere le sovrapposizioni tra i due set di dati. Proseguendo con l'esempio delle tabelle degli studenti, se desideri trovare solo gli studenti presenti in entrambe le tabelle, puoi scrivere:

```
SELECT Nome, Cognome FROM STUDENTI_BASIDATI

INTERSECT

SELECT Nome, Cognome FROM STUDENTI_ANALISI1;
```

In questo caso, il risultato includerebbe solo gli studenti che sono elencati in entrambe le tabelle, il che può essere particolarmente utile per analisi comparative o per verificare l'appartenenza a gruppi specifici.

EXCEPT

L'operatore **EXCEPT** serve a trovare le righe presenti nella prima query ma non nella seconda. Questo è utile per isolare le differenze tra i due set di risultati. Tornando all'esempio delle tabelle, se vuoi ottenere un elenco di studenti che si trovano in **STUDENTI_BASIDATI** ma non in **STUDENTI_ANALISI1**, puoi utilizzare:

```
SELECT Nome, Cognome FROM STUDENTI_BASIDATI

EXCEPT

SELECT Nome, Cognome FROM STUDENTI_ANALISI1;
```

In questo modo, otterrai solo gli studenti che non sono inclusi nella seconda tabella, fornendo un insight utile per analisi e reportistica.

Quando si utilizzano gli operatori insiemistici, è importante prestare attenzione alle performance, soprattutto con dataset di grandi dimensioni. Le operazioni insiemistiche possono richiedere tempo, poiché il database deve confrontare i dati per trovare duplicati o sovrapposizioni. È quindi consigliabile testare le singole query per assicurarsi che funzionino come previsto prima di combinarle.

Inoltre, è possibile ordinare i risultati finali utilizzando la clausola **ORDER BY**, ma questa deve essere applicata solo all'ultima query della catena di operazioni. Infine, se si desidera mantenere anche i duplicati, è possibile utilizzare **UNION ALL**, che non rimuove le righe duplicate e può offrire migliori performance.

Query Nidificate

Le query nidificate, note anche come sottointerrogazioni o subquery, sono una caratteristica potente di SQL che consente di utilizzare il risultato di una query all'interno di un'altra. Questa funzionalità permette di eseguire interrogazioni più complesse e dettagliate, facilitando l'analisi e la manipolazione dei dati.

Struttura delle Query Nidificate

In una query nidificata, la sottointerrogazione viene generalmente inserita nella clausola **WHERE** della query principale. La condizione di confronto può essere effettuata utilizzando vari operatori, inclusi quelli di confronto standard e alcuni operatori speciali che estendono le capacità di confronto.

Operatori di Confronto

ALL: Quando si utilizza **ALL**, la condizione specificata deve essere verificata per tutti gli elementi restituiti dalla sottointerrogazione. Ad esempio, se vuoi trovare tutti i dipendenti il cui stipendio è maggiore di quello di tutti i manager, potresti scrivere:

```
SELECT Nome

FROM DIPENDENTI

WHERE Stipendio > ALL (SELECT Stipendio FROM MANAGER);
```

In questo caso, il nome del dipendente verrà restituito solo se il suo stipendio è superiore a quello di tutti i manager.

ANY: Utilizzando **ANY**, la condizione deve essere verificata per almeno uno degli elementi restituiti dalla sottointerrogazione. Ad esempio:

```
SELECT Nome

FROM DIPENDENTI

WHERE Stipendio < ANY (SELECT Stipendio FROM MANAGER);
```

Qui, il nome del dipendente sarà restituito se il suo stipendio è inferiore a quello di almeno un manager.

IN e NOT IN: Questi operatori funzionano in modo simile a **=ANY** e **≠ALL**. Usando **IN**, puoi verificare se un valore è presente nel risultato della sottointerrogazione, mentre **NOT IN** verifica se un valore non è presente.

```
SELECT Nome

FROM DIPENDENTI

WHERE Stipendio IN (SELECT Stipendio FROM MANAGER);
```

Questo esempio restituirà i nomi dei dipendenti il cui stipendio è esattamente uguale a quello di almeno un manager.

EXISTS e NOT EXISTS: Questi operatori sono utilizzati per verificare se una sottointerrogazione restituisce un insieme di risultati non vuoto.

```
SELECT Nome

FROM DIPENDENTI D

WHERE EXISTS (SELECT 1 FROM MANAGER M WHERE D.Stipendio = M.Stipendio);
```

In questo caso, il nome del dipendente verrà restituito se esiste almeno un manager con lo stesso stipendio.

Struttura delle Query Nidificate

Una query nidificata può contenere altre sottointerrogazioni, creando una gerarchia di interrogazioni. Questo consente di costruire interrogazioni molto complesse. Ad esempio, potresti voler trovare tutti i dipendenti il cui stipendio è superiore alla media degli stipendi dei manager:

```
SELECT Nome

FROM DIPENDENTI

WHERE Stipendio > (SELECT AVG(Stipendio) FROM MANAGER);
```

In questo caso, la sottointerrogazione calcola prima la media degli stipendi dei manager e poi confronta il risultato con il salario di ciascun dipendente.

Considerazioni sull'Uso delle Query Nidificate

Le query nidificate sono estremamente utili per l'analisi dei dati, ma possono anche influenzare le performance, soprattutto quando si lavora con dataset molto grandi. È consigliabile testare le sottointerrogazioni separatamente per assicurarsi che funzionino come previsto prima di integrarle in query più complesse.

Inoltre, l'uso eccessivo di sottointerrogazioni può rendere le query più difficili da leggere e mantenere. Pertanto, è importante bilanciare la complessità delle query con la loro chiarezza e comprensibilità.

Esempio 1: Trovare i nomi e i cognomi delle persone che partecipano al progetto con codice 'EUI05'

Questa query cerca i dipendenti che partecipano al progetto con un codice specifico. Si utilizza una sottointerrogazione per cercare le matricole dei dipendenti che partecipano al progetto 'EUI05' e poi, con la query esterna, si recuperano i loro nomi e cognomi.

```
SELECT A.Nome, A.Cognome

FROM ANAGRAFICA A

WHERE A.Matricola IN (
```

```
    SELECT T.Mat
    FROM TEAM T
    WHERE T.Prog = 'EUI05'
);
```

Esempio 2: Trovare i nomi e i cognomi delle persone che partecipano al progetto “Sistemi Informatici e Calcolo Parallelo”

In questo caso, si utilizza una doppia sottointerrogazione: la prima per ottenere il codice del progetto tramite il nome, e la seconda per trovare le matricole dei partecipanti al progetto.

```
SELECT A.Nome, A.Cognome
FROM ANAGRAFICA A
WHERE A.Matricola IN (
    SELECT T.Mat
    FROM TEAM T
    WHERE T.Prog IN (
        SELECT P.Codice
        FROM PROGETTI P
        WHERE P.Nome = 'Sistemi Informatici e Calcolo Parallelo'
    )
);
```

Esempio 3: Trovare i nomi e i cognomi delle persone che partecipano al progetto con codice 'EUI05' (usando EXISTS)

In questo esempio, si utilizza l'operatore EXISTS per verificare l'esistenza di almeno una tupla nella sottointerrogazione.

```
SELECT A.Nome, A.Cognome
FROM ANAGRAFICA A
WHERE EXISTS (
    SELECT *
    FROM TEAM T
    WHERE T.Prog = 'EUI05'
      AND T.Mat = A.Matricola
);
```

Esempio 4: Trovare il nome dei progetti i cui fondi sono maggiori del progetto "IDEA"

In questa query si cerca il nome di tutti i progetti che hanno fondi maggiori rispetto al progetto "IDEA", utilizzando l'operatore ANY.

```
SELECT P1.Codice
FROM PROGETTI P1
WHERE P1.Fondi > ANY (
    SELECT P2.Fondi
    FROM PROGETTI P2
    WHERE P2.Nome = 'IDEA'
```

```
);
```

Esempio 5: Trovare il codice dei progetti con maggiori fondi (usando ALL)

Questa query utilizza l'operatore ALL per trovare il progetto con i fondi più alti confrontandolo con tutti gli altri progetti.

```
SELECT P1.Codice

FROM PROGETTI P1

WHERE P1.Fondi >= ALL (

    SELECT P2.Fondi

    FROM PROGETTI P2

);
```

Le Viste

Le **viste** in SQL rappresentano un concetto potente e flessibile per creare tabelle virtuali che consentono di presentare i dati in modi personalizzati senza duplicare le informazioni fisicamente. Ecco una spiegazione più articolata del concetto:

Che cos'è una vista?

Una vista è una **tabella virtuale** derivata da una o più tabelle esistenti tramite una query SQL. Tuttavia, a differenza delle tabelle tradizionali, le viste non memorizzano i dati in sé; piuttosto, memorizzano una definizione della query che può essere utilizzata per accedere ai dati originali. Questo fa sì che le viste non occupino spazio aggiuntivo nel database, ma agiscano come uno strato di astrazione sopra le tabelle esistenti.

Caratteristiche principali delle viste

Virtualità: Una vista non memorizza fisicamente i dati; le righe e le colonne che mostra sono il risultato di una query eseguita al momento della richiesta. Questo rende le viste efficienti in termini di spazio.

Interfaccia di accesso: Le viste sono un'utile interfaccia per gli utenti o le applicazioni che necessitano di accedere frequentemente a determinati set di dati, ma che potrebbero non aver bisogno di vedere l'intera tabella. Ad esempio, una vista può essere creata per mostrare solo determinati attributi di una tabella o solo le righe che soddisfano determinate condizioni.

Limitazioni sugli aggiornamenti: Una conseguenza del fatto che le viste sono tabelle virtuali è che non sempre possono essere aggiornate direttamente. In altre parole, esistono delle restrizioni nel modificare i dati attraverso una vista, soprattutto se la query che la definisce è complessa (ad esempio, se coinvolge più tabelle o funzioni aggregate).

Esempio di sintassi per la creazione di una vista

La creazione di una vista in SQL avviene tramite la seguente sintassi:

```
CREATE VIEW nome_vista AS

SELECT ...

FROM ...

WHERE ...
```

Esempio pratico

Supponiamo di avere una tabella ANAGRAFICA che contiene informazioni sui dipendenti, e di voler creare una vista per mostrare solo i dipendenti di un particolare dipartimento, in questo caso il dipartimento "DIS". La sintassi sarà la seguente:

```
CREATE VIEW IMPIEGATI_DIS AS

SELECT *

FROM ANAGRAFICA
```

```
WHERE NomeDip = 'DIS';
```

In questo esempio:

`IMPIEGATI_DIS` è il nome della vista.

La query seleziona tutte le righe dalla tabella ANAGRAFICA dove il campo NomeDip (che indica il dipartimento) è uguale a 'DIS'.

Operazioni set-oriented

Le **operazioni set-oriented** in SQL permettono di applicare comandi di inserimento, aggiornamento o cancellazione a **insiemi di tuple** piuttosto che a singole righe, rendendo l'interazione con la base di dati più efficiente. Queste operazioni si basano su comandi che operano su interi set di dati, grazie alla potenza delle query SELECT che identificano le tuple su cui intervenire. Vediamo nel dettaglio come funzionano.

Inserimento set-oriented (INSERT)

L'operazione INSERT può essere estesa per aggiungere **insiemi di dati** in una tabella, utilizzando il risultato di una query SELECT. Invece di inserire i dati riga per riga, è possibile inserire in blocco le righe che soddisfano determinate condizioni.

Esempio:

```
INSERT INTO IMPIEGATI_DIS (

SELECT *

FROM ANAGRAFICA

WHERE NomeDip = 'DIS');
```

Questo comando copia tutte le righe della tabella ANAGRAFICA in cui il campo NomeDip è uguale a 'DIS', inserendole nella tabella IMPIEGATI_DIS.

Si sfrutta la potenza del SELECT per specificare il set di dati che devono essere inseriti.

Cancellazione set-oriented (DELETE)

Similmente, l'operazione di cancellazione può essere applicata a **più righe contemporaneamente**, utilizzando condizioni che determinano quali tuple devono essere eliminate.

Esempio:

```
DELETE FROM ANAGRAFICA

WHERE NomeDip = 'DIS';
```

Questo comando cancella tutte le righe della tabella ANAGRAFICA in cui il campo NomeDip ha il valore 'DIS'.

In questo modo, si eliminano tutte le informazioni relative ai dipendenti del dipartimento 'DIS' in un'unica operazione.

Aggiornamento set-oriented (UPDATE)

L'operazione UPDATE può essere utilizzata per modificare **più tuple** contemporaneamente. Anche in questo caso, si utilizza una condizione che determina quali righe devono essere aggiornate.

Esempio:

```
UPDATE ANAGRAFICA

SET Cognome = ''

WHERE NomeDip = 'DIS';
```

Questo comando aggiorna il campo Cognome a un valore vuoto per tutte le righe della tabella ANAGRAFICA dove il campo NomeDip è 'DIS'.

L'operazione UPDATE viene eseguita su tutte le righe che soddisfano la condizione specificata.

Normalizzazione e Forme Normali

La **normalizzazione** è il processo di riorganizzazione delle relazioni in una base di dati per ridurre la **ridondanza** e migliorare l'integrità dei dati. L'obiettivo principale della normalizzazione è quello di eliminare anomalie di aggiornamento, cancellazione e inserimento, ottenendo schemi di relazione che soddisfino determinati requisiti, chiamati **forme normali**.

Una **forma normale** rappresenta un insieme di condizioni che un determinato schema deve rispettare per evitare problemi strutturali. Il raggiungimento di una forma normale migliora la qualità dello schema stesso, riducendo duplicazioni e migliorando la coerenza dei dati.

Problemi negli Schemi Non Normalizzati

Uno schema non normalizzato può presentare **ridondanza** di informazioni, cioè la ripetizione di dati senza aggiungere nuove informazioni utili. Questo genera **anomalie** quando si cerca di aggiornare, eliminare o inserire dati, perché ogni modifica può comportare l'intervento su più tuple, riducendo l'efficienza e aumentando la possibilità di errore.

Esempio di Schema Non Normalizzato

Consideriamo la relazione **Progettazioni (Impiegato, Stipendio, Progetto, Bilancio, Funzione)**, dove:

Ogni **Impiegato** ha un solo **Stipendio**, indipendentemente dai progetti a cui partecipa.

Ogni **Progetto** ha un **Bilancio** unico.

Se un impiegato partecipa a tre progetti, lo stipendio sarà ripetuto tre volte, causando ridondanza. Inoltre, se lo stipendio dell'impiegato cambia, dovrà essere aggiornato in tutte le tuple in cui compare, generando **anomalia di aggiornamento**.

Le Anomalie

Le anomalie sono problemi derivanti dall'utilizzo di un'unica relazione per rappresentare informazioni eterogenee. Quando concetti diversi vengono fusi in una singola relazione, si creano problemi di ridondanza e anomalie che rendono difficoltose le operazioni di manipolazione dei dati.

Le principali **anomalie** sono:

- **Anomalia di aggiornamento**: ogni volta che si modifica un valore ripetuto (ad esempio, lo stipendio), bisogna aggiornare tutte le tuple che lo contengono.
- **Anomalia di cancellazione**: eliminando un progetto da cui dipendono altre informazioni (ad esempio, l'impiegato), si rischia di perdere anche dati ancora validi (ad esempio, lo stipendio dell'impiegato).
- **Anomalia di inserimento**: non è possibile inserire un impiegato che non partecipa a nessun progetto, perché mancherebbe una parte delle informazioni necessarie per completare la tupla.

Esempio di Anomalia di Cancellazione

Supponiamo che un impiegato partecipi a un solo progetto. Se quel progetto viene eliminato, anche tutte le informazioni sull'impiegato verranno cancellate, nonostante potrebbe essere ancora un dipendente dell'azienda.

Dipendenze Funzionali (DF)

Per eliminare le anomalie, è necessario identificare le relazioni logiche tra gli attributi di una relazione, note come **dipendenze funzionali (DF)**. Una dipendenza funzionale è una relazione tra due insiemi di attributi di una relazione, in cui un attributo (o un insieme di attributi) determina univocamente un altro attributo.

Se Y e Z sono due sottoinsiemi di attributi di una relazione, una dipendenza funzionale $Y \rightarrow Z$ indica che il valore di Z è determinato univocamente dal valore di Y. Formalmente: $Y \rightarrow Z$ si legge: "Y determina Z" o "Z dipende funzionalmente da Y".

Definizione Formale di Dipendenza Funzionale

Consideriamo:

Una relazione r su uno schema di relazione R(X) e due sottoinsiemi Y e Z di X.

Esiste una dipendenza funzionale da Y a Z, indicata con Y→Z, se, per ogni coppia di tuple t_1 e t_2 in r, che hanno lo stesso valore in Y, risulta che t_1 e t_2 hanno anche lo stesso valore in Z: $\forall t_1, t_2 \in r: t_1[Y] = t_2[Y] \Longrightarrow t_1[Z] = t_2[Z]$
Questa condizione stabilisce che se due tuple concordano nei valori di Y, devono concordare anche nei valori di Z.

Esempi di Dipendenze Funzionali

Ogni impiegato ha un solo stipendio:

$$Impiegato \rightarrow Stipendio$$

Anche se l'impiegato partecipa a più progetti, il suo stipendio rimane unico.

Ogni progetto ha un bilancio:

$$Progetto \rightarrow Bilancio$$

Il bilancio di un progetto dipende solo dal progetto e non dagli impiegati.

Ogni impiegato ha una funzione specifica in ogni progetto:

$$Impiegato, Progetto \rightarrow Funzione$$

Anche se un impiegato può svolgere più ruoli in progetti diversi, in un progetto specifico avrà una sola funzione.

Dipendenze Funzionali Banali e Non Banali

Le **dipendenze funzionali banali** sono quelle che si verificano per definizione e non forniscono nuove informazioni utili sulla struttura della relazione. Per esempio, Impiegato,Progetto→Progetto è una dipendenza banale, poiché "Progetto" è già parte dell'insieme a sinistra dell'implicazione.

Una dipendenza funzionale Y→Z è **non banale** se nessuno degli attributi in Z è già presente in Y. Ad esempio:

$$Impiegato \rightarrow Stipendio$$

è una dipendenza non banale perché lo stipendio dipende interamente dall'impiegato e non è parte dell'insieme Y.

Chiavi e Attributi

Una **chiave** in uno schema di relazione è un insieme di attributi che identifica univocamente una tupla. Gli attributi che partecipano a una chiave si chiamano **attributi primi**, mentre quelli che non partecipano a nessuna chiave sono detti **attributi non primi**.

Se K è una chiave per una relazione R(X), allora ogni attributo non primo dipende funzionalmente da K. Questo significa che possiamo esprimere una dipendenza funzionale completa come:

$$K \rightarrow Z$$

dove Z è un insieme di attributi non primi e $X \supseteq K \cup Z$.

Esempio di Chiave e Attributi

In una relazione **Impiegati (Impiegato, Stipendio)**, la chiave è Impiegato, e quindi ogni altro attributo della relazione, come Stipendio, dipende da esso:

$$Impiegato \rightarrow Stipendio$$

Dipendenza Funzionale Completa

Una dipendenza funzionale Y→Z è detta **completa** se nessun sottoinsieme proprio di Y determina Z. In altre parole, l'intero insieme di attributi Y è necessario per determinare univocamente Z.

Esempio di Dipendenza Funzionale Completa

Nella relazione **Partecipazioni (Impiegato, Progetto, Funzione)**, l'attributo Funzione dipende sia da Impiegato che da Progetto. Quindi, la dipendenza Impiegato,Progetto→Funzione è completa. Se considerassimo solo uno di questi attributi, non saremmo in grado di determinare la funzione di un impiegato senza conoscere anche il progetto specifico.

Forme Normali

Le forme normali rappresentano un insieme di regole o criteri progettuali, mirati a ridurre la ridondanza e prevenire le anomalie nei database relazionali. Tali regole definiscono come organizzare i dati in modo efficiente e consistente all'interno delle relazioni, o tabelle, di un database, e guidano il processo di normalizzazione, che è il processo attraverso cui si trasforma uno schema di dati in modo da soddisfare determinate condizioni per evitare problemi.

Prima forma normale (1NF)

La **Prima Forma Normale (1NF)** rappresenta la base del modello relazionale, *che richiede che ogni attributo di una relazione contenga un valore atomico*, ossia indivisibile. In altre parole, **non sono ammessi attributi composti** o multivalore all'interno delle tabelle. Ciò significa che una tabella deve essere strutturata in modo tale che ogni cella contenga un solo dato, e ogni riga abbia un valore unico per un determinato attributo.

L'applicazione della 1NF ha l'obiettivo di rendere le tabelle più facili da gestire ed elaborare. Se una tabella contiene colonne con insiemi di valori, può risultare difficile da gestire nelle query, e potrebbero emergere problemi come la ridondanza o le difficoltà nella manipolazione dei dati.

Seconda forma normale (2NF)

Una relazione è in **Seconda Forma Normale (2NF)** se soddisfa due condizioni: è in 1NF e ogni attributo non primo dipende completamente da tutta la chiave primaria. In altre parole, gli attributi non chiave non devono dipendere da una parte sola della chiave primaria (dipendenze parziali), ma devono essere totalmente determinati dalla chiave.

Per comprendere meglio, supponiamo di avere una relazione che descrive gli impiegati e i progetti a cui lavorano, dove la chiave primaria è composta da "Impiegato" e "Progetto". Se esiste un attributo come "Stipendio", che dipende solo dall'impiegato e non dal progetto, si creerebbe una dipendenza parziale. Questa condizione violerebbe la 2NF perché lo stipendio dipende solo da una parte della chiave e non dalla combinazione "Impiegato-Progetto". La decomposizione di tale relazione è una delle soluzioni per riportare la tabella in 2NF, separando le informazioni dipendenti esclusivamente dall'impiegato in una nuova tabella.

Decomposizione e decomposizione senza perdita

Quando una relazione non soddisfa la 2NF o una forma normale più elevata, è possibile decomporre la relazione in più tabelle che soddisfano i requisiti della normalizzazione. Tuttavia, è fondamentale che la decomposizione sia eseguita senza perdita di informazioni, garantendo che il join naturale tra le nuove tabelle ricostruisca la relazione originale.

La decomposizione senza perdita è ottenuta quando il join naturale delle proiezioni della relazione originale sulle nuove tabelle restituisce esattamente l'istanza originale. Per garantire questo, è necessario che gli attributi comuni tra le tabelle risultanti contengano una superchiave di almeno una delle tabelle. In altre parole, gli attributi condivisi tra le tabelle decomposte devono permettere di ricostruire le tuple originarie senza creare dati spuri.

Conservazione delle dipendenze

Un altro aspetto cruciale nella decomposizione è la **conservazione delle dipendenze funzionali**, ossia la capacità di preservare i vincoli esistenti tra gli attributi nel processo di decomposizione. Se una decomposizione separa gli attributi di una dipendenza funzionale, può diventare difficile verificare il soddisfacimento di tale dipendenza senza dover ricostruire più tabelle. Ciò può compromettere l'integrità dei dati e rendere più complessa la gestione del database.

Terza forma normale (3NF)

Una relazione è in **Terza Forma Normale (3NF)** se è in 2NF e non contiene dipendenze transitive tra attributi non primi e la chiave primaria. Una dipendenza transitiva si verifica quando un attributo non primo è determinato da un altro attributo non primo, che a sua volta dipende dalla chiave. Per esempio, se abbiamo una relazione con gli attributi "Impiegato", "Progetto" e "Sede", e se "Sede" dipende da "Progetto", e "Progetto" dipende da "Impiegato", allora "Sede" dipende transitivamente da "Impiegato". Questa condizione viola la 3NF, poiché l'attributo "Sede" non dovrebbe dipendere da un attributo che non è parte della chiave primaria.

La 3NF si concentra dunque sull'eliminazione delle dipendenze transitive per evitare anomalie come ridondanze o difficoltà negli aggiornamenti. Per raggiungere questa forma normale, è spesso necessario eseguire ulteriori decomposizioni delle relazioni.

Forma normale di Boyce-Codd (BCNF)

La **Forma Normale di Boyce-Codd (BCNF)** è una versione più rigorosa della 3NF. Una relazione è in BCNF se per ogni dipendenza funzionale non banale, l'insieme di attributi che determina gli altri attributi (determinante) deve essere una superchiave della relazione. In altre parole, una relazione non può avere dipendenze funzionali in cui l'insieme di attributi determinante non sia una superchiave.

Sebbene una relazione in 3NF soddisfi la maggior parte dei requisiti per evitare anomalie, esistono casi in cui alcune dipendenze funzionali violano comunque il principio della superchiave, che la BCNF risolve. Tuttavia, passare alla BCNF non garantisce sempre la conservazione delle dipendenze funzionali, quindi il processo deve essere bilanciato attentamente.

Comportamento attivo delle basi di dati e trigger

Tradizionalmente, i DBMS erano considerati passivi: eseguivano solo le istruzioni transazionali inviate dagli utenti, come inserimenti, cancellazioni o aggiornamenti. Tuttavia, il comportamento attivo, introdotto con i trigger, consente ai DBMS di reagire autonomamente a determinati eventi. Questa caratteristica si basa sul paradigma E-C-A (evento-condizione-azione), che permette alla base di dati di rispondere automaticamente a una situazione specifica quando si verifica un evento particolare.

I trigger, dunque, non sono altro che regole definite che si attivano a fronte di eventi transazionali predefiniti. Ogni trigger segue il paradigma E-C-A, suddiviso in tre componenti fondamentali:

- **Evento**: rappresenta il momento di attivazione del trigger, ossia una modifica dello stato dei dati, come un'operazione di INSERT, DELETE o UPDATE.
- **Condizione**: una condizione logica che specifica se il trigger debba effettivamente essere eseguito quando si verifica l'evento.
- **Azione**: rappresenta una sequenza di comandi SQL o una procedura memorizzata che viene eseguita se la condizione è soddisfatta.

Grazie ai trigger, una parte delle logiche applicative diventa "condivisa" all'interno della base di dati, garantendo l'indipendenza della conoscenza: le regole e la logica reattiva vengono estratte dal codice dell'applicazione e centralizzate nella gestione della base di dati stessa. Ciò rende la manutenzione delle regole aziendali più agevole e, allo stesso tempo, aumenta la consistenza dei dati, poiché tutti i sistemi che accedono alla base di dati condividono lo stesso insieme di regole.

Eventuali **condizioni** e **azioni**, cioè le istruzioni che il trigger deve eseguire se la condizione è verificata.

Modo di esecuzione dei trigger: BEFORE e AFTER

I trigger possono essere configurati per essere eseguiti prima (BEFORE) o dopo (AFTER) l'operazione specificata:

- **BEFORE**: esegue il trigger prima che l'evento venga effettivamente applicato sulla base di dati. Questa modalità è utile quando è necessario controllare o validare le modifiche prima che vengano registrate, garantendo integrità e consistenza dei dati in fase di modifica.
- **AFTER**: esegue il trigger subito dopo che l'operazione è stata completata. Questo è il comportamento più comune e viene utilizzato in moltissime applicazioni, come la gestione dei log o l'aggiornamento di record collegati in altre tabelle.

Granularità dei trigger: statement-level e row-level

La granularità di un trigger definisce se l'attivazione debba avvenire a livello di singola istruzione (statement-level) o di singola riga (row-level):

- **Statement-level**: il trigger viene eseguito una sola volta per ciascuna istruzione SQL, indipendentemente dal numero di righe coinvolte. Questo approccio è coerente con l'approccio set-oriented dei comandi SQL, che si applicano a insiemi di dati.
- **Row-level**: il trigger viene eseguito una volta per ogni riga modificata dall'istruzione SQL. È utile per operazioni che richiedono la verifica o la modifica di singole righe, ma può risultare meno efficiente se l'operazione coinvolge un elevato numero di tuple.

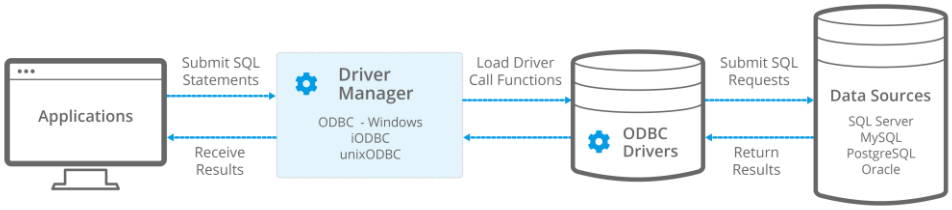
Clausola REFERENCING e variabili di transizione

Per i trigger row-level, si possono utilizzare le variabili di transizione OLD e NEW, che rappresentano i valori della riga prima e dopo la modifica. Per i trigger statement-level, sono invece disponibili OLD TABLE e NEW TABLE, ossia due tabelle di transizione che contengono le tuple modificate prima e dopo l'operazione:

- **OLD** e **NEW** sono utilizzabili solo nei trigger row-level e permettono di riferirsi ai valori di una singola tupla.
- **OLD TABLE** e **NEW TABLE** sono disponibili per i trigger statement-level e consentono di accedere a tutte le tuple coinvolte nell'operazione.

ODBC: Interoperabilità e Accesso ai Dati Multi-Piattaforma

L'ODBC (Open Database Connectivity) è una tecnologia progettata per risolvere un problema molto concreto: permettere a un'applicazione di interagire con database diversi senza dover scrivere codice specifico per ciascun sistema. Prima della sua introduzione, infatti, chiunque sviluppasse software capace di accedere ai dati si trovava a riscrivere parte del codice ogni volta che cambiava il database sottostante. Con ODBC, Microsoft ha creato uno standard comune, una sorta di "ponte" per l'accesso ai dati.

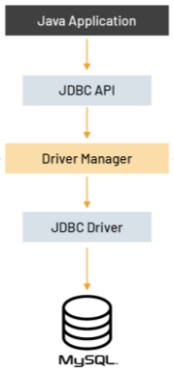


La struttura di ODBC è ideata per mantenere una separazione tra applicazione e database, utilizzando una serie di driver specifici per ciascun sistema di gestione dei dati (come MySQL, Oracle, PostgreSQL, ecc.). Questi driver fanno da **traduttori**, traducendo le richieste dell'applicazione in comandi che il database può comprendere. Grazie a questa architettura, lo sviluppatore può scrivere codice una volta sola e, cambiando solo il driver, far funzionare la propria applicazione con database diversi, senza dover apportare modifiche significative.

Un'altra caratteristica interessante di ODBC è la **possibilità di accedere ai dati da diversi linguaggi di programmazione**. Non essendo legato a un linguaggio specifico, ODBC supporta linguaggi come C, C++, Python, e molti altri, rendendolo una scelta ideale per le aziende che utilizzano un mix di tecnologie. Tuttavia, questa versatilità ha un **costo in termini di prestazioni**: poiché la connessione passa attraverso vari livelli di traduzione, le applicazioni ODBC possono risultare meno veloci rispetto a quelle che utilizzano protocolli nativi, soprattutto in contesti ad alte prestazioni.

Inoltre, ogni database richiede un driver specifico, il che può complicare la gestione dei progetti. Infatti, l'ODBC funziona tramite un sistema di DSN (Data Source Name), che contiene tutte le informazioni di connessione necessarie. Il DSN deve essere configurato per ogni database, introducendo un ulteriore livello di complessità nella gestione delle connessioni, soprattutto quando si ha a che fare con diversi database in un unico sistema.

JDBC: Un'interfaccia Nativa e Ottimizzata per Java



JDBC (Java Database Connectivity) è un termine chiave in ambito informatico, specialmente per lo sviluppo di applicazioni in Java che richiedono accesso a database. Si tratta di una libreria Java fondamentale, che si articola in varie API progettate per facilitare la connessione e l'interazione con una molteplicità di database, indipendentemente dal tipo di sistema di gestione del database (DBMS) impiegato. JDBC consente a un'applicazione Java di inviare istruzioni SQL, di ricevere e interpretare i risultati, e di gestire la comunicazione con il database tramite un'interfaccia unificata.

La tecnologia JDBC è composta da diverse implementazioni che variano a seconda delle necessità di accesso e delle prestazioni richieste. Una delle implementazioni più diffuse, soprattutto negli ambienti Windows, è la "bridge JDBC – ODBC", una soluzione che rappresenta un "ponte" tra l'applicazione Java e il database tramite il driver ODBC (Open Database Connectivity). Questa bridge consente l'accesso a un ampio numero di database purché il driver ODBC sia disponibile e

configurato correttamente sul server. In pratica, il driver JDBC – ODBC permette al codice Java di comunicare con il driver ODBC, che a sua volta si interfaccia con il database desiderato. Sebbene sia diffusa, questa soluzione potrebbe risultare meno performante rispetto ad altre implementazioni di JDBC, data la doppia traduzione delle istruzioni.

Funzionamento del Driver Manager e delle API JDBC

L'applicazione Java utilizza l'API JDBC per dialogare con il **JDBC Driver Manager**, un elemento centrale che coordina la gestione delle connessioni e la distribuzione delle richieste SQL. Quando un'applicazione invia una query, il Driver Manager sceglie il driver appropriato e invia la richiesta al DBMS tramite un'API specifica del driver. Questa API, nota come **JDBC Driver API**, si occupa di convertire i comandi e di comunicare con il DBMS secondo il protocollo necessario. Questo approccio rende l'accesso ai dati più standardizzato e indipendente, semplificando notevolmente lo sviluppo.

Nel contesto Windows, il driver più utilizzato per gestire la connessione a database attraverso i driver ODBC è proprio il bridge JDBC – ODBC, che permette a un'applicazione Java di interfacciarsi con i driver ODBC disponibili, ampliando notevolmente la compatibilità con diversi DBMS. Tuttavia, è utile notare che, nonostante sia una soluzione versatile, il bridge JDBC – ODBC è meno efficiente rispetto a un driver puro Java, in quanto introduce un ulteriore livello di traduzione e dipende da un componente esterno.

Componenti Principali dell'Architettura JDBC

L'architettura JDBC si struttura attorno a due componenti fondamentali: il **Driver Manager** e i **driver specifici JDBC**. Questi ultimi sono definiti in base al DBMS specifico con cui l'applicazione deve interfacciarsi. Ad esempio, esistono driver JDBC progettati appositamente per MySQL, altri per Oracle e così via. Il ruolo del Driver Manager è essenziale poiché rappresenta il "layer di astrazione", che consente alle applicazioni Java di comunicare con i database mediante un set di API standardizzato (JDBC API). In pratica, il Driver Manager funge da regista, coordinando le connessioni e caricando i driver necessari in base al database utilizzato. In questo modo, un'applicazione può essere implementata con un approccio modulare, in cui la logica del codice rimane indipendente dal database specifico.

Tecnologia dei DBMS

Un sistema di gestione di basi di dati (DBMS) include una serie di moduli specializzati che operano insieme per garantire l'efficienza, l'affidabilità e la sicurezza delle operazioni sui dati. La struttura di un DBMS è complessa e orientata a ottimizzare ogni aspetto della gestione dei dati, dalla semplice memorizzazione fino alla gestione della concorrenza e del recupero in caso di guasti. Vediamo in dettaglio come funziona ciascun modulo e come questi contribuiscono al corretto funzionamento del sistema.

1. **Gestione delle Query**

   Ogni query viene interpretata e processata da un modulo chiamato **Gestore delle Query**. Questo modulo analizza la query e, se necessario, la "ottimizza" selezionando il piano di esecuzione più efficiente per portarla a termine. L'ottimizzazione consiste nella scelta delle operazioni di basso livello da eseguire (come scansioni di tabelle, accessi diretti, ordinamenti e join) in base al costo computazionale e di accesso ai dati.

   L'analisi di una query richiede quindi un piano dettagliato in termini di **operatori di basso livello**. Per tradurre una query in questi operatori, il DBMS utilizza metodi specifici di accesso ai dati, con il supporto di strutture come indici e tabelle di gestione dei file.

2. **Gestore dei Metodi di Accesso e dei File**

   Per garantire l'accesso ai dati in modo efficiente, il **Gestore dei Metodi di Accesso e dei File** coordina le operazioni che riguardano l'accesso e la gestione delle informazioni sui file della base di dati. Da un punto di vista logico, ogni file può essere visto come una sequenza di record o una collezione di **pagine di record**.

3. **Gestore del Buffer di Memoria**

   Il **Gestore del Buffer di Memoria** determina quando e come trasferire le pagine di record tra la memoria centrale e la memoria di massa. Questo meccanismo è essenziale per garantire che solo le informazioni necessarie siano mantenute in memoria, riducendo al minimo i tempi di accesso a disco e ottimizzando l'uso della memoria centrale.

4. **Gestore dello Spazio su Disco**

Le operazioni di lettura, scrittura, allocazione e rilasciamento delle pagine su disco sono controllate dal **Gestore dello Spazio su Disco**. Questo modulo fornisce un controllo accurato sull'uso del disco, assicurando che lo spazio sia utilizzato in modo efficiente e che le pagine vengano allocate o liberate correttamente.

5. **Gestore della Concorrenza**

   In un DBMS, molteplici utenti possono eseguire transazioni contemporaneamente. Il **Gestore della Concorrenza** è responsabile di assicurare che queste operazioni avvengano in modo corretto e coerente, mantenendo le proprietà ACID (Atomicità, Consistenza, Isolamento, Durabilità). Questo modulo garantisce che le transazioni non interferiscano tra loro, evitando conflitti e inconsistenze nei dati.

6. **Gestore dell'Affidabilità**

   Il **Gestore dell'Affidabilità** interviene in caso di guasti del sistema, ad esempio a causa di interruzioni di corrente o malfunzionamenti hardware. Questo modulo registra in tempo reale tutte le modifiche effettuate ai dati (attraverso un **log** delle transazioni) per poter ripristinare lo stato del sistema in caso di errore. In caso di un guasto, il log permette di ricostruire la base di dati fino a un punto di stato consistente.

7. **Gestore dell'Integrità**

   La consistenza dei dati è assicurata dal **Gestore dell'Integrità**, che controlla il rispetto dei vincoli di integrità definiti all'interno del database. Questo modulo entra in azione ogni volta che vengono eseguite operazioni di modifica sui dati per garantire che nessun vincolo venga violato, mantenendo la qualità e la coerenza delle informazioni memorizzate.

8. **Gestore degli Accessi**

   Il **Gestore degli Accessi** si occupa della sicurezza dei dati, assicurando che solo utenti e applicazioni autorizzati possano accedere alle informazioni della base di dati. Questo modulo controlla i privilegi degli utenti e verifica che le operazioni effettuate siano compatibili con i livelli di accesso assegnati, prevenendo accessi non autorizzati.

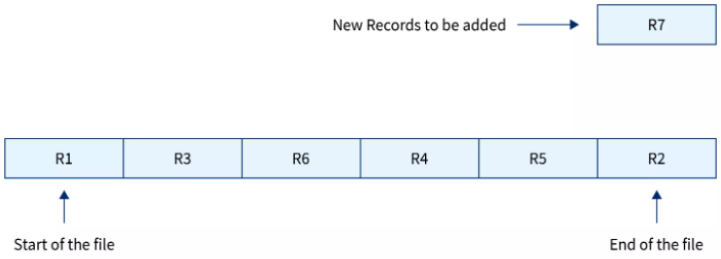
Memorizzazione dei Dati

Per gestire in modo persistente le informazioni di una base di dati, il DBMS memorizza i dati su **dispositivi di memoria di massa**, come dischi o, meno comunemente, nastri. La memoria di massa è fondamentale per mantenere i dati anche quando il sistema è spento, permettendo un accesso rapido alle informazioni necessarie.

Quando una query richiede l'elaborazione di un'informazione, i dati vengono trasferiti dalla memoria di massa alla memoria centrale per poter essere utilizzati nel calcolo. Dopo l'elaborazione, i risultati possono essere riscritti su disco per garantire la persistenza delle modifiche.

File di Record

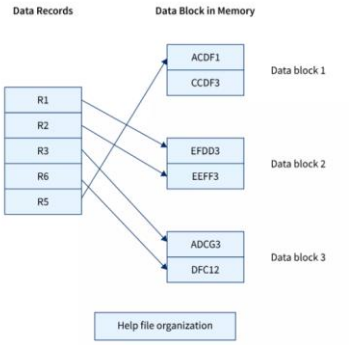
La struttura dati tipica per la memorizzazione delle informazioni in un database è il **file di record**. Ogni file è formato da una sequenza di record, dove **ogni record rappresenta una singola unità di informazione** composta da uno o più campi. Questi campi possono includere vari tipi di dati e ognuno di essi è identificato da un identificatore univoco che permette di individuare rapidamente l'indirizzo fisico della pagina su cui risiede.



Organizzazione dei File in un DBMS

Nel contesto di un sistema di gestione di basi di dati (DBMS), l'organizzazione dei file è fondamentale per ottimizzare le operazioni di memorizzazione e recupero dei dati. I file possono essere organizzati in vari modi a seconda delle esigenze specifiche del sistema e delle operazioni che devono essere supportate. Le principali tipologie di file utilizzate in un DBMS sono i **file non ordinati (heap)**, i **file ordinati (sequenziali)** e i **file ad accesso calcolato (hash)**. Ciascuna di queste strutture presenta vantaggi e svantaggi, che influenzano il modo in cui i dati vengono memorizzati, recuperati e aggiornati.

File HEAP



I file heap sono una delle strutture più semplici e vengono utilizzati per memorizzare i record senza un ordine specifico. In un file heap, i record vengono inseriti in maniera casuale. Ogni nuovo record viene posizionato alla fine del file, e se lo spazio disponibile in una pagina non è sufficiente, viene creata una nuova pagina e aggiunta alla fine del file. Questa struttura consente operazioni di inserimento molto rapide, ma l'accesso ai dati non è altrettanto efficiente.

Poiché i record non sono ordinati, l'unico metodo per recuperare i dati è la ricerca lineare, che richiede la scansione dell'intero file. Per quanto riguarda la cancellazione, quando un record viene eliminato, esso viene semplicemente "marcato" come cancellato, ma lo spazio che occupava non viene effettivamente liberato.

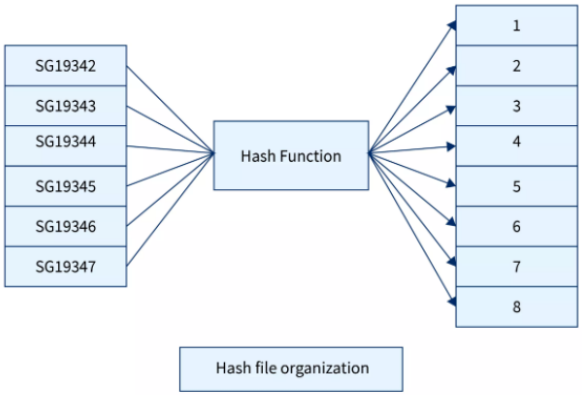
Questo porta, con il tempo, a un deterioramento delle prestazioni del file, soprattutto in caso di frequenti cancellazioni, poiché si generano aree vuote che necessitano di essere recuperate tramite una riorganizzazione del file.

File ORDINATI

I file ordinati, al contrario, memorizzano i record in ordine crescente o decrescente, solitamente su uno o più campi di ciascun record, come ad esempio una chiave primaria. L'ordinamento consente operazioni di ricerca più efficienti, in particolare mediante l'uso della ricerca binaria, che riduce significativamente il numero di operazioni necessarie per trovare un record. Tuttavia, le operazioni di inserimento e cancellazione diventano più complesse: ogni volta che si inserisce un nuovo record, è necessario trovare la posizione giusta in cui inserirlo e, se non c'è spazio sufficiente in una pagina, si crea una nuova pagina o si spostano i record esistenti. La cancellazione richiede anch'essa una riorganizzazione del file per rimuovere lo spazio libero lasciato dai record eliminati. Inoltre, per mantenere il file ordinato, è necessario eseguire periodiche riorganizzazioni, come il **merge-sort**, per evitare che il file si degradi nel tempo. Nonostante la complessità delle operazioni di aggiornamento, i file ordinati sono molto efficienti per operazioni di ricerca, poiché l'ordinamento permette di ridurre drasticamente il numero di confronti necessari per recuperare un record.

File HASH

I file hash utilizzano una funzione matematica, chiamata **funzione di hash**, per determinare la posizione in cui un record deve essere memorizzato nel file. La funzione di hash prende uno o più campi del record (ad esempio una chiave) e restituisce un valore che rappresenta l'indirizzo fisico del record nel file.



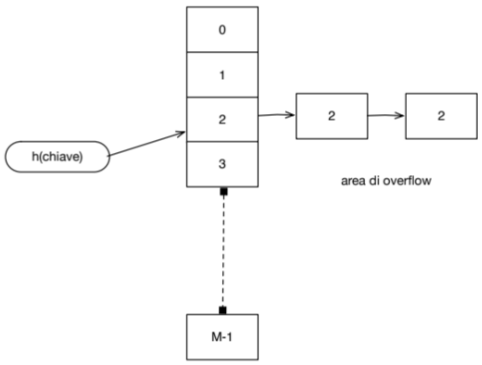
Questo approccio consente un accesso molto rapido ai dati, poiché la posizione del record è determinata direttamente senza bisogno di eseguire ricerche lineari o binarie. Tuttavia, la funzione di hash non può garantire che ogni record abbia una posizione unica, poiché il numero di valori di hash possibili può essere maggiore rispetto al numero di posizioni disponibili nel file.

Questo fenomeno, chiamato **collisione**, si verifica quando due o più record hanno lo stesso valore di hash e quindi finiscono nella stessa posizione del file. Per gestire le collisioni, vengono utilizzati **bucket**, che sono zone di memoria in cui vengono memorizzati più record con lo stesso valore di hash. Se un bucket è pieno, i record successivi vengono inseriti in una **zona di overflow**, che può essere gestita in modo libero o tramite una lista concatenata. Nonostante le

collisioni, i file hash offrono un accesso molto veloce ai dati, ma è essenziale che la funzione di hash sia ben progettata per evitare una distribuzione inefficiente dei record.

Collisioni nei File di HASH

Quando si utilizza una **funzione di hash** per determinare la posizione di un record in un file, l'idea è quella di calcolare un valore numerico (l'hash) che corrisponde a un indirizzo specifico del record nel file. Tuttavia, non è possibile garantire che ogni record abbia un indirizzo univoco, poiché il numero di possibili valori che una funzione di hash può generare è molto più ampio rispetto al numero di posizioni disponibili nel file.



Questo porta inevitabilmente alla possibilità di **collisioni**, ossia situazioni in cui due o più record hanno lo stesso valore di hash e quindi sono indirizzati alla stessa posizione nel file.

Cos'è una Collisione?

Una collisione si verifica quando due o più record, anche se distinti, vengono mappati dalla funzione di hash sulla stessa posizione del file. Poiché ogni posizione nel file è associata a un unico **bucket** (una zona di memoria che contiene uno o più record), più record che hanno lo stesso valore di hash finiscono nello stesso bucket.

Gestione delle Collisioni

Quando si verifica una collisione e il bucket in questione è già pieno, è necessario adottare una strategia per gestire l'inserimento dei record. Esistono due principali tecniche di gestione delle collisioni:

1. **Area di Overflow**: Quando un bucket non è in grado di contenere ulteriori record, si può creare un'area di overflow. Questo è uno spazio aggiuntivo in cui i record che causano collisioni vengono inseriti. Ci sono due modalità comuni per l'uso dell'area di overflow:
    - **Inserimento libero**: I record vengono posizionati nell'area di overflow in modo sequenziale, seguendo l'ordine di arrivo. Questo approccio non richiede una struttura complessa, ma potrebbe rallentare l'accesso ai dati se l'area di overflow diventa molto grande.
    - **Lista concatenata**: Ogni bucket può essere associato a una lista concatenata che contiene i record con lo stesso valore di hash. In questo modo, quando si verifica una collisione, il record viene semplicemente aggiunto alla lista collegata al bucket. Questa soluzione consente di mantenere un'organizzazione più ordinata e facilmente navigabile, ma comporta un ulteriore livello di indirezione (bisogna navigare nella lista concatenata per trovare il record desiderato).
2. **Riorganizzazione dei Bucket (Open Addressing)**: Un'altra tecnica per risolvere le collisioni è l'**open addressing**, che consiste nell'assegnare al record un'altra posizione nel file se la posizione calcolata è già occupata. In questo caso, quando si verifica una collisione, la funzione di hash può cercare una nuova posizione disponibile nel file, spostandosi in modo sequenziale o utilizzando altre strategie come il **linear probing** (scansione lineare delle posizioni) o il **quadratic probing** (scansione quadratica).

Indici di Accesso

Un **indice** è una struttura dati progettata per ottimizzare l'accesso ai record di un database, migliorando l'efficienza nelle operazioni di ricerca e recupero delle informazioni. Pur non essendo strettamente necessari per il funzionamento di un DBMS, gli indici sono strumenti fondamentali per velocizzare le query, riducendo i tempi di ricerca nei dati. In sostanza, un indice permette di localizzare rapidamente un record all'interno di un file senza dover eseguire una scansione completa del contenuto, migliorando notevolmente la performance del sistema.

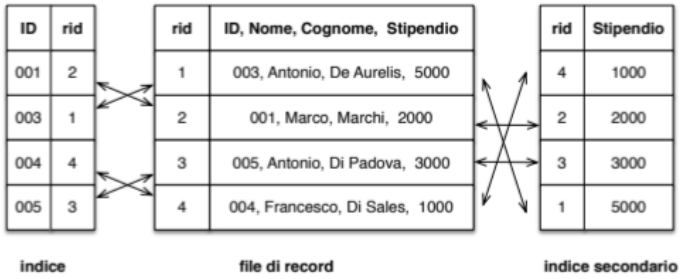
Gli indici sono legati a uno o più **campi di ricerca**, che costituiscono la **chiave di ricerca**. Questa chiave può essere un singolo campo o una combinazione di campi della tabella, e viene utilizzata per determinare rapidamente la posizione del record all'interno del file di dati. Ogni indice di solito associa alla chiave di ricerca un **identificatore di record (RID)**, che indica la posizione fisica del record nel file. In alcuni casi, un indice può essere progettato per memorizzare una **lista di identificatori** se diversi record condividono la stessa chiave di ricerca, permettendo così di gestire situazioni in cui la chiave non è univoca.

Tipi di Indici

Gli indici possono essere classificati in diverse tipologie, ciascuna con caratteristiche e vantaggi specifici. Il **indice primario** è uno dei più comuni e viene costruito su un file **sequenziale ordinato** in base alla **chiave primaria** di una relazione. Poiché la chiave primaria deve essere univoca, l'indice primario permette un accesso rapido e diretto ai record del file. Un file può avere solo un **indice primario**, in quanto una chiave primaria è univoca per ogni record.

Accanto agli indici primari, troviamo gli **indici secondari**, che vengono creati su una **chiave non primaria**. Gli indici secondari sono utili per ottimizzare le ricerche su colonne che non sono chiavi primarie ma che sono comunque frequentemente interrogate. A differenza degli indici primari, un file può avere **più indici secondari**, ciascuno dedicato a una colonna diversa che si desidera indicizzare.

Esiste anche il **indice di clustering**, che è costruito su un campo che non è una chiave primaria, ma che raggruppa i record in base ai valori di quel campo. In altre parole, un indice di clustering raggruppa i record che hanno lo stesso valore in un campo e li memorizza fisicamente vicini nel file di dati. Un file può avere **un solo indice di clustering**, poiché la sua funzione è quella di determinare l'ordine fisico dei record nel file.



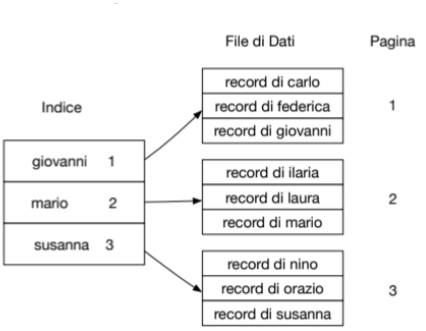
Indici Sparsi e Indici Densi

Oltre ai diversi tipi di indice, è possibile fare una distinzione tra **indici sparsi** e **indici densi**. Un **indice sparso** è un tipo di indice che contiene un record per solo alcuni valori della chiave di ricerca. In altre parole, non tutti i possibili valori della chiave sono indicizzati, ma solo quelli più significativi o utilizzati frequentemente nelle query. Questo tipo di indice è utile quando non è necessario avere una corrispondenza completa tra ogni record e la sua chiave di ricerca.

Un **indice denso**, invece, è un indice che contiene un record per ogni possibile valore della chiave di ricerca, assicurando che ogni singolo record del file dati sia rappresentato nell'indice. Sebbene più completo, questo tipo di indice può richiedere più spazio di memoria e una gestione più complessa, ma offre una precisione maggiore nelle operazioni di ricerca.

Indexed Sequential Files

Un **file sequenziale indicizzato** è un tipo di file ordinato che utilizza un indice primario per migliorare l'accesso ai dati. Questa struttura è stata sviluppata inizialmente con il metodo **ISAM** (Indexed Sequential Access Method) da IBM e successivamente evoluta in **VSAM** (Virtual Sequential Access Method) per un maggiore adattamento alle tecnologie di storage moderne.

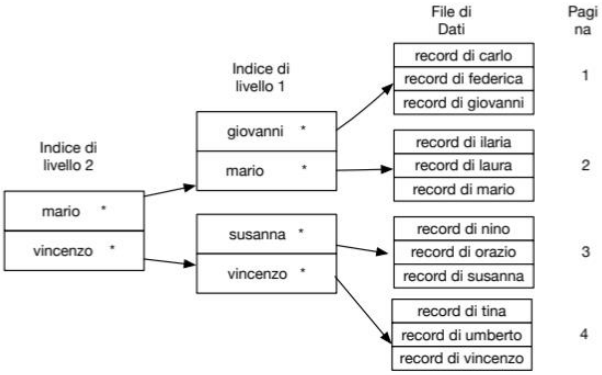


L'indice primario consente di accedere rapidamente ai record memorizzati nel file ordinato, riducendo notevolmente i tempi di ricerca rispetto a una ricerca sequenziale.

Il file sequenziale indicizzato è solitamente composto da tre principali sezioni: l'**area di memorizzazione primaria**, dove i dati vengono memorizzati in modo ordinato; un **indice primario**, che contiene le chiavi di ricerca e i puntatori ai record nel file sequenziale; e un'**area di overflow**, che gestisce l'aggiunta di nuovi record quando lo spazio dell'area primaria è esaurito. Grazie a questa struttura, la lettura e la scrittura dei dati possono essere effettuate in modo più efficiente.

Indice Multilivello

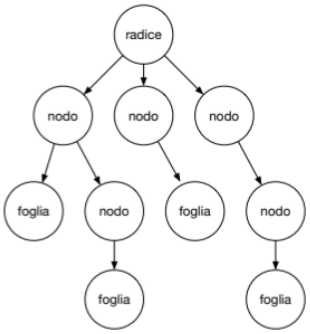
Un indice **multilivello** è utilizzato per organizzare file con un numero elevato di pagine, suddividendo il file in indici più piccoli. Ogni livello dell'indice contiene voci che puntano a blocchi di dati o ad altri indici di livello inferiore. In pratica, per accedere ai dati, è necessario attraversare diversi livelli di indici, simili a un indice degli indici. Questo approccio migliora la velocità di accesso ai dati, riducendo il numero di passaggi necessari per localizzare un record.



Indici B-Tree e B$^+$-Tree

Gli **indici B-Tree** e **B$^+$-Tree** sono strutture dati fondamentali per la gestione efficiente dei dati in un database. Entrambe le strutture sono progettate per consentire operazioni rapide di ricerca, inserimento e cancellazione su file di grandi dimensioni, garantendo un accesso bilanciato ai dati. Questi indici sono particolarmente utili per migliorare le prestazioni dei database, poiché riducono significativamente il numero di operazioni di I/O richieste per trovare un record o per fare operazioni su intervalli di dati.

Il B-Tree



Il **B-Tree** è una struttura ad albero auto-bilanciata, in cui ogni nodo interno ha più di un figlio e contiene un numero variabile di chiavi. Le chiavi nei nodi sono sempre ordinate, e ogni chiave agisce come una "guida" per indirizzare la ricerca ai nodi figli. La principale caratteristica del B-Tree è che tutti i nodi foglia sono alla stessa profondità, garantendo così un accesso uniforme ai dati.

Ogni nodo di un B-Tree contiene sia chiavi che puntatori ai sottolivelli (sottoalberi), il che permette di navigare rapidamente verso i record ricercati.

Il numero di chiavi che ogni nodo può contenere dipende dall'ordine dell'albero, e i nodi interni devono contenere almeno metà del numero massimo di chiavi consentito.

La struttura è progettata per ridurre al minimo il numero di accessi ai dischi, poiché ogni livello dell'albero contiene più chiavi e quindi consente di ridurre il numero di livelli complessivi.

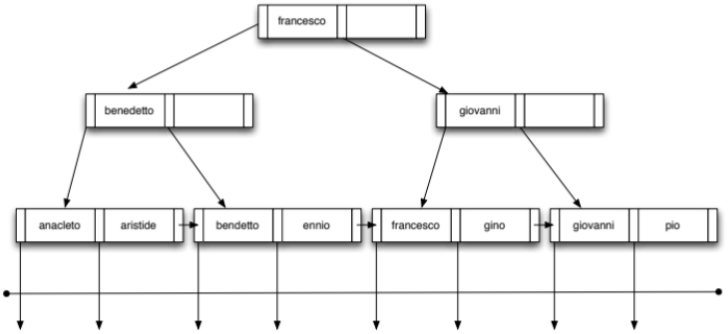
Le operazioni di **ricerca** in un B-Tree sono particolarmente efficienti grazie alla capacità di restringere progressivamente la ricerca a sottolivelli più piccoli, riducendo il numero di operazioni necessarie per trovare un dato. Anche le operazioni di **inserimento** e **cancellazione** sono abbastanza efficienti, ma possono comportare una ristrutturazione dell'albero, come la **divisione** dei nodi in caso di overflow o la **fusione** dei nodi in caso di sottoutilizzo.

Il B$^+$-Tree

Il **B$^+$-Tree** è una variante del B-Tree che ottimizza ulteriormente l'accesso ai dati. La differenza principale tra B-Tree e B$^+$-Tree è che, mentre nel B-Tree i nodi interni e foglia contengono sia chiavi che dati, nel B$^+$-Tree solo i nodi foglia contengono i dati veri e propri. I nodi interni, invece, contengono solo le chiavi per guidare la ricerca, ma non memorizzano i dati associati.

Questa separazione tra chiavi e dati rende il B$^+$-Tree più efficiente nelle operazioni di ricerca, poiché i nodi interni contengono solo chiavi di indice che riducono il carico di lavoro per la navigazione. In un B$^+$-Tree, i nodi foglia sono anche organizzati in una **lista concatenata**, che permette una scansione sequenziale molto efficiente dei dati, un'operazione particolarmente utile quando si lavora con intervalli di valori o con **query di intervallo**.

Un'altra caratteristica distintiva del B^+-Tree è che, poiché i nodi foglia sono concatenati, è più facile eseguire operazioni sequenziali come la lettura di tutti i record in ordine. Questo rende il B^+-Tree ideale per i database che devono gestire ricerche di intervallo o operazioni di scansione sequenziale, poiché consente di accedere ai dati in modo molto rapido.



Differenze tra B-Tree e B^+-Tree

La principale differenza tra i due alberi sta nell'organizzazione dei dati: nel B-Tree, sia i nodi interni che i nodi foglia contengono dati e chiavi, mentre nel B^+-Tree i nodi interni contengono solo chiavi e i dati sono concentrati nei nodi foglia. Questo porta a diverse implicazioni per l'efficienza. Nel B^+-Tree, la separazione tra chiavi e dati semplifica la struttura e permette una gestione più efficiente della memoria e dei dati, poiché i nodi interni contengono solo informazioni di navigazione.

Inoltre, i nodi foglia del B^+-Tree sono collegati tra loro, facilitando operazioni come la scansione di un intervallo di dati, che è più complessa in un B-Tree. Questo significa che il B^+-Tree è più adatto a scenari in cui le operazioni di intervallo sono frequenti, mentre il B-Tree può essere preferito in contesti dove non si prevede un uso intensivo di ricerche su intervalli.

Un'altra differenza importante è la gestione delle **operazioni di inserimento e cancellazione**. Poiché nel B^+-Tree i dati sono contenuti solo nei nodi foglia, ogni modifica (inserimento o cancellazione) coinvolge solo i nodi foglia, semplificando la gestione rispetto a un B-Tree, dove anche i nodi interni possono essere aggiornati.

Indici per Data Werehousing

Gli indici di Bitmap e gli indici di Join sono tecniche di ottimizzazione ampiamente utilizzate nei database, specialmente in contesti di data warehousing e analisi dati, per migliorare le prestazioni di query complesse. Entrambi permettono di velocizzare l'accesso ai dati, riducendo il tempo di risposta per le query e alleggerendo il carico sul sistema. Approfondiamo come funzionano e in quali casi trovano impiego.

Indici di Bitmap

Gli indici di Bitmap sono particolarmente efficaci quando vengono applicati a campi con un numero limitato di valori distinti, detti attributi a **bassa cardinalità**. Esempi tipici sono attributi come il genere (maschio o femmina), lo stato civile (single, sposato, divorziato) o variabili booleane (vero/falso). L'indice di Bitmap costruisce, per ogni valore dell'attributo, un **vettore di bit** che rappresenta la presenza o l'assenza del valore in ciascuna riga della tabella.

Nel dettaglio, per ogni valore dell'attributo, si crea una sequenza di bit lunga quanto il numero delle righe della tabella. Ogni bit corrisponde a una riga: il bit sarà **1** se la riga contiene quel valore specifico dell'attributo, **0** altrimenti.

Indici di Join

Gli indici di Join sono pensati per ottimizzare le query che richiedono di unire dati provenienti da due o più tabelle. Un join è un'operazione fondamentale nei database relazionali, che permette di combinare dati da più tabelle basandosi su colonne correlate, ma, su grandi quantità di dati, può diventare molto oneroso in termini di prestazioni.

Per risolvere questo problema, un indice di join pre-calcola l'unione tra due o più tabelle, salvando l'operazione di join e il risultato in una struttura di indice dedicata. In questo modo, quando si esegue una query che richiede il join tra le stesse tabelle, il database può accedere direttamente all'indice pre-calcolato anziché dover rifare l'intero processo di join.

Il Buffer Manager

La gestione della memoria in un sistema di gestione di basi di dati (DBMS) è fondamentale per garantire che le operazioni di lettura e scrittura sui dati siano eseguite nel modo più efficiente possibile. In questo contesto, il **Buffer Manager** (gestore del buffer) gioca un ruolo cruciale, poiché è responsabile del trasferimento di pagine di dati tra la memoria secondaria (disco) e la memoria centrale (RAM). Vediamo come funziona in dettaglio e quali politiche adotta per garantire efficienza e integrità dei dati.

Buffer e Buffer Manager: Definizioni e Ruoli

- **Buffer**: il buffer è un'area della memoria centrale, gestita dal DBMS, che serve come spazio temporaneo dove vengono memorizzate le pagine dei dati. È condiviso tra le varie transazioni e organizzato in unità chiamate **pagine**, che di solito hanno dimensioni pari o multipli dei blocchi di memoria secondaria (di solito tra 1 KB e 100 KB). L'obiettivo principale del buffer è ridurre il numero di accessi alla memoria secondaria, che sono costosi in termini di tempo rispetto all'accesso alla memoria centrale.
- **Buffer Manager**: il buffer manager è il modulo del DBMS che gestisce il buffer e si occupa di trasferire le pagine tra il disco e la RAM. La sua funzione è di assicurare che le pagine necessarie alle transazioni siano disponibili in memoria centrale e di ridurre al minimo gli accessi alla memoria secondaria.

Funzionamento del Buffer Manager

Il funzionamento del Buffer Manager si articola in diversi passaggi che ottimizzano l'utilizzo del buffer e la gestione della memoria. Quando un livello superiore del DBMS richiede una pagina specifica (per una lettura o una scrittura), il Buffer Manager deve:

1. **Controllare la presenza della pagina nel buffer**: se la pagina richiesta è già presente nel buffer, viene direttamente utilizzata senza bisogno di accedere alla memoria secondaria.

   In questo caso, viene aggiornato un contatore, detto **count**, che traccia quante transazioni stanno utilizzando quella pagina.

   Se la pagina viene modificata, viene segnato come **dirty** (sporco), indicandone la modifica in memoria centrale e la necessità di un eventuale aggiornamento sul disco.

2. **Caricare una pagina dal disco**: se la pagina richiesta non è presente nel buffer, il Buffer Manager deve leggere la pagina dalla memoria secondaria e trasferirla nel buffer. Tuttavia, se il buffer è già pieno, il gestore deve liberare spazio, adottando una strategia specifica per decidere quale pagina eliminare per fare posto alla nuova.

Strategie di Gestione della Memoria: Politiche FIFO e LRU

Il Buffer Manager può adottare diverse politiche per decidere quali pagine rimuovere dal buffer quando questo è pieno. Le principali politiche sono:

- **FIFO (First In, First Out)**: con la politica FIFO, la prima pagina caricata nel buffer è anche la prima a essere rimossa. È una politica semplice, ma non sempre ottimale, poiché non tiene conto dell'utilizzo recente della pagina.
- **LRU (Least Recently Used)**: questa politica cerca di mantenere in memoria le pagine utilizzate di recente, eliminando invece quelle che non sono state accedute da più tempo. LRU è più efficiente rispetto a FIFO in molti casi, poiché le pagine utilizzate di recente hanno una maggiore probabilità di essere richieste nuovamente.

Funzionamento Dettagliato del Buffer Manager

Per garantire un corretto funzionamento, il Buffer Manager gestisce variabili di stato come **count** e **dirty** per ciascuna pagina, e segue una serie di passaggi:

- **Contatore di utilizzo (count)**: ogni pagina ha un contatore che traccia il numero di transazioni che stanno usando quella pagina. Questo contatore aumenta ogni volta che una transazione accede alla pagina e diminuisce quando una transazione la rilascia.

- **Stato "sporco" (dirty)**: il flag **dirty** indica se la pagina è stata modificata mentre si trova nel buffer. Se la pagina è stata modificata (cioè ha dirty = 1), deve essere scritta sul disco prima di essere rimossa dal buffer per garantire la persistenza delle modifiche.

Il processo di gestione delle pagine avviene nel modo seguente:

1. **Richiesta di pagina**: quando il Buffer Manager riceve una richiesta di lettura o scrittura per una pagina, verifica se la pagina è già presente nel buffer.
2. **Incremento di count e aggiornamento di dirty**: se la pagina è già presente, il valore di **count** viene incrementato di 1. Se la pagina viene modificata, il flag **dirty** viene posto a 1.
3. **Sostituzione di una pagina**: se la pagina richiesta non è nel buffer e non ci sono pagine libere, il Buffer Manager seleziona una pagina da sostituire usando una politica come FIFO o LRU.
    - Se la pagina selezionata per la sostituzione è **dirty**, il Buffer Manager la scrive su disco per assicurarsi che tutte le modifiche siano salvate.
    - Se non ci sono pagine libere, può adottare strategie come **steal** o **no-steal**:

      Steal: il Buffer Manager può prelevare (steal) una pagina ancora in uso da un'altra transazione, salvandola su disco se è sporca.

      No-steal: il Buffer Manager evita di rimuovere pagine ancora in uso e cerca di aspettare che una pagina venga rilasciata.
4. **Caricamento della nuova pagina**: una volta liberato lo spazio, la pagina richiesta viene caricata nel buffer. Il valore **count** è impostato a 1, e **dirty** viene inizialmente posto a 0.

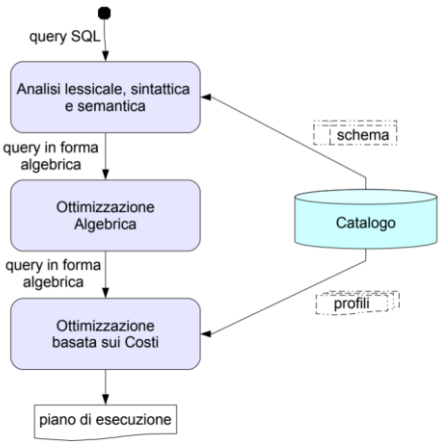
Query Processor

Il **Query Processor** (gestore delle query) è una componente centrale di un sistema di gestione di basi di dati (DBMS), responsabile di prendere in carico le query SQL, verificarne la correttezza, ottimizzarle e produrre un piano di esecuzione che consenta al database di rispondere alle richieste nel modo più efficiente possibile. Il Query Processor si avvale di varie tecniche di ottimizzazione, sia algebriche che basate sui costi, per minimizzare il tempo di risposta e l'uso delle risorse.

Processo di Ottimizzazione delle Query

Il processo di ottimizzazione delle query comprende diverse fasi, ciascuna delle quali è mirata a migliorare l'efficienza dell'esecuzione:

Analisi della Query: In questa fase, la query SQL viene sottoposta a controlli di tipo lessicale, sintattico e semantico. Questi controlli sono necessari per verificare che la query sia ben formata e valida rispetto al modello del database. Le informazioni relative allo schema del database sono memorizzate nel catalogo del DBMS, che viene consultato in questa fase per accertare la correttezza della query:



- **Analisi lessicale**: esamina i token della query, ossia le parole chiave e i simboli, per assicurarsi che la sintassi sia corretta.
- **Analisi sintattica**: verifica che la struttura della query sia conforme alle regole grammaticali del linguaggio SQL.
- **Analisi semantica**: verifica che gli attributi e le tabelle citati nella query esistano e che le operazioni richieste siano valide rispetto ai tipi di dati e alle relazioni definite nel database.

Alla fine di questa fase, la query viene tradotta in una **forma algebrica** usando l'algebra relazionale, ovvero una rappresentazione che consente di manipolare la query attraverso trasformazioni equivalenti.

Ottimizzazione Algebrica: La query in forma algebrica viene ulteriormente trasformata applicando regole di equivalenza dell'algebra relazionale. Lo scopo è quello di ottenere una versione della query che sia logicamente

equivalente a quella originale ma più efficiente da eseguire. Questa fase si basa sul concetto di **equivalenza algebrica**, per cui due espressioni sono equivalenti se producono lo stesso risultato indipendentemente dall'istanza attuale del database. Esempi di ottimizzazioni algebriche includono:

- **Riordinamento delle operazioni**: ad esempio, applicare prima le selezioni più restrittive per ridurre il numero di tuple da processare in operazioni successive.
- **Pushing di selezioni e proiezioni**: spostare le operazioni di selezione e proiezione il più vicino possibile alle operazioni di scansione delle tabelle, riducendo il numero di righe e colonne da elaborare nelle fasi successive.
- **Eliminazione di ridondanze**: semplificare o eliminare parti della query che risultano ridondanti o superflue.

Concetto di Equivalenza Algebrica

Alla base dell'ottimizzazione algebrica vi è il concetto di **equivalenza algebrica** dell'algebra relazionale. Due espressioni dell'algebra relazionale sono dette equivalenti se restituiscono lo stesso risultato per qualsiasi stato attuale della base di dati. Questo permette di trasformare la query iniziale in una forma equivalente più efficiente, senza alterare il risultato finale

Ottimizzazione Basata sui Costi: Una volta ottenuta una forma algebrica ottimizzata, il Query Processor utilizza informazioni dettagliate sulle caratteristiche del database (ad esempio, il numero di righe nelle tabelle, la presenza di indici e le distribuzioni di valori degli attributi) per stimare i costi associati a varie strategie di esecuzione della query. Questa stima dei costi permette di determinare il **piano di esecuzione** più efficiente per eseguire la query.

Al termine di questa fase, viene scelto un **piano di esecuzione finale**, che è una sequenza di operazioni di basso livello, come scansioni di tabelle, accesso agli indici e join, progettata per minimizzare il costo stimato.

Piani di Query e Ottimizzazione delle Query

La progettazione dei piani di esecuzione delle query è una componente fondamentale del funzionamento di un Database Management System (DBMS). Un piano di query rappresenta una sequenza di operazioni di basso livello, ottimizzate per ottenere il risultato desiderato con il minor costo computazionale possibile. Durante l'elaborazione di una query, il DBMS utilizza informazioni statistiche sulla struttura e sui dati presenti nella base di dati per stimare le dimensioni dei risultati intermedi e scegliere l'ordine di esecuzione ottimale delle operazioni.

Le operazioni più comuni includono scansioni (che analizzano l'intero contenuto di una tabella), accessi diretti (che utilizzano indici per recuperare rapidamente specifici record), ordinamenti e operazioni di join (che combinano dati provenienti da più tabelle). L'obiettivo dell'ottimizzazione è ridurre al minimo il tempo di esecuzione della query e l'uso delle risorse del sistema.

Un piano di query può essere determinato in due modalità principali:

- **Compile & Store:** In questa modalità, il piano di esecuzione viene generato una sola volta e memorizzato nel catalogo del DBMS. Questo approccio è particolarmente utile per query che devono essere eseguite frequentemente e non subiscono variazioni, poiché consente di evitare l'overhead di calcolare il piano ogni volta.
- **Compile & Go:** In questo caso, il piano di esecuzione viene determinato dinamicamente ogni volta che la query viene eseguita. Questo approccio è più adatto a query occasionali o con parametri variabili, dove i cambiamenti nei dati potrebbero influenzare la scelta del piano ottimale.

Gestione delle Transazioni nei DBMS

Un aspetto cruciale nella gestione di un DBMS è garantire che le operazioni simultanee effettuate da più utenti o programmi applicativi non compromettano la consistenza e l'integrità della base di dati. Questo compito è reso complesso dal numero elevato di operazioni concorrenti che un sistema può trovarsi a gestire in un ambiente multi-utente.

Definizione di Transazione

Una transazione rappresenta un'unità logica di elaborazione all'interno di un DBMS. Essa consiste in una sequenza di operazioni, quali letture e scritture sulla base di dati, che vengono eseguite come un blocco unico. Può corrispondere

all'esecuzione di un intero programma, a una sua parte o a un singolo comando SQL, come un'istruzione `INSERT`, `UPDATE` o `DELETE`.

L'aspetto fondamentale di una transazione è la sua indivisibilità. Questo implica che le modifiche effettuate durante una transazione sono atomiche: devono essere tutte applicate alla base di dati oppure nessuna. Una transazione, inoltre, porta la base di dati da uno stato consistente a un altro stato consistente, nel rispetto dei vincoli definiti sul modello dei dati.

Stati di una Transazione: Commit e Abort

Al termine della sua esecuzione, una transazione può concludersi in due modi:

- **Commit:** Se la transazione si completa con successo, tutte le modifiche effettuate sono confermate e registrate permanentemente nella base di dati. A questo punto, il nuovo stato della base di dati diventa visibile agli altri utenti e applicazioni.
- **Abort:** Se durante l'esecuzione si verifica un errore (ad esempio, un vincolo di integrità non viene rispettato, si verifica un crash del sistema o si decide di annullare la transazione), tutte le modifiche effettuate vengono annullate. Questo processo è noto come rollback e garantisce che la base di dati venga riportata al suo stato iniziale, evitando di lasciare effetti parziali o inconsistenti.

Proprietà ACID delle Transazioni

Un DBMS garantisce il corretto funzionamento delle transazioni mediante il rispetto delle proprietà ACID, che rappresentano i requisiti fondamentali per la gestione dei dati in un ambiente multi-utente:

1. **Atomicità:**
   La proprietà di atomicità assicura che una transazione venga eseguita interamente o per nulla. Se una parte della transazione fallisce, tutte le modifiche parziali vengono annullate. Questo comportamento protegge la base di dati da stati intermedi incoerenti.
2. **Consistenza:**
   Una transazione deve sempre portare la base di dati da uno stato consistente a un altro. Ciò significa che i vincoli definiti sul modello dei dati (come chiavi primarie, vincoli di unicità o di integrità referenziale) devono essere rispettati durante l'esecuzione della transazione.
3. **Isolamento:**
   In un ambiente con transazioni concorrenti, l'isolamento garantisce che le transazioni vengano eseguite in modo indipendente. Gli effetti parziali di una transazione in corso non devono essere visibili ad altre transazioni fino al momento del commit. Questo evita problemi come letture inconsistenti o "dirty reads".
4. **Durabilità (Persistenza):**
   Una volta che una transazione viene confermata tramite commit, i suoi effetti devono essere permanenti, anche in caso di guasti del sistema o interruzioni improvvise. Per garantire questa proprietà, il DBMS utilizza tecniche come il logging e i checkpoint.

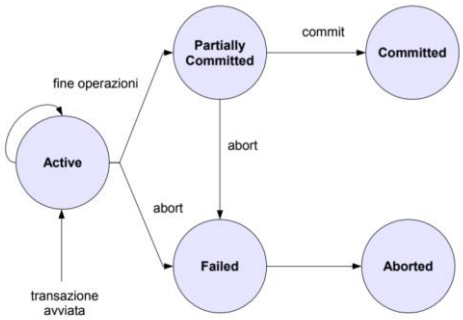
Gestione della Concorrenza e dell'Affidabilità

Per supportare un accesso simultaneo alla base di dati da parte di più utenti o applicazioni, il DBMS include moduli dedicati alla gestione della concorrenza e dell'affidabilità.

- **Gestore della Concorrenza:**
  Questo modulo si occupa di garantire che le transazioni concorrenti non interferiscano tra loro, preservando l'integrità e la consistenza della base di dati. Tecniche comuni includono il locking (blocco delle risorse) e il timestamping, che assicurano un'esecuzione controllata delle transazioni.
- **Gestore dell'Affidabilità:**
  Questo modulo protegge la base di dati da eventi imprevedibili, come crash del sistema o interruzioni di corrente. Attraverso tecniche come il logging (registro delle operazioni) e il ripristino (recovery), il sistema è in grado di annullare le transazioni incomplete e ripristinare la base di dati a uno stato consistente.

Diagramma degli Stati di una Transazione

Il **diagramma degli stati di una transazione** rappresenta i diversi stadi che una transazione può attraversare durante la sua esecuzione. Ogni stato corrisponde a una specifica condizione della transazione, dal momento in cui viene avviata fino a quando termina con successo (commit) o viene annullata (abort). Di seguito, viene fornita una descrizione dettagliata degli stati e del flusso tra di essi:



Stati di una Transazione

1. **Active (Attivo):**
   La transazione è stata avviata ed è in esecuzione.
   Durante questo stato, vengono eseguite le istruzioni della transazione, come letture, scritture e altre operazioni sulla base di dati. Questo è lo stato iniziale di ogni transazione.
   Una transazione può lasciare lo stato *Active* nei seguenti casi:
   - Passa allo stato *Partially Committed* dopo aver completato tutte le istruzioni con successo.
   - Entra nello stato *Failed* in caso di errore (ad esempio, violazione di vincoli di integrità o crash del sistema).
2. **Partially Committed (Parzialmente Committed):**
   La transazione si trova in questo stato quando ha completato con successo l'esecuzione dell'ultima istruzione prevista. Tuttavia, non è ancora completamente *Committed* perché i dati aggiornati potrebbero non essere stati ancora scritti su memoria secondaria (persistenza).
   In questa fase, la transazione è ancora vulnerabile:
   - Se si verifica un errore (ad esempio, un crash di sistema), passa allo stato *Failed*.
   - Se invece tutti gli aggiornamenti vengono salvati correttamente su disco, passa allo stato *Committed*.
3. **Committed (Confermato):**
   La transazione ha completato con successo tutte le operazioni e tutti i suoi effetti sono stati registrati permanentemente nella base di dati. Questo stato rappresenta la conclusione positiva della transazione.
4. **Failed (Fallito):**
   Una transazione entra nello stato *Failed* quando non può essere completata con successo. Questo può accadere per vari motivi, come la violazione di un vincolo di integrità o un'interruzione improvvisa (crash). Una transazione nello stato *Failed* è destinata a essere abortita.
5. **Aborted (Annullato):**
   La transazione è stata annullata e tutte le modifiche apportate sono state annullate (rollback). Lo stato della base di dati è stato riportato a quello precedente all'inizio della transazione. In alcuni casi, è possibile riavviare la transazione dopo un abort, tentando una nuova esecuzione.

Gestione della Concorrenza nel DBMS

La gestione della concorrenza è un aspetto cruciale nei DBMS, in particolare nei sistemi **OLTP (Online Transaction Processing)**, dove decine o centinaia di transazioni vengono generate ogni secondo. L'obiettivo principale è consentire l'accesso simultaneo ai dati condivisi, garantendo al contempo la consistenza e l'integrità dei dati.

Problemi legati agli accessi concorrenti

Quando più transazioni accedono contemporaneamente agli stessi dati, possono verificarsi anomalie come:

1. **Dirty Read:** Una transazione legge dati modificati da un'altra transazione che non ha ancora effettuato il commit.
2. **Non-Repeatable Read:** Una transazione legge lo stesso dato due volte, ma ottiene risultati diversi perché un'altra transazione ha effettuato un aggiornamento nel frattempo.

3. **Phantom Read:** Una transazione rileva l'inserimento o l'eliminazione di nuovi record da parte di un'altra transazione durante la sua esecuzione.

Serializzazione: la soluzione ideale, ma impraticabile

Una soluzione banale a questi problemi sarebbe eseguire le transazioni in modo strettamente sequenziale (serializzazione). Tuttavia, questa strategia è impraticabile nei sistemi moderni, poiché ridurrebbe drasticamente il throughput e aumenterebbe i tempi di risposta. Nei sistemi OLTP, il DBMS deve essere in grado di eseguire operazioni in concorrenza massimizzando le transazioni per secondo (TPS).

Soluzioni al Problema della Concorrenza nei DBMS

Nei moderni sistemi di gestione delle basi di dati (DBMS), l'accesso concorrente ai dati è una caratteristica fondamentale. Tuttavia, questa concorrenza può portare a problemi di integrità e consistenza, soprattutto quando più transazioni accedono simultaneamente agli stessi dati. L'obiettivo primario del DBMS è garantire che tali accessi simultanei non compromettano la coerenza del sistema, permettendo nel contempo un'elevata efficienza operativa. Questo è particolarmente critico nei sistemi OLTP, dove il volume di transazioni al secondo (TPS) è molto elevato e il tempo di risposta deve rimanere basso per garantire prestazioni ottimali.

Un approccio teorico, e al tempo stesso semplice, per risolvere i conflitti potrebbe consistere nell'eseguire tutte le transazioni in modo strettamente sequenziale. In altre parole, ogni transazione verrebbe completata interamente prima che un'altra abbia inizio. Tuttavia, questa soluzione non è praticabile, poiché comporterebbe un drastico rallentamento del sistema, rendendolo inadatto alle necessità operative delle applicazioni moderne. Di conseguenza, il DBMS adotta strategie più sofisticate per alternare e sovrapporre l'esecuzione di transazioni (esecuzione interleaved) senza sacrificare l'affidabilità dei dati.

Per raggiungere questo equilibrio tra concorrenza ed efficienza, il DBMS implementa specifiche tecniche di controllo della concorrenza. Queste tecniche si concentrano sull'evitare situazioni problematiche, come la possibilità che una transazione legga dati parzialmente aggiornati (dirty read) o che il risultato finale sia influenzato da una sovrapposizione non gestita correttamente.

Locking (Gestione dei Blocchi)

Uno dei metodi principali utilizzati dai DBMS per prevenire interferenze tra transazioni è la gestione dei blocchi, o **locking**. Questo approccio si basa sull'idea di assegnare un controllo temporaneo su determinati dati a una transazione, impedendo ad altre di accedere o modificare quegli stessi dati fino a quando il controllo non viene rilasciato.

Ad esempio, supponiamo che una transazione A stia aggiornando un record specifico. Durante questa operazione, un lock esclusivo (exclusive lock) viene posto sul record, garantendo che nessun'altra transazione possa né leggere né modificare quel dato fino al completamento dell'operazione. In modo simile, una transazione che deve solo leggere i dati può acquisire un lock condiviso (shared lock), consentendo ad altre transazioni di leggere gli stessi dati ma impedendo aggiornamenti simultanei.

Questa strategia è particolarmente efficace, ma non è priva di rischi. Problemi come il *deadlock*, ovvero una situazione in cui due o più transazioni restano bloccate in attesa di risorse che non possono essere rilasciate, devono essere gestiti con attenzione tramite algoritmi di rilevamento e risoluzione.

Timestamp Ordering

Un altro approccio per garantire l'integrità dei dati è basato sull'ordinamento temporale delle transazioni. Ogni transazione riceve un timestamp univoco al momento della sua creazione, che determina l'ordine in cui le operazioni devono essere eseguite. Questo metodo assicura che le transazioni più "vecchie" abbiano priorità su quelle più recenti, evitando così situazioni in cui le transazioni più nuove sovrascrivano modifiche ancora in corso.

Ad esempio, immaginiamo due transazioni, T1 e T2, che tentano di accedere a un dato condiviso. Se T1 ha un timestamp precedente rispetto a T2, al primo tentativo di accesso T2 sarà messa in attesa fino al completamento di T1. Questo sistema elimina il rischio di anomalie legate alla sovrapposizione delle operazioni, anche se potrebbe comportare un aumento del tempo di attesa in situazioni di carico elevato.

Multiversion Concurrency Control (MVCC)

Un approccio più avanzato e flessibile per gestire la concorrenza è rappresentato dal controllo della concorrenza multiversione, noto come MVCC. Questo metodo consente di mantenere più versioni dello stesso dato, permettendo alle transazioni di accedere a versioni diverse a seconda del loro stato. Ad esempio, una transazione in lettura può accedere a una versione "vecchia" di un dato, mentre una transazione in scrittura lavora su una versione aggiornata.

Questo approccio elimina completamente i blocchi di lettura, aumentando significativamente le prestazioni in scenari con elevato numero di operazioni di lettura. Tuttavia, la complessità nella gestione delle versioni può comportare un maggiore utilizzo di memoria e tempi più lunghi per operazioni di garbage collection.

Two-Phase Locking (2PL)

Un'altra tecnica ampiamente utilizzata è la pianificazione a due fasi, o **Two-Phase Locking (2PL)**. In questo approccio, ogni transazione attraversa due fasi distinte:

1. **Fase di acquisizione dei lock:** Durante questa fase, la transazione acquisisce tutti i lock necessari per completare le sue operazioni.
2. **Fase di rilascio dei lock:** Una volta che la transazione ha acquisito tutti i lock e completato le operazioni, i lock vengono rilasciati.

Questo modello garantisce che una transazione non possa interferire con un'altra che si trova in una fase diversa, ma richiede un'attenta implementazione per evitare situazioni di stallo o inutili ritardi.

Accesso Concorrente nei Sistemi di Basi di Dati

Un sistema di basi di dati deve garantire accesso concorrente ai dati condivisi preservandone integrità e consistenza. L'accesso simultaneo, tuttavia, può introdurre **anomalie** che compromettono i dati, soprattutto quando transazioni si sovrappongono parzialmente.

Principali Anomalie nei Sistemi di Basi di Dati

Le anomalie più comuni che si verificano in un contesto di accesso concorrente sono:

1. **Perdita di Aggiornamento (Lost Update):** Quando due transazioni aggiornano lo stesso dato senza sincronizzazione, una sovrascrive l'altra, annullandone di fatto l'operazione. Questo avviene se una transazione non completa il proprio ciclo prima che l'altra intervenga.
2. **Lettura Sporca (Dirty Read):** Una transazione legge un valore modificato da un'altra che non ha ancora confermato le proprie operazioni tramite commit. Se la transazione che ha effettuato l'aggiornamento iniziale viene poi annullata (rollback), il dato letto risulta incoerente.
3. **Aggiornamento Fantasma (Ghost Update):** Si verifica quando una transazione aggiorna più oggetti correlati, ma un'altra transazione sovrapposta non rileva correttamente tutti gli aggiornamenti. Per esempio, una transazione potrebbe ignorare vincoli d'integrità tra dati correlati, rendendo alcuni aggiornamenti "invisibili".

Soluzioni Tradizionali e Limiti dell'Accesso Serializzato

Un approccio banale per evitare queste anomalie è eseguire le transazioni in modo serializzato, una alla volta, garantendo che nessuna interferisca con le altre. Questo metodo, però, è impraticabile nei sistemi OLTP (Online Transaction Processing), dove centinaia di transazioni al secondo richiedono tempi di risposta rapidi. L'accesso seriale, infatti, ridurrebbe drasticamente il throughput, penalizzando l'efficienza complessiva del sistema.

Controllo della Concorrenza (CdC): Teoria

Il controllo della concorrenza è un elemento essenziale nella gestione delle basi di dati moderne. Quando più transazioni accedono simultaneamente agli stessi dati, è necessario un sistema che garantisca la coerenza e l'integrità delle informazioni, evitando conflitti o anomalie che potrebbero compromettere l'affidabilità del sistema. In questa prospettiva, ogni transazione viene vista come una sequenza ordinata di operazioni di lettura e scrittura su oggetti della base di dati.

Per esempio, si possono considerare due transazioni come segue:

- **T1**: legge un oggetto, lo aggiorna, poi legge e aggiorna altri oggetti (es. r1(X), w1(X), r1(Y), r1(Z), w1(Z)).

- **T2**: legge vari oggetti e aggiorna uno di essi (es. r2(X), r2(Y), r2(Z), w2(Y)).

Nella pratica, spesso vengono omessi dettagli come i comandi di inizio transazione (begin transaction), conferma (commit) o annullamento (rollback), poiché l'interesse si concentra principalmente sulle interazioni tra le operazioni e sugli effetti complessivi sui dati.

Concetti Chiave e Definizioni

Per gestire correttamente la concorrenza tra transazioni, è utile definire alcuni concetti fondamentali:

- **Schedule**: uno schedule è una sequenza di operazioni di lettura e scrittura generate da un insieme di transazioni concorrenti. Deve rispettare l'ordine temporale delle operazioni all'interno di ciascuna transazione.
- **Scheduler**: lo scheduler è il componente del sistema che accetta, rifiuta o riordina le operazioni richieste dalle transazioni, al fine di preservare la consistenza dei dati.

Un aspetto cruciale è rappresentato dal **commit**, il punto in cui una transazione raggiunge uno stato consistente. Quando una transazione viene annullata tramite abort, tutte le sue operazioni vengono rimosse dallo schedule.

Tipi di Schedule

Gli schedule possono essere classificati in base al loro comportamento:

- **Schedule seriale**: in questo caso, tutte le operazioni di una transazione vengono completate prima che inizi la successiva. Sebbene sia il metodo più semplice e sicuro, risulta inefficiente in termini di prestazioni.
- **Schedule serializzabile**: rappresenta uno schedule non seriale che garantisce tuttavia lo stesso risultato di uno schedule seriale. Questo approccio consente di bilanciare concorrenza e consistenza.

Obiettivi del Controllo della Concorrenza

Il controllo della concorrenza ha l'obiettivo di assicurare che l'esecuzione concorrente delle transazioni produca risultati consistenti e corretti. Per raggiungere questo scopo, è necessario, creare schedule serializzabili, evitando conflitti tra transazioni e implementare tecniche efficienti per verificare la serializzabilità, minimizzando i costi computazionali.

I sistemi di gestione delle basi di dati (DBMS) moderni adottano meccanismi che garantiscono automaticamente la serializzabilità. Questi si dividono principalmente in due categorie:

Metodi Basati sui Lock

I metodi basati sui **lock** si fondano sull'idea di proteggere ogni oggetto del database con un meccanismo di blocco. Ogni volta che una transazione vuole accedere a un oggetto, deve prima acquisire un lock appropriato. Esistono due tipi principali di lock:

1. **Read Lock (blocco di lettura)**: Questo tipo di lock consente a una transazione di leggere un oggetto. Più transazioni possono acquisire contemporaneamente un read lock sullo stesso oggetto, perché la lettura non altera lo stato dell'oggetto.
2. **Write Lock (blocco di scrittura)**: Questo tipo di lock è esclusivo e permette di modificare un oggetto. Solo una transazione per volta può acquisire un write lock su un determinato oggetto, impedendo così che altre transazioni possano leggerlo o scriverlo contemporaneamente.

Un aspetto importante è che, se una transazione intende prima leggere e poi scrivere un oggetto, può richiedere inizialmente un read lock e poi effettuare un'**escalation** al write lock. Questo approccio riduce il livello di contesa sugli oggetti del database, poiché il lock esclusivo viene acquisito solo quando strettamente necessario.

Una volta terminata l'operazione, la transazione deve **rilasciare il lock** acquisito con un'operazione di unlock, restituendo così l'oggetto al suo stato libero e permettendo ad altre transazioni di accedervi.

Ruolo del Lock Manager

La gestione dei lock è affidata a una componente del DBMS chiamata **lock manager**, che riceve le richieste di blocco dalle transazioni e decide se concederle o rifiutarle, basandosi su una **tabella dei conflitti**. In questa tabella, si definisce

se una richiesta di lock può essere accettata, considerando lo stato attuale dell'oggetto (libero, bloccato in lettura o bloccato in scrittura).

Two-Phase Locking (2PL) e delle sue Varianti

Il protocollo **Two-Phase Locking (2PL)** è uno dei metodi più utilizzati nei database per garantire la **serializzabilità** delle transazioni. La serializzabilità è una proprietà fondamentale per assicurarsi che le transazioni concorrenti, eseguite contemporaneamente, non compromettano l'integrità dei dati. Il 2PL viene applicato per evitare la cosiddetta **incoerenza serializzabile**, in cui l'ordine delle operazioni tra transazioni non rispetta un ordine seriale (un ordine che sarebbe stato eseguito se le transazioni fossero eseguite una dopo l'altra, senza sovrapposizioni).

Il principio base del 2PL si fonda sul controllo dell'accesso agli oggetti del database tramite **lock** (blocco), in modo che le transazioni non possano interferire tra di loro. Esso prevede due fasi principali nell'esecuzione di una transazione: una fase crescente e una fase decrescente.

Fase Crescente del 2PL

La **fase crescente** è la fase in cui una transazione acquisisce tutti i **lock** necessari per compiere le operazioni di lettura e scrittura sugli oggetti del database.

- **Acquisizione dei lock**: Una transazione può acquisire lock in lettura (read lock) o in scrittura (write lock), ma **non può rilasciare alcun lock durante questa fase**. Il suo scopo principale in questa fase è quello di raccogliere tutti i lock necessari per garantire che nessun'altra transazione possa interferire con la sua esecuzione.
- **Obiettivo**: Durante questa fase, la transazione si prepara a completare tutte le sue operazioni, proteggendo le risorse (oggetti del database) da accessi concorrenti che potrebbero causare incoerenza. Questo impedisce che due transazioni possano scrivere nello stesso oggetto nello stesso momento, o che una transazione legga un dato che sta per essere modificato da un'altra transazione.

Fase Decrescente del 2PL

La **fase decrescente** si verifica una volta che la transazione ha acquisito tutti i lock necessari e ha completato le operazioni di lettura e scrittura sugli oggetti. Ora, la transazione può iniziare a **rilasciare i lock**.

- **Rilascio dei lock**: La transazione inizia a rilasciare i lock che aveva acquisito nella fase crescente. Tuttavia, un aspetto cruciale del protocollo è che, una volta che una transazione inizia a rilasciare un lock, **non può più acquisirne di nuovi**.
- **Obiettivo**: La fase decrescente segna la fine delle operazioni di scrittura o lettura su oggetti protetti da lock. Quando la transazione ha rilasciato tutti i lock, è terminata e può essere **committata** (finalizzata), assicurando che tutte le modifiche siano permanenti, o **abbandonata** (abort), nel caso in cui si verifichi un errore.

Il **vincolo chiave** del 2PL è che una volta che una transazione rilascia un lock, non può acquisirne di nuovi. Questo impedisce che una transazione possa continuare a modificare un oggetto mentre sta già rilasciando altri lock, evitando così conflitti tra transazioni che potrebbero portare a uno stato inconsistente dei dati.

Strict 2PL

Il **Strict 2PL** è una variante del 2PL che aggiunge una restrizione importante per migliorare ulteriormente la coerenza dei dati. In particolare, nel **Strict 2PL**, i lock acquisiti da una transazione **non vengono rilasciati fino al commit** o all'abort della transazione. Questo approccio ha un impatto significativo sul controllo delle anomalie e sull'affidabilità generale del sistema.

- **Rilascio dei lock post-commit**: In Strict 2PL, i lock vengono mantenuti fino al commit (finalizzazione) della transazione o all'abort (annullamento). Ciò significa che le modifiche fatte dalla transazione non sono visibili ad altre transazioni finché la transazione non è completamente conclusa. Questa restrizione previene fenomeni come le **letture sporche (dirty reads)**, in cui una transazione legge dati che potrebbero essere annullati (abortiti) da un'altra transazione.

- **Affidabilità maggiore**: Con **Strict 2PL**, le transazioni sono più sicure poiché non possono mai leggere dati che potrebbero essere modificati o annullati da altre transazioni. Questo aumenta la **consistenza** e l'affidabilità complessiva del sistema, riducendo la possibilità di ottenere uno stato del database errato o incoerente.

Problemi del 2PL: Deadlock

Uno dei problemi principali associati all'uso dei **lock** è il **deadlock**. Un deadlock si verifica quando due o più transazioni sono in uno stato di stallo, ciascuna in attesa che un'altra transazione rilasci un lock su un oggetto. Ad esempio:

1. La **Transazione A** ha un lock di scrittura su oggetto X e sta aspettando un lock di lettura su oggetto Y.
2. La **Transazione B** ha un lock di scrittura su oggetto Y e sta aspettando un lock di lettura su oggetto X.

In questo scenario, entrambe le transazioni sono **bloccate**, poiché ognuna sta aspettando un lock che l'altra non può rilasciare finché non completa la sua operazione. Questo porta a una **situazione di deadlock**, dove nessuna delle transazioni può proseguire.

Metodi Basati sul Timestamp

I metodi basati sul timestamp offrono un approccio alternativo ai lock per garantire la serializzabilità delle transazioni, utilizzando l'ordine temporale come criterio principale per gestire l'accesso concorrente ai dati. L'idea fondamentale è che ogni transazione riceve un timestamp univoco al momento del suo inizio, e l'ordine di esecuzione delle operazioni è determinato da questo valore temporale. Questo consente di evitare la contesa esplicita sui dati e i relativi problemi associati, come i deadlock.

Principio di Funzionamento

Ogni transazione T_i riceve un **timestamp univoco** $TS(T_i)$ all'avvio. Questo timestamp rappresenta il momento in cui la transazione è iniziata e stabilisce la sua priorità rispetto alle altre. All'interno del database, per ogni oggetto X, vengono mantenuti due valori temporali associati al suo stato:

- **TSR(X)**: il più recente timestamp della transazione che ha **letto** l'oggetto X;
- **TSW(X)**: il più recente timestamp della transazione che ha **scritto** sull'oggetto X.

Questi valori vengono aggiornati dinamicamente in base alle operazioni eseguite dalle transazioni. Il controllo di concorrenza si basa sul confronto tra il timestamp della transazione e questi valori temporali per determinare se l'operazione è consentita.

Regole di Conflitto

Per garantire la serializzabilità, il metodo stabilisce delle regole per l'accesso agli oggetti, distinguendo tra le operazioni di lettura e scrittura.

1. **Operazione di Lettura**:

   Se il timestamp della transazione è **minore di TSW(X)**, significa che la transazione sta tentando di leggere un valore sovrascritto da una transazione successiva. Questo renderebbe impossibile mantenere un ordine seriale coerente. Di conseguenza, l'operazione viene **rifiutata** e T_i deve essere abortita e riavviata con un nuovo timestamp.

   Se il timestamp della transazione è maggiore o uguale a **TSW(X),** l'operazione viene accettata. In questo caso, il valore viene aggiornato al timestamp, indicando che l'oggetto è stato letto da una transazione con quel timestamp.

2. **Operazione di Scrittura**:

   Se il timestamp della transazione è **minore di TSR(X)**, significa che T_i sta tentando di sovrascrivere un valore che è stato già letto da una transazione successiva. Per evitare anomalie, l'operazione viene **rifiutata** e la transazione deve essere abortita e riavviata.

Se il timestamp della transazione è **minore di TSW(X)**, significa che si sta tentando di scrivere su un oggetto che è già stato modificato da una transazione successiva. Anche in questo caso, l'operazione viene **rifiutata** e la transazione abortisce.

Se nessuna delle condizioni precedenti si verifica, l'operazione è accettata. Il valore del timestamp in scrittura viene aggiornato a $TS(T_i)$, e l'oggetto può essere modificato.

Controllo di Affidabilità nei Sistemi di Gestione di Basi di Dati (DBMS)

Il **controllo di affidabilità** è un aspetto cruciale per garantire che un sistema di database mantenga l'integrità dei dati anche in caso di guasti hardware o software. L'obiettivo del controllo di affidabilità è quello di **ripristinare lo stato corretto del sistema** dopo un malfunzionamento, assicurando che le **proprietà ACID (Atomicità, Coerenza, Isolamento, Durabilità)** siano rispettate.

In particolare, il controllo di affidabilità garantisce che le transazioni siano **atomiche** (tutto o nulla), e che i dati siano **persistenti** anche in presenza di guasti. Se un guasto si verifica durante l'esecuzione di una transazione, il sistema deve essere in grado di ripristinare la base di dati nel suo stato precedente, come se il guasto non fosse mai accaduto.

Memorie nei DBMS

La gestione della memoria è fondamentale nel contesto del controllo di affidabilità. Le memorie coinvolte sono divise in tre principali tipologie:

1. **Memoria centrale (RAM)**: Questa è la memoria volatile, usata per la gestione immediata dei dati durante l'esecuzione delle transazioni. È molto veloce ma non persistente, il che significa che i dati possono andare persi in caso di guasto.
2. **Memoria di massa (dischi rigidi o SSD)**: Questo storage è persistente e offre una capacità di archiviazione molto maggiore rispetto alla memoria centrale. Tuttavia, la velocità di accesso ai dati è più lenta rispetto alla memoria centrale.

Gestore dell'Affidabilità

Il **Gestore dell'Affidabilità** è il componente del DBMS responsabile della gestione della persistenza dei dati, garantendo che tutte le modifiche delle transazioni completate siano memorizzate in modo permanente. Questo gestore è particolarmente importante per garantire che i dati rimangano coerenti dopo un guasto e che tutte le transazioni vengano ripristinate correttamente.

La principale difficoltà è che le operazioni di scrittura non sono sempre **atomiche**. In altre parole, se una transazione ha effettuato delle modifiche in memoria centrale ma non ha ancora scritto i dati sulla memoria di massa al momento di un guasto, quei cambiamenti potrebbero essere persi.

Concetti di Recovery

Il **recupero (recovery)** si basa su vari concetti chiave che aiutano a riportare il sistema nel suo stato consistente dopo un guasto. I principali concetti di recovery sono i seguenti:

File di Log: Il **log** è un file fondamentale per il recovery, poiché registra tutte le operazioni delle transazioni in ordine cronologico. È una sorta di "diario di bordo" che permette di ricostruire lo stato corretto del database dopo un guasto. Il log contiene dettagli come l'identificativo della transazione, l'operazione effettuata (es. lettura, scrittura), e l'oggetto del database coinvolto, oltre ai valori prima e dopo la modifica.

Checkpoint: Il **checkpoint** è un'operazione che viene effettuata per ottimizzare le operazioni di recovery. Durante un checkpoint, il sistema "congela" lo stato delle transazioni e trasferisce tutte le modifiche in memoria di massa, registrando le transazioni attive e sincronizzando il contenuto della base di dati con il log. Questo riduce la quantità di lavoro necessario durante il recovery, poiché il sistema sa esattamente quali transazioni erano in corso e quali sono già state completate.

Dump (Backup): Il **dump** è una copia del contenuto del database che viene archiviata su memoria stabile, di solito durante periodi di inattività del sistema. Questo backup è necessario per ricostruire i dati in caso di guasti gravi, come danni irreparabili alla memoria secondaria.

Dettagli sul Log di Sistema

Il log di sistema è il cuore del processo di recovery, e contiene diversi tipi di record. Ogni transazione che interagisce con il database è registrata nel log con i seguenti dettagli:

- **ID della transazione (T)**: Identificativo univoco della transazione.
- **Timestamp (TS)**: Il momento in cui l'operazione è stata eseguita.
- **Operazione (Op)**: Tipo di operazione effettuata dalla transazione (`begin, update, delete, insert, commit, abort`).
- **Oggetto (O)**: L'oggetto del database che è stato modificato.
- **Before-image (B_I)**: Il valore dell'oggetto prima della modifica da parte della transazione.
- **After-image (A_I)**: Il valore dell'oggetto dopo la modifica da parte della transazione.

Inoltre, il log include:

- **Record di Dump (DP)**: Indica il momento in cui è stato effettuato l'ultimo backup e fornisce dettagli sui file coinvolti.
- **Record di Checkpoint (CK)**: Contiene un elenco delle transazioni attive al momento del checkpoint, facilitando il processo di recovery.

Operazioni di Ripristino (Recovery)

Il processo di **ripristino** (recovery) ha come obiettivo riportare il database in uno stato consistente dopo un guasto, utilizzando le informazioni contenute nel log.

Undo e Redo

Le due operazioni principali per il recovery sono:

- **Undo** (disfare): Se un guasto si verifica prima che una transazione completi il suo **commit**, tutte le sue modifiche devono essere annullate (undo). Ciò significa che, attraverso il log, il sistema ripristina i dati allo stato precedente alla transazione.
- **Redo** (rifare): Se un guasto si verifica dopo che una transazione ha effettuato il commit ma prima che i dati siano scritti sulla memoria di massa, tutte le modifiche devono essere ripetute (redo). Il sistema ripristina i dati, applicando nuovamente le modifiche registrate nel log, per assicurarne la persistenza.

Ripristino dopo un Guasto

1. **Se il guasto avviene prima del commit**: Il sistema deve **annullare** tutte le modifiche delle transazioni che non sono state completate. Queste modifiche vengono ripristinate allo stato precedente all'inizio della transazione (operazione di **undo**).
2. **Se il guasto avviene dopo il commit**: Il sistema deve **rifare** tutte le operazioni che sono state committate ma non ancora scritte in memoria di massa (operazione di **redo**).

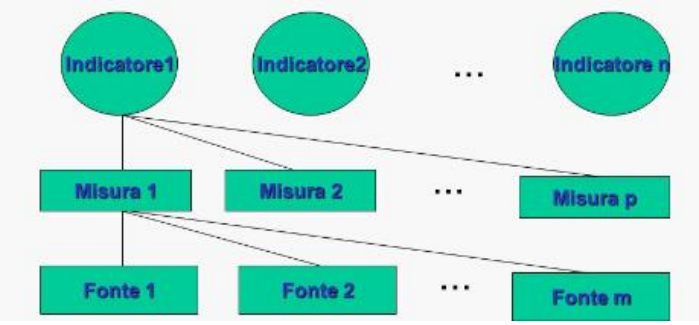
Operazioni di Checkpoint

Un checkpoint aiuta a ridurre il lavoro necessario durante il ripristino. Quando un checkpoint viene effettuato, il sistema assicura che tutte le transazioni che hanno effettuato il commit abbiano avuto i loro dati scritti sulla memoria di massa. Inoltre, registra le transazioni ancora attive al momento del checkpoint. In caso di guasto, il sistema può ripartire dal checkpoint più recente, limitando il numero di operazioni di **undo** e **redo**.

Le operazioni di **checkpoint** sono quindi fondamentali per garantire che, in caso di guasto, il sistema possa riprendersi rapidamente senza dover analizzare l'intero log.

Basi di Dati Direzionali

Sistemi Informativi Direzionali: Struttura e Funzioni



I sistemi informativi direzionali rappresentano una componente fondamentale nella gestione strategica di un'azienda moderna. Essi si pongono come supporto tecnologico e informativo alle decisioni aziendali, agevolando la direzione nel definire obiettivi, strategie e azioni da intraprendere per garantirne il successo e la sostenibilità a lungo termine. La complessità crescente del mercato e la necessità di adattamento alle variazioni economiche e tecnologiche richiedono una gestione informata e attenta dei processi, che è resa possibile da un sistema informativo adeguato e ben integrato nei livelli aziendali.

Livelli di Gestione: Direzione e Operatività

In un'azienda, le informazioni gestionali possono essere categorizzate e gestite su più livelli. Il livello direzionale è il punto di riferimento per la pianificazione a lungo termine e la definizione degli obiettivi aziendali. Questo livello comprende i dirigenti e i quadri che hanno il compito di analizzare dati complessi, interpretare le tendenze di mercato, anticipare le esigenze dei clienti e pianificare strategie per il futuro dell'azienda. Queste informazioni devono essere basate su dati attendibili e analisi precise, che permettano di valutare le opportunità e i rischi.

Al livello operativo, invece, l'attenzione è focalizzata sulle attività quotidiane che garantiscono la produzione dei beni o la fornitura dei servizi dell'azienda. Gli operatori a questo livello svolgono compiti specifici e spesso ripetitivi, seguendo le linee guida stabilite dai dirigenti, con un focus principale sull'efficienza e sulla qualità del prodotto finale. Le attività operative generano un flusso continuo di dati riguardanti l'andamento produttivo, le vendite, le scorte di magazzino, i tempi di lavorazione e la qualità, che vengono poi inviati al livello direzionale sotto forma di reportistica.

Tipologia delle Informazioni Direzionali

I sistemi informativi direzionali operano elaborando informazioni ad alto livello, ovvero fortemente aggregate, che vengono sintetizzate per fornire ai dirigenti una visione chiara e facilmente interpretabile dello stato dell'azienda. Queste informazioni, raccolte e organizzate da vari sottosistemi operativi e gestionali, vengono trasformate in dati significativi chiamati "indicatori prestazionali". Gli indicatori prestazionali forniscono una rappresentazione quantitativa dei risultati raggiunti, permettendo alla direzione di valutare sia il successo delle strategie intraprese sia l'andamento complessivo dell'azienda.

La Natura Sintetica delle Informazioni Direzionali

Un sistema informativo direzionale deve selezionare e aggregare i dati più rilevanti per consentire ai dirigenti di visualizzare in pochi parametri gli aspetti cruciali della gestione aziendale. A differenza delle informazioni operative, che possono comprendere una vasta gamma di dettagli puntuali (es. transazioni giornaliere, livelli di produzione, e tempi di ciclo), le informazioni direzionali devono ridurre questi dettagli a sintesi utili alla valutazione globale delle prestazioni. Ad esempio, invece di mostrare il numero esatto di prodotti venduti per ciascun punto vendita, il sistema può sintetizzare tali dati in metriche aggregate come il volume di vendite mensile, il ricavo totale per area geografica o la percentuale di crescita rispetto all'anno precedente.

Il Paradigma "Indicatori – Misure – Fonti" nei Sistemi Informativi Direzionali

Il paradigma "Indicatori – Misure – Fonti" è un modello strutturale utilizzato per organizzare le informazioni necessarie alla direzione aziendale in modo chiaro e funzionale. Questo approccio si fonda su tre elementi principali: *gli indicatori*, che rappresentano le metriche chiave per valutare le performance aziendali; *le misure*, che quantificano questi indicatori in valori specifici; e *le fonti*, ossia i dati e i sistemi che generano le informazioni necessarie.

1. Indicatori

Gli indicatori sono variabili strategiche selezionate per monitorare l'andamento aziendale rispetto agli obiettivi stabiliti. Essi rappresentano i parametri essenziali che la direzione deve monitorare per comprendere se l'azienda sta progredendo nella direzione desiderata e se le strategie intraprese sono efficaci.

2. Misure

Le misure sono i valori numerici attribuiti agli indicatori e rappresentano la quantificazione specifica degli obiettivi di performance. Se l'indicatore è il parametro da monitorare, la misura è il dato numerico che quantifica questo parametro.

3. Fonti

Le fonti rappresentano i dati grezzi e i sistemi informativi dai quali vengono estratti gli indicatori e le misure. Possono includere sistemi di gestione delle risorse aziendali (ERP), database, reportistica, sondaggi di soddisfazione, e persino dati raccolti tramite strumenti IoT nei casi in cui si monitorano attività operative automatizzate.

Le fonti sono suddivisibili in:

- **Fonti interne**: Dati generati direttamente dalle operazioni aziendali, come i report di vendita, i dati di produzione e il feedback raccolto dai clienti tramite sistemi di CRM.
- **Fonti esterne**: Dati provenienti dall'esterno dell'azienda, come le analisi di mercato, i benchmark di settore, i dati economici globali e altre informazioni rilevanti per le decisioni strategiche.

Dimensioni (Variabili) dell'Analisi nelle Informazioni Direzionali

Per una gestione efficace e informata, l'analisi delle informazioni direzionali in un'azienda deve considerare più dimensioni, ognuna delle quali offre una prospettiva specifica sulle prestazioni e sul raggiungimento degli obiettivi. Le dimensioni principali dell'analisi direzionale includono il tempo, il prodotto, i processi, la responsabilità e il cliente. Queste dimensioni costituiscono le variabili chiave attraverso cui è possibile ottenere una visione completa e dettagliata dell'andamento aziendale.

1. Dimensione Tempo

La dimensione temporale consente di valutare l'andamento dell'azienda lungo un arco temporale specifico, come giorni, mesi, trimestri o anni. Questa dimensione è fondamentale perché permette alla direzione di monitorare l'evoluzione delle performance e di identificare trend o cicli. Ad esempio, analizzando i ricavi e le vendite su base trimestrale, la direzione può osservare come le strategie adottate in un dato periodo abbiano influito sui risultati complessivi e pianificare interventi in risposta a variazioni stagionali o cambiamenti di mercato.

2. Dimensione Prodotto

La dimensione prodotto è cruciale per valutare costi e ricavi associati a ciascun prodotto o servizio offerto. Si tratta di un'analisi finalizzata non solo a monitorare la redditività di ciascun prodotto, ma anche a comprendere quali linee di prodotti contribuiscono maggiormente al fatturato e quali, eventualmente, devono essere rinnovate o potenziate. In termini contabili, questa dimensione si basa su indicatori di tipo monetario (ad esempio, margini di profitto, costi unitari, incidenza sul totale delle vendite) che misurano il valore economico generato da ciascun prodotto.

3. Dimensione Processi

La dimensione processi si concentra sull'efficienza e sull'efficacia delle attività interne. È volta a monitorare le performance operative, valutando, ad esempio, la tempestività nella produzione e nella consegna, l'efficienza dei cicli produttivi e il rispetto dei tempi di progetto. I parametri di questa dimensione includono misure di qualità, tempi di ciclo, e tassi di errore, che consentono alla direzione di identificare i punti di forza e le aree di miglioramento nei processi aziendali. Analizzando i processi, è possibile apportare modifiche che migliorino la produttività e riducano gli sprechi.

4. Dimensione Responsabilità

La dimensione responsabilità è fondamentale per valutare le prestazioni dei singoli dirigenti e dei rispettivi centri di responsabilità, che possono essere dipartimenti, unità di business o specifici team all'interno dell'azienda. Ogni centro di responsabilità è associato a una serie di indici di performance che riflettono il contributo specifico alle performance aziendali, come l'efficienza del dipartimento, la gestione del budget, e il raggiungimento degli obiettivi. Questa

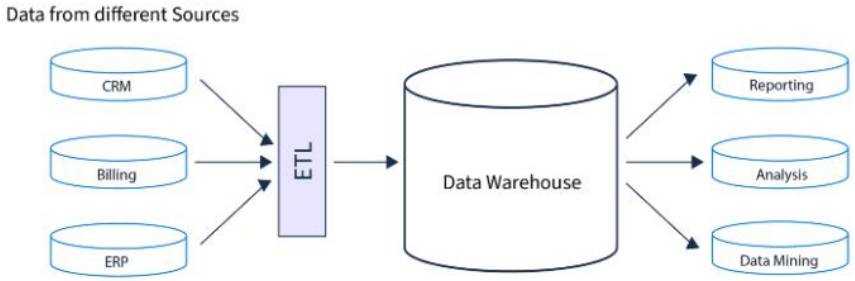
dimensione permette di assegnare in maniera chiara le responsabilità dei risultati ottenuti e di incentivare il miglioramento continuo.

5. Dimensione Cliente

La dimensione cliente è fondamentale per comprendere la redditività e il volume di affari generati dai diversi segmenti di clientela. Questa dimensione include l'analisi della soddisfazione del cliente, la fidelizzazione, e la redditività di ciascun cliente o gruppo di clienti. Esaminare questa dimensione permette alla direzione di capire quali segmenti sono più profittevoli, quali necessitano di strategie di fidelizzazione, e dove possono essere implementati miglioramenti nei servizi. Inoltre, questa dimensione aiuta a valutare l'efficacia delle strategie di marketing e le risposte della clientela ai prodotti e ai servizi offerti.

Architettura dei Sistemi Informativi Direzionali (SID)

L'architettura dei Sistemi Informativi Direzionali (SID) si basa su una struttura suddivisa in due principali sottosistemi: *front-end* e *back-end*, supportati da una base dati direzionale comunemente nota come *Data Warehouse*. Questa architettura consente di raccogliere, organizzare, e presentare in modo efficace le informazioni necessarie per le decisioni strategiche aziendali.



1. Data Warehouse: La Base Dati Direzionale

Il *Data Warehouse* è il cuore del Sistema Informativo Direzionale, una base dati direzionale in cui vengono raccolte e centralizzate le informazioni provenienti da diverse fonti aziendali. Queste informazioni vengono estratte dai sistemi operativi, riorganizzate e integrate per fornire una visione globale e aggiornata delle operazioni aziendali, utile per supportare le decisioni della direzione. Il Data Warehouse permette di mantenere una *storicità* dei dati, accumulando informazioni storiche utili per analisi di lungo termine e trend storici.

Orientato al Soggetto

Il DWH è progettato per essere orientato ai soggetti dell'elaborazione, il che significa che i dati sono organizzati attorno a entità specifiche rilevanti per l'analisi direzionale, come ad esempio clienti, prodotti, o regioni geografiche. Questo approccio si differenzia dalle tradizionali applicazioni di database, in cui i dati sono orientati principalmente ai processi o alle funzioni operative.

Essere orientato al soggetto implica che le informazioni sono aggregate e strutturate secondo le dimensioni di interesse strategico, facilitando la comprensione del contesto e l'individuazione di tendenze e anomalie nei settori chiave.

Integrato

Un elemento fondamentale del DWH è la sua natura *integrata*. Le informazioni provengono da diverse basi dati aziendali (spesso eterogenee) e quindi necessitano di essere armonizzate per consentire analisi coerenti. Per raggiungere questa coerenza, il DWH unifica i dati tramite vari meccanismi di integrazione, tra cui:

- **Misure consistenti delle variabili**: Ogni dato viene riportato a un'unità di misura e formato standard, eliminando le discrepanze tra le varie fonti.
- **Attributi fisici uniformi**: Gli attributi dei dati (come le date o i codici prodotto) seguono convenzioni standard per semplificare la loro interpretazione.

- **Strutture di codifica**: Vengono unificate le codifiche, come quelle relative ai prodotti o ai clienti, per evitare ambiguità nell'analisi.
- **Convenzioni sui nomi**: I nomi delle variabili vengono standardizzati per facilitare la comprensione e l'utilizzo da parte degli utenti.

Questa integrazione garantisce che tutte le informazioni siano consistenti e pronte per essere utilizzate nelle analisi, indipendentemente dalla fonte di origine, consentendo così una visione aziendale coerente.

Tempo-Variante

A differenza dei database operativi, in cui i dati riflettono solo lo stato corrente (valido per pochi giorni o settimane), il Data Warehouse è *tempo-variante*, ossia mantiene dati storici che coprono periodi estesi, spesso di anni. La conservazione di dati storici permette di:

- Analizzare trend e cambiamenti nel tempo, essenziali per la previsione e la pianificazione.
- Effettuare confronti tra periodi differenti per valutare l'efficacia delle strategie aziendali.

Ogni dato nel DWH è accompagnato da un riferimento temporale, che può essere esplicito (come una data o un periodo) o implicito (incluso nella struttura stessa dei dati). Questo riferimento temporale è essenziale per l'analisi strategica, poiché permette di osservare l'evoluzione delle variabili aziendali, come le vendite o i costi, e fornisce la base per un'analisi multidimensionale che utilizza il tempo come una dimensione fondamentale.

Non Volatile

Una caratteristica distintiva del Data Warehouse è la sua *non-volatilità*. I dati in un DWH non vengono aggiornati, modificati o cancellati una volta inseriti. Al contrario, i dati rappresentano delle istantanee o "foto" storiche dei processi aziendali che possono essere caricate e successivamente accedute per l'analisi.

Questa non-volatilità implica che il DWH si limita ad accumulare progressivamente informazioni, evitando modifiche retroattive che potrebbero alterare la coerenza storica dei dati. Il vantaggio di questa caratteristica è che le analisi basate su DWH possono essere ripetibili e comparabili nel tempo, poiché i dati restano invariati rispetto al momento in cui sono stati acquisiti.

2. Sottosistema Back-End

Il *back-end* del Sistema Informativo Direzionale è responsabile dell'alimentazione continua del Data Warehouse. Questo sottosistema ha il compito di raccogliere periodicamente i dati da diverse fonti aziendali, quali sistemi transazionali, database operativi e altre piattaforme. Le informazioni sono quindi estratte, trasformate e caricate (ETL) nel Data Warehouse, dove possono essere analizzate e utilizzate dal sistema direzionale.

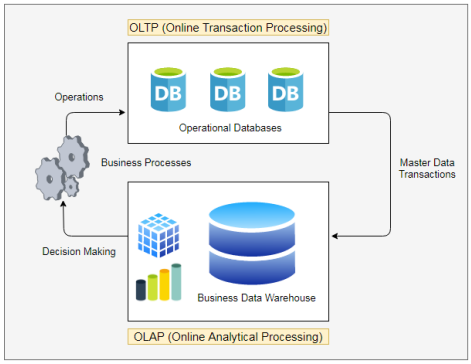
3. Sottosistema Front-End

Il *front-end* è l'interfaccia con cui gli utenti finali (in particolare i dirigenti e i decisori aziendali) interagiscono con il Data Warehouse. Questo sottosistema comprende gli strumenti e le applicazioni di *Business Intelligence* (BI), come dashboard, report, e strumenti di analisi interattiva, che rendono possibile visualizzare e interpretare i dati in modo efficace.

Funzionamento Integrato dei Sottosistemi

L'interazione tra back-end, Data Warehouse e front-end costituisce un ciclo continuo e automatizzato, garantendo che il sistema direzionale riceva informazioni aggiornate e accessibili. Il back-end alimenta periodicamente il Data Warehouse, mantenendo aggiornata la base dati aziendale e preparando le informazioni per le analisi. Il front-end, a sua volta, consente agli utenti di accedere ai dati elaborati, favorendo un processo decisionale basato su una conoscenza accurata e completa dello stato aziendale.

Sistemi Operativi OLTP e Sistemi Direzionali OLAP



I sistemi informativi aziendali si basano su due tipologie fondamentali di sistemi: i sistemi *OLTP* (Online Transaction Processing) e i sistemi *OLAP* (Online Analytical Processing). Questi due tipi di sistemi rispondono a esigenze diverse e complementari all'interno dell'azienda. Mentre gli OLTP supportano le operazioni quotidiane e transazionali, gli OLAP sono progettati per analizzare i dati e aiutare nelle decisioni strategiche. Di seguito vengono descritte le caratteristiche e le differenze principali tra i due sistemi.

Sistemi OLTP (Online Transaction Processing)

I sistemi OLTP sono i sistemi operativi che gestiscono le operazioni e le transazioni quotidiane dell'azienda. Questi sistemi sono utilizzati per registrare, elaborare e memorizzare grandi volumi di transazioni in tempo reale. Gli OLTP costituiscono il "motore" delle operazioni aziendali, poiché permettono di gestire attività come le vendite, la contabilità, la gestione dei clienti e le risorse umane.

Sistemi OLAP (Online Analytical Processing)

I sistemi OLAP, invece, sono progettati per analizzare i dati e supportare il processo decisionale a livello direzionale. Si basano su un approccio di tipo multidimensionale, in cui i dati vengono organizzati per consentire analisi complesse e dettagliate su diverse dimensioni, come il tempo, i prodotti, le regioni o i clienti. L'obiettivo principale degli OLAP è fornire informazioni aggregate, storiche e sintetiche, utili per le decisioni strategiche e la pianificazione aziendale.

Caratteristica	OLTP	OLAP
Funzione Principale	Supporto alle operazioni quotidiane	Supporto alle decisioni strategiche
Orientamento	Transazioni	Analisi multidimensionale
Tipo di Dati	Dati operativi e in tempo reale	Dati storici e consolidati
Tempo di Risposta	Molto rapido, a bassa latenza	Generalmente più lento
Frequenza Aggiornamenti	Continua (in tempo reale)	Periodica (giornaliera o settimanale)
Accesso ai Dati	Lettura e scrittura frequente	Solo lettura
Struttura Dati	Normalizzata per minimizzare la ridondanza	Denormalizzata per migliorare le prestazioni analitiche
Esempi di Utilizzo	Sistemi POS, gestione ordini, CRM	BI, dashboard, reportistica, analisi KPI

Data Sources, ETL, Staging e Architettura del Data Warehouse: Aspetti Chiave

Un Data Warehouse (DWH) è una struttura complessa che permette di centralizzare e organizzare i dati provenienti da fonti differenti per renderli facilmente accessibili e utili all'analisi aziendale. Questo processo coinvolge una serie di fasi e componenti fondamentali, tra cui la raccolta e l'integrazione dei dati, il processo di ETL, la fase di staging, l'integrazione centralizzata, i metadati e l'utilizzo di *data marts* per le esigenze dipartimentali. Di seguito, vengono esaminate queste componenti chiave del DWH.

Data Sources (Sorgenti Dati)

Le sorgenti di dati del DWH sono eterogenee e provengono sia da fonti interne che esterne all'azienda. Le principali tipologie di sorgenti includono:

1. **Sistemi operazionali**: Si tratta dei database utilizzati per le operazioni quotidiane, spesso basati su database relazionali che gestiscono transazioni in tempo reale (OLTP).

2. **Applicazioni "legacy"**: Sistemi aziendali esistenti, spesso basati su tecnologie obsolete, che possono non rispettare i requisiti moderni. Nonostante ciò, contengono dati preziosi per il business.
3. **Sistemi informativi esterni**: Fonti esterne strutturate, come database di settore, oppure **flat files** (file piatti), che rappresentano dati non organizzati in modo relazionale, come file di testo o file CSV.

ETL (Extraction, Transformation, and Loading)

Il processo di ETL è fondamentale per raccogliere e rendere coerenti i dati provenienti dalle varie sorgenti. L'ETL si articola in tre fasi principali:

- **Estrazione (Extraction)**: I dati vengono estratti dalle varie sorgenti, che possono avere formati diversi e livelli di qualità variabili.
- **Trasformazione (Transformation)**: I dati estratti vengono puliti per eliminare incongruenze e incoerenze, completati per compensare eventuali dati mancanti e armonizzati secondo uno schema comune. Questa fase richiede spesso l'applicazione di regole specifiche per integrare le informazioni secondo le esigenze aziendali.
- **Caricamento (Loading)**: I dati trattati vengono caricati nel Data Warehouse per poter essere successivamente analizzati.

Gli strumenti ETL sono essenziali per automatizzare questi processi, permettendo di mantenere il DWH aggiornato e consistente.

Staging

Prima di essere definitivamente caricati nel DWH, i dati passano attraverso una fase detta *staging*. La staging area è uno spazio temporaneo in cui i dati integrati, corretti e validati vengono immagazzinati come dati "riconciliati" per:

- Costituire un modello di riferimento comune per l'azienda.
- Essere pronti per l'inserimento nel Data Warehouse centrale.

La staging area è un ambiente intermedio che consente di isolare le operazioni di elaborazione e trasformazione dei dati dalla base dati principale del DWH, migliorando la coerenza dei dati e riducendo l'impatto sui sistemi operazionali.

Integrazione del Data Warehouse

Il Data Warehouse è pensato come un unico "contenitore" logicamente centralizzato, progettato per raccogliere tutte le informazioni necessarie per l'analisi strategica. Il DWH ospita anche i *metadati*, che sono informazioni aggiuntive sui dati stessi (i cosiddetti "dati sui dati"). I metadati sono cruciali per la gestione e la navigazione del DWH, fornendo:

- **Identificazione e descrizione dei dati**: Informazioni sui dati contenuti nel DWH, come la loro provenienza e significato.
- **Gestione del caricamento**: Dettagli sulle operazioni di caricamento, incluse le sorgenti dati e le trasformazioni eseguite.
- **Ottimizzazione delle query**: Informazioni sulle query utilizzate più frequentemente, utili per migliorare le prestazioni e ottimizzare l'accesso ai dati.
- **Struttura del DWH**: Informazioni su tabelle, viste e indici, essenziali per la gestione dei dati.

Data Marts

I *data marts* sono sottoinsiemi di un DWH progettati per rispondere alle esigenze specifiche di singole aree di business o dipartimenti, come il reparto vendite o la produzione. Essendo più piccoli e mirati rispetto al DWH centrale, i data marts risultano più gestibili, più efficienti e consentono un accesso più rapido alle informazioni pertinenti per un determinato gruppo di utenti.

Caratteristiche principali dei data marts:

- **Focalizzazione dipartimentale**: Sono progettati per soddisfare i requisiti di uno specifico dipartimento o funzione aziendale.

- **Dati aggregati**: Contengono dati ad alto livello di aggregazione, limitando la quantità di informazioni per semplificare l'analisi.
- **Facile navigazione**: Grazie alla loro dimensione ridotta e alla specializzazione, i data marts sono più rapidi da esplorare e più intuitivi.

Acquisizione e Gestione dei Dati nel Data Warehouse: Flussi e Processi Principali

La gestione e l'organizzazione dei dati in un Data Warehouse (DWH) richiedono una serie di processi che ne permettono l'estrazione, l'integrazione, la conservazione, la messa a disposizione degli utenti finali e la gestione dei metadati. Questi flussi operativi vengono definiti con i termini **inflow**, **upflow**, **downflow**, **outflow** e **metaflow**, ciascuno con compiti specifici nell'assicurare che i dati siano coerenti, sicuri e pronti per essere utilizzati nelle analisi aziendali. Vediamo in dettaglio ciascuno di questi flussi.

Inflow: L'ingresso dei dati nel Data Warehouse

Il processo di inflow rappresenta il punto di ingresso dei dati nel DWH, ed è un momento essenziale che permette di tradurre le informazioni provenienti da vari sistemi operazionali (come quelli OLTP) in dati analizzabili. In questa fase, i dati grezzi vengono estratti dai sistemi di produzione e vengono sottoposti a una serie di trattamenti per adattarli alle finalità analitiche del DWH. Spesso, i dati provenienti dai sistemi operazionali sono progettati per soddisfare esigenze di gestione transazionale e non sono immediatamente pronti per l'analisi aggregata o storica. Per questo motivo, i dati devono essere:

1. **Riorganizzati e normalizzati**: L'ordine e la struttura dei dati vengono adattati per meglio rispondere agli scopi del DWH. Questo processo può comportare sia la rimozione di campi non necessari sia l'aggiunta di nuove informazioni che consentano una lettura più completa e storica.
2. **Filtrati e validati**: Nella fase di inflow, la coerenza è un obiettivo primario. I dati estratti vengono confrontati con quelli già presenti nel DWH per identificare e risolvere possibili incongruenze, come duplicati o anomalie. Questa validazione consente di preservare l'integrità dei dati lungo l'intero processo di caricamento.
3. **Conservati in un'area di staging**: Prima di essere integrati nel DWH, i dati vengono trasferiti in un'area temporanea, detta staging area, dove sono sottoposti a ulteriori test e a operazioni di trasformazione. Questo passaggio intermedio garantisce che solo i dati conformi e consistenti vengano effettivamente trasferiti nel DWH, proteggendo il sistema principale da eventuali errori o inconsistenze.

Upflow: Aggregazione dei dati per l'analisi

Nel flusso di upflow, i dati, ormai validati e corretti, vengono aggregati per renderli fruibili in diverse forme e livelli di dettaglio. Questo processo di aggregazione è pensato per trasformare i dati operativi in informazioni sintetiche che risultino significative per l'analisi direzionale.

1. **Aggregazione graduale**: L'upflow permette di ottenere una visione graduale dei dati, che possono essere visualizzati a diversi livelli di sintesi: da report molto dettagliati, che si concentrano su singole transazioni, a dati aggregati che mostrano solo le informazioni globali e strategiche. Questo rende possibile un uso flessibile del DWH, che può così rispondere a varie esigenze informative, dai dettagli operativi alle visioni complessive.
2. **Supporto alle decisioni**: La presenza di dati aggregati agevola l'accesso alle informazioni sintetiche, permettendo ai dirigenti di monitorare rapidamente l'andamento aziendale, come il volume di vendite o i ricavi complessivi. Gli utenti possono quindi attingere a informazioni che guidano e supportano le decisioni strategiche con la giusta profondità.

Downflow: Sicurezza e conservazione dei dati

Un aspetto spesso trascurato nella gestione dei dati è quello della loro conservazione a lungo termine. Il flusso di downflow si occupa proprio di questo: proteggere i dati da perdite accidentali e garantirne la disponibilità anche in caso di guasti tecnici. Un DWH è infatti una fonte di informazioni storiche, e la perdita di dati può significare la perdita di anni di storia aziendale, con un impatto significativo sulle analisi.

1. **Backup e recupero**: Con il downflow, vengono create copie di sicurezza periodiche dei dati, essenziali per la continuità aziendale. In caso di guasto, queste copie permettono di ripristinare lo stato del DWH, assicurando che i dati storici siano sempre accessibili e coerenti.

2. **Archivio storico**: Grazie al downflow, il DWH può rappresentare una vera e propria memoria storica dell'azienda, con dati organizzati e protetti per il lungo periodo. Questo consente di effettuare confronti intertemporali, valutare tendenze e misurare l'impatto di strategie passate.

Outflow: L'accesso degli utenti finali

Se il downflow garantisce la sicurezza dei dati, il flusso di outflow ne permette l'accesso e la fruibilità da parte degli utenti finali. Il DWH, infatti, deve poter rispondere velocemente alle richieste, fornendo dati in modo rapido ed efficiente per supportare analisi immediate e reportistica.

1. **Interrogazioni e query ottimizzate**: I dati contenuti nel DWH sono accessibili agli utenti attraverso query ottimizzate, progettate per rispondere in modo rapido e preciso. In questo modo, i dirigenti aziendali possono ottenere informazioni aggiornate senza rallentamenti o difficoltà di accesso.
2. **Soddisfacimento delle esigenze informative**: Il flusso di outflow è pensato per agevolare l'interazione con gli utenti finali, facilitando l'accesso a informazioni sintetiche e specifiche, necessarie per il monitoraggio delle performance e la pianificazione strategica.

Metaflow: Gestione e organizzazione dei metadati

Oltre ai dati principali, il DWH include i metadati, che forniscono una descrizione dettagliata delle informazioni contenute e del loro uso. I metadati, gestiti attraverso il metaflow, sono dati sui dati e svolgono una funzione chiave nell'organizzazione del DWH.

1. **Descrizione e catalogazione dei dati**: I metadati forniscono una mappatura delle informazioni contenute nel DWH, descrivendo per ciascun campo il suo significato, la fonte, l'ultimo aggiornamento e le caratteristiche strutturali. Questa documentazione è essenziale per facilitare la comprensione e la navigazione del DWH da parte degli utenti finali.
2. **Ottimizzazione delle performance**: Grazie ai metadati, è possibile migliorare le prestazioni del DWH, ad esempio monitorando e ottimizzando le query utilizzate più frequentemente dagli utenti. I metadati forniscono anche indicazioni utili sulle esigenze informative dei diversi dipartimenti, permettendo di adeguare il sistema alle necessità reali.

Querying e Reporting

Il Querying & Reporting è uno degli strumenti più basilari per estrarre informazioni dai dati presenti nella data warehouse. Consiste in un processo strutturato che comprende vari passaggi, finalizzati alla trasformazione e presentazione dei dati in un formato comprensibile e utile per l'utente finale. Questo processo include le seguenti fasi:

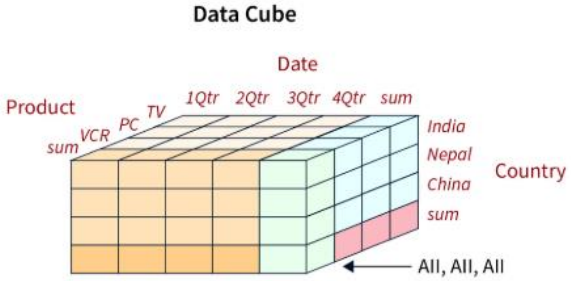
1. **Interrogazione dei dati**: In primo luogo, si pone una domanda specifica alla data warehouse, con l'obiettivo di estrarre dati rilevanti per il contesto analitico di interesse. Questa interrogazione è formulata tramite query specifiche, generalmente in linguaggio SQL, per selezionare i dati che rispondono al bisogno informativo.
2. **Rilevazione e trasformazione dei dati**: Una volta estratti, i dati devono essere trasformati in un formato appropriato, che renda chiaro il contesto e consenta un'interpretazione rapida e precisa.
3. **Preparazione e consegna dei dati**: I dati trasformati vengono preparati in un formato leggibile, spesso in forma tabellare o grafica, per facilitarne la consultazione e l'interpretazione da parte dell'utente finale. Questo consente una visione chiara delle informazioni, utile per generare report accurati e facilmente interpretabili.

Analisi Multidimensionale dei Dati

L'analisi multidimensionale dei dati è una metodologia fondamentale nell'ambito del **Data Warehousing**, che consente di esaminare grandi volumi di dati da molteplici prospettive, comunemente definite "dimensioni". Questo approccio strutturato aiuta le organizzazioni a ottenere insight approfonditi e a prendere decisioni informate basandosi su una rappresentazione organica e aggregata delle informazioni.

Struttura dei Cubi Multidimensionali

Al centro dell'analisi multidimensionale si trovano i **cubi multidimensionali**, che organizzano i dati in un formato che facilita l'esplorazione interattiva. Ogni cubo è definito da:



- **Dimensioni**: Rappresentano i diversi punti di vista dell'analisi. Le dimensioni descrivono le caratteristiche degli eventi o dei fenomeni analizzati, come il tempo, il luogo o il prodotto.
- **Misure**: Sono gli indicatori quantitativi di interesse, come il totale delle vendite, il numero di unità vendute o il margine di profitto.
- **Gerarchie**: All'interno di ciascuna dimensione, esistono livelli di dettaglio organizzati in una struttura gerarchica. Ad esempio, nella dimensione tempo si possono distinguere i livelli: giorno, mese, trimestre e anno. Queste gerarchie permettono di esplorare i dati con livelli di granularità differenti.

Questa organizzazione consente di rappresentare i dati in una forma intuitiva e navigabile, dove ogni cella del cubo rappresenta una combinazione specifica di dimensioni e misure.

Operazioni Fondamentali nell'Analisi Multidimensionale

Le operazioni che possono essere effettuate sui cubi multidimensionali permettono una grande flessibilità nell'analisi dei dati. Tra le principali:

1. **Slice & Dice**

   L'operazione di *slice* consiste nel selezionare una porzione del cubo multidimensionale fissando un valore specifico per una o più dimensioni. Ad esempio, si potrebbe isolare il cubo delle vendite relative a un particolare anno.

   Il *dice*, invece, permette di selezionare un sottoinsieme dei dati che soddisfa criteri su più dimensioni. Ad esempio, analizzare solo i dati delle vendite per una specifica regione e categoria di prodotto.

2. **Roll-Up**

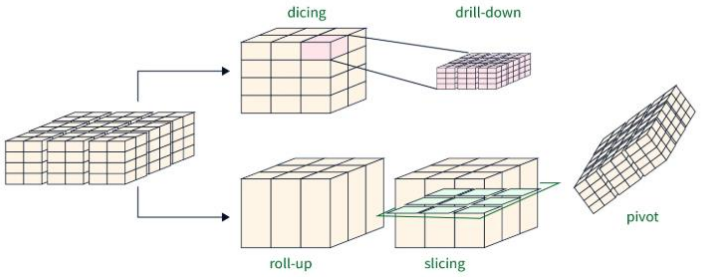
   Il *roll-up* è un'operazione di aggregazione che consente di salire nella gerarchia di una dimensione, passando da un livello di dettaglio più specifico a uno più generale. Ad esempio, aggregare i dati delle vendite giornaliere per ottenere le vendite mensili o annuali.

3. **Drill-Down**

   Il *drill-down* è l'operazione inversa del roll-up, che permette di esplorare i dati a livelli di dettaglio sempre più granulari. Ad esempio, si possono analizzare le vendite settimanali in un dato mese, scendendo fino alle vendite giornaliere o persino orarie.

4. **Pivoting**

Il *pivoting* consente di riorientare il cubo multidimensionale, modificando l'ordine delle dimensioni per osservare i dati da diverse prospettive. Questa operazione è particolarmente utile per individuare schemi nascosti o per riorganizzare i dati in modo che rispondano a specifiche domande di analisi.



Knowledge Discovery in Database (KDD)

Il **Knowledge Discovery in Database** (KDD) è un processo complesso e articolato che permette di estrarre valore dai dati tramite l'identificazione di schemi nascosti, modelli e regole. Questo processo va oltre la semplice analisi, combinando diversi strumenti e tecniche di *data mining* per costruire conoscenze operative, spesso difficili da individuare con metodi tradizionali.

Il KDD si compone di diverse fasi, ciascuna delle quali gioca un ruolo fondamentale per la qualità e la validità dei risultati ottenuti. Vediamo in dettaglio ogni fase e le considerazioni necessarie per ottimizzare il processo.

1. Identificazione del Problema

L'identificazione del problema rappresenta il punto di partenza e uno dei passaggi più critici nel KDD. In questa fase, si definisce il contesto dell'analisi e si chiariscono gli obiettivi specifici. È fondamentale che chi conduce l'analisi comprenda a fondo il contesto aziendale e abbia accesso a tutte le informazioni rilevanti sul problema da risolvere. La chiarezza degli obiettivi aiuta a orientare tutto il processo successivo, evitando sprechi di risorse e aumentando la precisione delle conclusioni.

Per esempio, un'azienda che desidera ridurre il tasso di abbandono dei clienti deve focalizzare l'analisi sulle variabili che influenzano la fidelizzazione, come la frequenza di acquisto, il tempo trascorso dall'ultimo acquisto o le caratteristiche del servizio clienti. L'analista, in collaborazione con i team aziendali, definirà quindi domande specifiche come "Quali clienti sono a rischio di abbandono?" o "Quali fattori incidono maggiormente sul customer churn?".

2. Selezione dei Dati

Dopo aver definito il problema, si procede alla scelta dei dati che saranno oggetto di analisi. Questa fase è cruciale poiché una selezione accurata permette di ottenere risultati più affidabili e significativi. La selezione dei dati implica un'analisi delle fonti disponibili, che possono includere database interni, registri transazionali, dati di vendita, o anche informazioni da fonti esterne.

È importante considerare sia la qualità che la quantità dei dati selezionati. Includere troppi dati non pertinenti potrebbe rallentare l'analisi e produrre rumore che rende più difficile individuare informazioni utili. Al contrario, dati insufficienti o parziali possono portare a conclusioni limitate o distorte. Ad esempio, se l'obiettivo è analizzare il comportamento dei clienti, è necessario raccogliere dati su acquisti, interazioni con il servizio clienti, navigazione online e demografia.

3. Pulizia e Normalizzazione dei Dati

La fase di pulizia e normalizzazione è una delle più laboriose ma fondamentali, in quanto la qualità dei dati influisce direttamente sull'accuratezza dei risultati. Questo processo è finalizzato alla rimozione di dati errati, incompleti, rumorosi o che presentano anomalie.

- **Rimozione di dati errati**: Qui si eliminano le informazioni palesemente scorrette, come valori negativi in campi di prezzo o date incoerenti.

- **Gestione dei valori mancanti**: Ci sono diversi approcci per trattare i dati mancanti, tra cui l'eliminazione di tali record (quando la perdita di informazioni è accettabile), l'uso di valori medi o predittivi per riempire i campi vuoti, o la creazione di categorie apposite per segnalare l'assenza di valori.
- **Eliminazione dei valori estremi**: I valori anomali, o *outliers*, possono distorcere l'analisi. È importante stabilire se tali valori rappresentano errori o fenomeni significativi prima di eliminarli. Ad esempio, un valore di vendita estremamente alto in un database di vendite potrebbe rappresentare un errore oppure una vendita particolarmente rilevante.
- **Normalizzazione**: Infine, la normalizzazione consiste nel rendere omogenee le scale dei dati, ad esempio convertendo i formati di data in un unico standard o utilizzando unità di misura coerenti.

4. Individuazione delle Caratteristiche Rilevanti (Feature Selection)

Questa fase è cruciale per rappresentare in modo efficace il fenomeno analizzato. L'individuazione delle caratteristiche rilevanti, o *feature selection*, consiste nella selezione delle variabili che meglio rappresentano il problema e l'obiettivo dell'analisi. Una selezione attenta delle variabili permette di migliorare la qualità del modello, ridurre il rumore e aumentare la velocità di elaborazione, mantenendo solo gli attributi che influenzano realmente l'output.

Ad esempio, in un'analisi sul comportamento d'acquisto, le variabili come il prezzo medio d'acquisto, la frequenza degli acquisti, il tipo di prodotti acquistati e il periodo dell'anno possono rappresentare meglio il comportamento del cliente rispetto ad altre informazioni meno rilevanti come l'orario di acquisto.

La selezione delle caratteristiche può essere eseguita tramite metodi statistici (come l'analisi delle correlazioni), tecniche di riduzione della dimensionalità (come l'analisi delle componenti principali, o PCA), o con algoritmi di machine learning supervisionato, che individuano automaticamente le variabili più significative.

5. Interpretazione e Valutazione dei Risultati

La fase finale del processo di KDD è dedicata all'interpretazione e valutazione dei risultati. È fondamentale che i modelli e le conoscenze estratte siano comprensibili e applicabili, ossia che abbiano una chiara utilità per l'obiettivo aziendale.

L'interpretazione dei risultati prevede l'analisi degli output del data mining, convalidando i risultati attraverso metodi statistici o confronti con dati di test. Ad esempio, se un modello di classificazione ha individuato un gruppo di clienti a rischio di abbandono, si dovrà valutare la correttezza del modello controllando le previsioni con dati storici o applicando il modello su un set di dati di test.

La valutazione si conclude con un confronto tra i risultati e gli obiettivi iniziali, verificando che le informazioni estratte siano effettivamente utili per il processo decisionale. In caso contrario, si può rivedere il processo o iterare alcune fasi per migliorare ulteriormente i risultati.

Modelli di Data Warehouse: Schema Relazionale a Stella

Lo **schema a stella** è uno dei modelli di organizzazione dati più utilizzati nei data warehouse, grazie alla sua struttura semplice e intuitiva, ideale per facilitare l'analisi multidimensionale. Questo modello è molto adatto per applicazioni di tipo OLAP (Online Analytical Processing), poiché permette di esplorare i dati attraverso molteplici dimensioni (come tempo, prodotto o luogo) e di ottenere aggregazioni utili per il business in modo rapido e coerente.

La Struttura dello Schema a Stella

Nel cuore dello schema a stella troviamo la **tabella dei fatti**, che contiene i dati principali su eventi o transazioni, ovvero ciò che definiamo come "fatti". Questa tabella rappresenta il nucleo attorno al quale ruotano le altre tabelle di supporto, chiamate **tabelle di dimensione**, che contengono dettagli descrittivi su aspetti diversi dei fatti. La struttura a stella si ottiene poiché le tabelle di dimensione si collegano alla tabella dei fatti centralmente, dando una disposizione visiva simile a una stella.

La Tabella dei Fatti

La **tabella dei fatti** è la componente principale dello schema, contenente dati misurabili che rappresentano eventi o azioni. Questi dati sono spesso di tipo quantitativo, numerico e continuo, il che li rende adatti per essere sommati, mediati o aggregati a più livelli. Ad esempio, in un sistema di vendite, la tabella dei fatti potrebbe registrare ogni

transazione di vendita, includendo dati come l'incasso, il numero di prodotti venduti o lo sconto applicato. Tali dati, detti "misure", sono i veri obiettivi dell'analisi, in quanto forniscono informazioni chiave per comprendere l'andamento aziendale.

La tabella dei fatti è organizzata in modo da ottimizzare l'efficienza e la coerenza. La sua chiave primaria è solitamente composta da riferimenti (foreign key) alle chiavi di ciascuna tabella di dimensione, consentendo di correlare ogni misura ai dettagli presenti nelle tabelle di dimensione corrispondenti. Questa struttura a chiavi esterne, unita alla terza forma normale (3NF) della tabella dei fatti, garantisce l'integrità dei dati e riduce le ridondanze.

Le Tabelle di Dimensione

Le **tabelle di dimensione** sono tabelle ausiliarie che completano le informazioni contenute nella tabella dei fatti. Esse contengono dati descrittivi e qualitativi che forniscono contesto ai fatti registrati. Ogni tabella di dimensione rappresenta un aspetto rilevante per l'analisi: ad esempio, una tabella di dimensione potrebbe essere quella del "Prodotto", che contiene informazioni come nome del prodotto, categoria e descrizione. Altre dimensioni comuni sono il "Tempo" (che descrive giorno, mese e anno) e il "Luogo" (con dettagli su negozio, città, regione, e così via).

Le tabelle di dimensione spesso includono strutture gerarchiche per consentire livelli multipli di analisi. Ad esempio, la dimensione Tempo potrebbe avere una gerarchia che parte dal livello giornaliero e si estende a mese, trimestre e anno, mentre la dimensione Luogo potrebbe partire dal negozio e aggregarsi fino a città, provincia e regione. Questa disposizione facilita l'aggregazione e l'analisi a vari livelli di dettaglio. A differenza della tabella dei fatti, le tabelle di dimensione sono spesso denormalizzate, ovvero presentano una certa ridondanza. Questo permette un accesso più veloce ai dati, che risulta vantaggioso per i tempi di risposta delle query.

Additività delle Misure e Importanza nell'Analisi

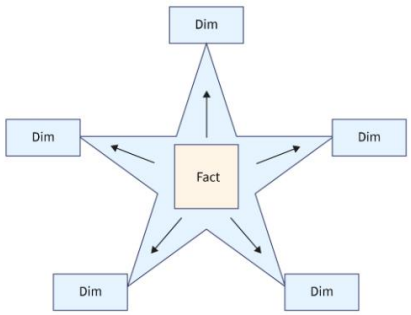
Una delle caratteristiche più potenti dello schema a stella è l'**additività delle misure** contenute nella tabella dei fatti. Quando una misura è additiva, può essere sommata lungo le varie dimensioni, permettendo così di ottenere facilmente informazioni aggregate e di alto valore. Questo significa, per esempio, che se l'incasso è una misura additiva, è possibile sommarlo per calcolare il totale delle vendite in un dato periodo, per una specifica categoria di prodotto o per un particolare negozio. Questo rende lo schema a stella particolarmente efficiente per rispondere a domande come "Qual è stato l'incasso totale della regione X nell'ultimo trimestre?" o "Quante unità di un certo prodotto sono state vendute in un negozio specifico?".

Non tutte le misure, però, sono additive. Esistono misure che sono considerate **semi-additive**, poiché possono essere sommate solo su alcune dimensioni. Ad esempio, il saldo di cassa ha senso essere sommato per diverse regioni geografiche, ma non ha senso aggregarlo nel tempo, dato che il saldo rappresenta uno stato in un determinato momento. Infine, alcune misure sono **non-additive**: per esempio, l'indice di soddisfazione del cliente o la percentuale di rendimento non possono essere sommati ma possono essere aggregati in altri modi, come il calcolo della media.

Differenza tra schema a stella e schema a fiocco di neve

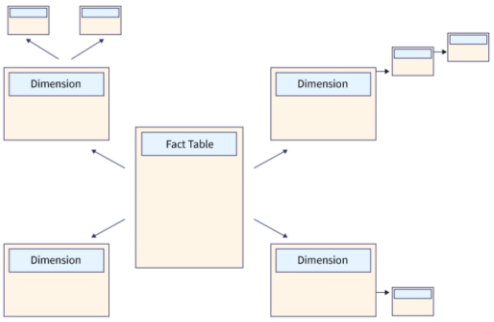
Star Schema e Snowflake Schema sono modelli centrali nella modellazione dei dati per il data warehousing, ognuno con caratteristiche distintive.

Lo **Star Schema** ha una struttura semplice, con una tabella dei fatti centrale circondata da tabelle dimensionali, formando una configurazione simile a una stella. Questa struttura minimizza i collegamenti (join) complessi, rendendo le query più veloci e più adatte per le analisi aziendali che richiedono risposte rapide. Lo **Snowflake Schema**, invece, normalizza i dati suddividendo le tabelle dimensionali in sotto-tabelle, riducendo la ridondanza ma introducendo una complessità maggiore nei collegamenti. Sebbene le query in uno schema a fiocco di neve possano risultare più lente, questo approccio ottimizza lo spazio di archiviazione grazie alla normalizzazione.



Cos'è uno Star Schema?

Lo **Star Schema** è un modello di dati essenziale per la creazione di un data warehouse orientato all'efficacia e alla velocità. Immaginato come una costellazione, lo schema si basa su una tabella dei fatti centrale, che memorizza le metriche di performance principali, e su tabelle dimensionali che forniscono il contesto (come il tempo, la geografia, i prodotti, e così via). Questa disposizione permette query rapide, poiché evita collegamenti complessi e consente un accesso diretto ai dati chiave necessari per le analisi.



Cos'è uno Snowflake Schema?

Lo **schema Snowflake** è un modello di data warehousing progettato per unire l'efficienza di archiviazione con buone prestazioni nelle query. In questo schema, i dati sono organizzati in vari livelli di tabelle normalizzate, riducendo la ridondanza e ottimizzando l'uso dello spazio di archiviazione.

Immaginabile come una struttura ramificata simile a un fiocco di neve, lo schema Snowflake suddivide le tabelle dimensionali in ulteriori tabelle per riflettere relazioni più granulari tra i dati.

Al centro dello schema c'è una tabella dei fatti che raccoglie i KPI aziendali principali, mentre le tabelle dimensionali si diramano in più livelli per memorizzare informazioni contestuali correlate, come dettagli su clienti, prodotti, o geografie. Questo approccio offre un'organizzazione più complessa rispetto al modello Star, ma è ideale per applicazioni che richiedono una struttura dati altamente normalizzata.

Differenza fondamentale tra schema a stella e schema a fiocco di neve

Il punto di forza dello schema a stella è la semplicità: una tabella dei fatti centrale si collega direttamente alle tabelle delle dimensioni. Questa struttura è denormalizzata, il che semplifica l'organizzazione dei dati e velocizza le query grazie a un numero minore di join, anche se comporta una certa duplicazione nei dati delle dimensioni.

Al contrario, lo schema a fiocco di neve mira a ridurre la ridondanza normalizzando le tabelle dimensionali in sottodimensioni. Questo approccio offre una migliore integrità dei dati e un'archiviazione più efficiente, ma richiede join più complessi, che possono rallentare le query.

In breve, scegli Star Schema per semplicità e prestazioni di query più rapide e Snowflake Schema per un'archiviazione ottimizzata e dati più coerenti. La scelta dipende dal bilanciamento tra questi aspetti in base alle necessità del progetto.

Confronto tra schema a stella e schema a fiocco di neve

Aspetto	Schema a stella	Schema a fiocco di neve
Struttura	Tabella dei fatti centrale collegata direttamente alle dimensioni.	Variante estesa dello schema a stella con dimensioni normalizzate in sottodimensioni.
Normalizzazione	Dati denormalizzati, che causano una certa ridondanza.	Dati più normalizzati, con meno duplicazioni tra le dimensioni.
Prestazioni delle query	Query generalmente più veloci grazie al minor numero di join.	Più join necessari, con potenziale impatto negativo sulla velocità.
Manutenzione	Semplice da gestire, con meno tabelle e relazioni.	Più complesso, richiede maggiore attenzione per tabelle e relazioni aggiuntive.
Efficienza di archiviazione	Maggior utilizzo di spazio per via della ridondanza.	Ottimizza lo spazio di archiviazione grazie alla normalizzazione.

Caso d'uso	Ideale per logiche aziendali semplici e analisi rapide.	Adatto a logiche aziendali più complesse e con necessità di dati normalizzati.

La Progettazione di un Data Warehouse (DW)

La progettazione di un **Data Warehouse (DW)** è un processo complesso che coinvolge diverse fasi, ciascuna focalizzata su un aspetto specifico della creazione del sistema. Dall'identificazione dei dati da raccogliere all'effettiva implementazione fisica, il percorso di progettazione segue un approccio strutturato, che garantisce che i dati siano organizzati in modo efficace per le analisi aziendali. Di seguito, analizziamo le principali fasi del processo di progettazione di un data warehouse.

1. Identificazione di Fatti, Misure e Dimensioni

La prima fase del processo di progettazione di un data warehouse consiste nell'identificare i dati rilevanti per l'organizzazione. In particolare, l'analisi iniziale si concentra su:

- **Fatti**: I fatti sono gli eventi o le transazioni fondamentali che il data warehouse deve registrare e analizzare. Questi sono tipicamente eventi misurabili che rappresentano azioni aziendali significative, come le **vendite** di un prodotto, gli **incassi** giornalieri o il **numero di prodotti venduti**. I fatti sono associati a misure numeriche, come le quantità, i valori monetari o altri indicatori quantitativi.
- **Misure**: Le misure sono le variabili quantitative che possono essere analizzate nel contesto dei fatti. Ad esempio, per un evento di vendita, le misure potrebbero includere **il totale delle vendite**, **i ricavi generati**, **il margine di profitto**, ecc. Le misure sono numeriche e devono essere strutturate in modo tale da permettere operazioni matematiche come sommare, mediare o fare calcoli aggregati.
- **Dimensioni**: Le dimensioni sono le variabili qualitative che descrivono i fatti e forniscono contesto per l'analisi. Le dimensioni rappresentano i punti di vista da cui i fatti possono essere esplorati. Esempi di dimensioni includono **il tempo** (giorno, mese, anno), **il prodotto** (categoria, marca, tipo), **la geografia** (paese, regione, città) e **il cliente** (segmento di mercato, età, genere). Le dimensioni consentono di segmentare e filtrare i fatti in base a variabili rilevanti per le analisi aziendali.

2. Ristrutturazione dello Schema Concettuale

Una volta identificati fatti, misure e dimensioni, è necessario ristrutturare lo schema concettuale del data warehouse. Questa fase implica:

- **Rappresentazione dei Fatti mediante Entità**: In questa fase, i fatti vengono mappati come entità all'interno di un modello concettuale. Le entità sono oggetti o concetti che devono essere modellati nel database, come, ad esempio, una **transazione di vendita** o un **incasso**. Ogni entità si associa a misure numeriche, come le vendite totali o i ricavi.
- **Individuazione di Nuove Dimensioni**: Durante la fase di progettazione concettuale, potrebbero emergere nuove dimensioni che non erano state inizialmente previste. È importante identificare tutte le possibili dimensioni utili per l'analisi dei fatti. Per esempio, potrebbero essere necessarie dimensioni aggiuntive come **il canale di vendita** (online, fisico), **la campagna pubblicitaria**, o **il fornitore**.
- **Raffinamento dei Livelli di Ogni Dimensione**: Ogni dimensione dovrebbe essere strutturata con livelli di dettaglio gerarchico che facilitano le operazioni di aggregazione. Ad esempio, la dimensione **Tempo** potrebbe avere i seguenti livelli gerarchici: Giorno > Mese > Anno. Analogamente, la dimensione **Luogo** potrebbe essere organizzata in: Negozio > Città > Regione > Paese.

3. Derivazione del Grafico Dimensionale

Un aspetto cruciale della progettazione concettuale è la creazione di un **grafo dimensionale**. Questo grafo rappresenta visivamente le dimensioni e le loro relazioni con i fatti. Ogni dimensione è rappresentata come un nodo nel grafo, con connessioni (o relazioni) che legano ciascuna dimensione alla tabella dei fatti.

- Il grafo dimensionale aiuta a visualizzare e comprendere come le dimensioni sono correlate tra loro. Ad esempio, una dimensione **Prodotto** può essere collegata sia alla dimensione **Categoria** che alla dimensione

Marca. Questo aiuta a progettare in modo efficiente il database e a capire quali informazioni devono essere memorizzate e come interagiscono tra loro.

4. Progettazione Logica: Derivazione dello Schema Multidimensionale

Una volta definito il modello concettuale, la fase successiva è la **progettazione logica**, che implica la traduzione del modello concettuale in uno schema multidimensionale. Qui, il focus è su come strutturare fisicamente i dati in modo che possano essere facilmente interrogati e aggregati.

- Il modello multidimensionale definisce come le dimensioni interagiscono con i fatti. Il risultato tipico di questa fase è la creazione di uno **schema a stella** o **schema a fiocco di neve**, in cui le tabelle dei fatti sono collegate a tabelle di dimensioni.
- Lo schema multidimensionale è progettato per ottimizzare l'analisi e le operazioni di reporting, come il **drill-down** (approfondire i dettagli), il **roll-up** (aggregare i dati) e il **pivoting** (ristrutturare i dati in base a nuove dimensioni). Questo schema deve essere facilmente estendibile e flessibile per supportare future analisi e nuovi requisiti aziendali.

5. Progettazione Fisica: Determinazione dello Schema Relazionale a Stella

La fase finale è la **progettazione fisica**, che riguarda la realizzazione effettiva del database. Qui si determina come le tabelle e i dati saranno effettivamente memorizzati nel sistema di gestione del database (DBMS).

- **Schema Relazionale a Stella**: In questa fase, lo schema multidimensionale viene tradotto in uno schema relazionale a stella. Le tabelle dei fatti vengono create con chiavi primarie e chiavi esterne che le collegano alle tabelle delle dimensioni. Ogni tabella di dimensione contiene una chiave primaria, che diventa una chiave esterna nella tabella dei fatti, creando il legame tra i dati numerici e quelli descrittivi.
- **Ottimizzazione delle Prestazioni**: La progettazione fisica si concentra anche sull'ottimizzazione delle prestazioni del sistema, come l'indicizzazione delle tabelle, la gestione dello spazio di archiviazione e l'efficienza delle query. Poiché il data warehouse è spesso utilizzato per analisi complesse, è cruciale che la progettazione fisica garantisca tempi di risposta rapidi e l'efficiente gestione dei dati.

Business Intelligence