Basi di Dati

Alessio Delgadillo

Anno accademico 2020-2021

Indice

1 Introduzione **4**
- 1.1 Sistemi informativi 4
- 1.2 Sistemi informatici 4
  - 1.2.1 Sistemi informatici operativi 6
  - 1.2.2 Sistemi informatici direzionali 6
- 1.3 Analisi dei dati 8
- 1.4 Sistemi per basi di dati (Data Base Management Systems - DBMS) 8
  - 1.4.1 Architettura dei DBMS centralizzati 8
  - 1.4.2 Funzionalità dei DBMS 9
  - 1.4.3 Conclusione 12

2 Progettazione di Basi di Dati **14**
- 2.1 Aspetto Ontologico: cosa si modella 16
- 2.2 Aspetto Logico: il modello dei dati a oggetti 16
- 2.3 Cosa si modella: la conoscenza concreta 16
  - 2.3.1 Conoscenza concreta: entità e proprietà 17
  - 2.3.2 Conoscenza concreta: collezioni 18
  - 2.3.3 Modellazione a oggetti: classi e oggetti 18
  - 2.3.4 Conoscenza concreta: le associazioni 18
  - 2.3.5 Conoscenza concreta: gerarchie di classi 20
- 2.4 Cosa si modella: la conoscenza astratta 21
- 2.5 La costruzione di una Base di Dati 22
  - 2.5.1 Analisi dei requisiti 22
  - 2.5.2 Progettazione concettuale di schemi settoriali 23

3 Il modello relazionale **24**
- 3.1 Il modello relazionale 25
  - 3.1.1 Vincoli di integrità 26
  - 3.1.2 Chiave 26
  - 3.1.3 Integrità referenziale 27

3.2 Trasformazione di schemi 27
3.2.1 Progettazione logica relazionale 27
3.2.2 Rappresentazione delle associazioni uno a molti 28
3.2.3 Rappresentazione delle associazioni uno ad uno 28
3.2.4 Vincoli sulla cardinalità delle associazioni uno a molti e uno ad uno 28
3.2.5 Rappresentazione delle associazioni molti a molti . . . 29
3.2.6 Rappresentazione delle gerarchie fra classi 29

4 Algebra Relazionale **31**
4.1 Linguaggi per basi di dati 31
4.1.1 Linguaggi Relazionali 31
4.1.2 Trasformazioni Algebriche 38
4.1.3 Calcolo relazionale su ennuple 39

5 SQL **40**
5.1 Interrogazione di una base dati 40
5.2 Ordinamento, Operatori aggregati e Raggruppamento 44
5.2.1 Ordinamento del risultato 44
5.2.2 Operatori aggregati 44
5.2.3 Raggruppamento 45
5.3 SQL e Algebra Relazionale 46
5.3.1 Join 47
5.4 Sub-Query 49
5.5 Unione, Intersezione, Differenza 51
5.6 SQL per la modifica di basi di dati 53

6 Data Definition Language(DDL) **55**
6.1 I vincoli 59
6.1.1 UNIQUE 59
6.1.2 Primary key 60
6.1.3 Vincoli interrelazionali 60
6.1.4 Check 61
6.1.5 Reazione alla violazione 61
6.2 Viste 62
6.2.1 Vantaggi delle viste 65
6.3 Procedure e Trigger 66
6.3.1 Trigger 66
6.4 Controllo degli accessi 68
6.5 Indice e catalogo 69

7 Programmazione e SQL **70**
7.1 Uso di SQL da programmi 70
7.1.1 Un linguaggio integrato: PL/SQL 71
7.1.2 Linguaggio con interfaccia API 71
7.1.3 SQLJ: Java che ospita l'SQL 71

8 La Normalizzazione **72**
8.1 Teoria Relazionale 72
8.1.1 Dipendenze funzionali 73
8.1.2 Regole di inferenza 75
8.1.3 Chiavi e attributi primi 77
8.2 Copertura canonica 77
8.3 Decomposizione di Schemi 79
8.4 Forme normali 81
8.4.1 Forma normale di Boyce e Codd 81
8.4.2 Terza forma normale 82

9 DBMS **84**
9.1 Gerarchia delle memorie 85
9.1.1 Gli Hard Disk 86
9.1.2 Gestore della memoria permanente e gestore del buffer 86
9.2 Gestore delle strutture di memorizzazione 88
9.2.1 Organizzazione seriale e sequenziale 88
9.2.2 Organizzazione per chiave 88
9.2.3 B^+-tree 89
9.2.4 Indici e B^+-tree 93
9.3 Ordinamento di archivi 94
9.3.1 Realizzazione degli operatori relazionali 95
9.4 Piani di accesso 98
9.4.1 Operatori fisici 98
9.5 Transazioni 100
9.5.1 Gestione delle transazioni 102
9.6 Gestore della concorrenza 106

Capitolo 1

Introduzione

Una **base di dati** è un insieme organizzato di dati utilizzati per il supporto allo svolgimento di attività (di un ente, azienda, ufficio, persona).

Le figure coinvolte nella costruzione di una base di dati sono:

- Committente: dirigente, operatore.
- Fornitore: direttore del progetto, analista, progettista di BD, programmatore di applicazioni che usano BD.
- Manutenzione e messa a punto della BD/Gestione del DBMS: amministratore del DBMS.

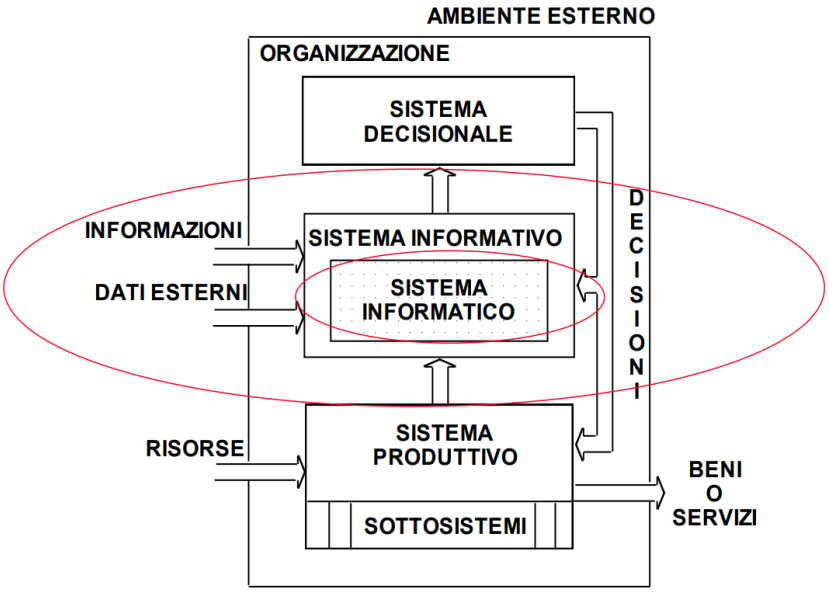
1.1 Sistemi informativi

Definizione 1.1.1. Un **sistema informativo** di un'organizzazione è una combinazione di risorse, umane e materiali, e di procedure organizzate per la raccolta, l'archiviazione, l'elaborazione e lo scambio delle informazioni necessarie alle attività operative (informazioni di servizio), di programmazione e controllo (informazioni di gestione) e di pianificazione strategica (informazioni di governo).

1.2 Sistemi informatici

Il **sistema informativo automatizzato** è quella parte del sistema informativo in cui le informazioni sono raccolte, elaborate, archiviate e scambiate *usando un sistema informatico.*

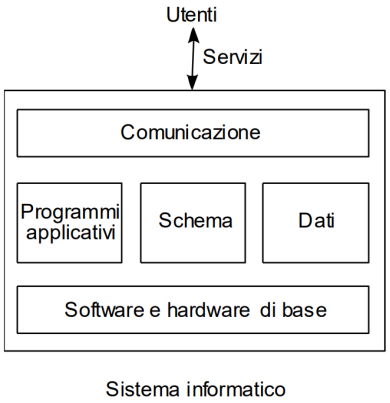
Il **sistema informatico** è l'insieme delle tecnologie informatiche e della comunicazione (Information and Communication Technologies, ICT) a supporto delle attività di un'organizzazione.



Sistema informatico nelle organizzazioni

Terminologia:

- sistema informativo ≈ sistema informativo automatizzato
- sistema informativo automatizzato ≈ sistema informatico

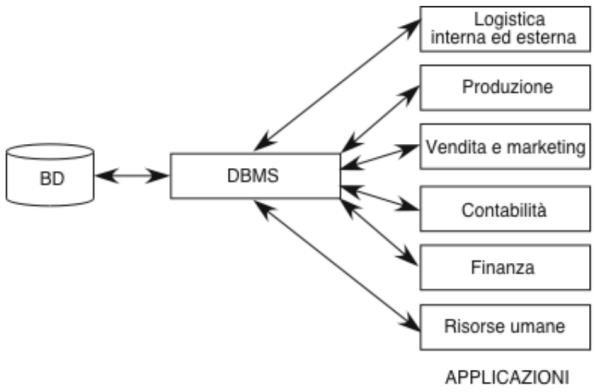


Componenti di un sistema informatico

1.2.1 Sistemi informatici operativi

I dati sono organizzati in basi di dati. Le applicazioni si usano per svolgere le classiche **attività strutturate e ripetitive dell'azienda** nelle aree amministrativa e finanziaria, vendite, produzione, risorse umane, ...

Le caratteristiche delle basi di dati sono garantite da un sistema per la gestione di basi di dati (**Data Base Management System**, **DBMS**), che ha il controllo dei dati e li rende accessibili agli utenti autorizzati.



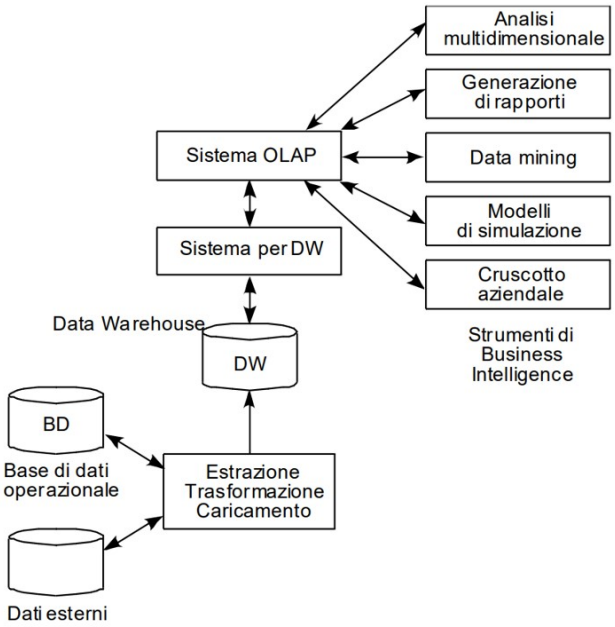
Elaborazioni su BD: OLTP (On-Line Transaction Processing)
Sistemi transazionali

Uso principale dei DBMS. Tradizionale elaborazione di transazioni, che realizzano i processi operativi per il funzionamento di organizzazioni:

- operazioni predefinite e relativamente semplici;
- ogni operazione coinvolge "pochi" dati;
- dati di dettaglio, aggiornati.

1.2.2 Sistemi informatici direzionali

I dati sono organizzati in Data Warehouse (DW) e gestiti da un opportuno sistema. Le applicazioni, dette di *business intelligence*, sono strumenti di supporto ai processi di controllo delle prestazioni aziendali e di decisione manageriale.



Elaborazioni su DW: OLAP (On-Line Analytical Processing)

Uso principale dei **data warehouse**:

- analisi dei dati di supporto alle decisioni;
- operazioni complesse e casuali;
- ogni operazione può coinvolgere molti dati;
- dati aggregati, storici, anche non attualissimi.

	OLTP	**OLAP**
Scopi	Supporto operatività	Supporto decisioni
Utenti	Molti, esecutivi	Pochi, dirigenti e analisti
Dati	Analitici, relazionali	Sintetici, multidimensionali
Usi	Noti a priori	Poco prevedibili
Quantità di dati per attività	Bassa (decine)	Alta (milioni)
Orientamento	Applicazione	Soggetto
Aggiornamenti	Frequenti	Rari
Visione dei dati	Corrente	Storica
Ottimizzati per	Transazioni	Analisi dei dati

Differenze tra OLTP e OLAP

1.3 Analisi dei dati

Dati **aggregati**: non interessa un dato, ma la somma, la media, il minimo, il massimo di una misura.

Presentazione **multidimensionale**: interessa incrociare le informazioni per analizzarle da punti di vista diversi e valutare i risultati del business per intervenire sui problemi critici o per cogliere nuove opportunità.

Analisi a **diversi livelli di dettaglio**, per esempio una volta scoperto un calo delle vendite in un determinato periodo in una regione specifica, si passa ad un'analisi dettagliata nell'area di interesse per cercare di scoprire le cause (dimensioni con gerarchie).

Big Data

Big Data è un termine ampio, riferito a situazioni in cui l'approccio "schema-first" tipico di DB o DW risulta troppo restrittivo o troppo lento. Le tre V: $v$olume, $v$arietà e $v$elocità.

I Big Data sono in genere associati a sistemi NoSQL, Machine Learning e approccio Data Lake.

1.4 Sistemi per basi di dati (Data Base Management Systems - DBMS)

Definizione 1.4.1. Un DBMS è un sistema centralizzato o distribuito che offre opportuni linguaggi per:

- definire lo **schema** di una basi di dati (lo schema va definito prima di creare dati);
- scegliere le **strutture dati** per la memorizzazione dei dati;
- memorizzare i dati rispettando i **vincoli** definiti nello schema;
- recuperare e modificare i dati interattivamente (linguaggio di interrogazione o **query language**) o da programmi.

1.4.1 Architettura dei DBMS centralizzati

Una base di dati è una raccolta di dati permanenti suddivisi in due categorie. I **metadati** descrivono fatti sullo schema dei dati, utenti autorizzati, applicazioni, parametri quantitativi sui dati, ...; sono descritti da uno schema

usando il modello dei dati adottato dal DBMS e sono interrogabili con le stesse modalità previste per i dati. I **dati** sono le rappresentazioni di certi fatti conformi alle definizioni dello schema, con le seguenti caratteristiche:

- sono organizzati in **insiemi strutturati e omogenei** fra i quali sono definite delle **relazioni**. La struttura dei dati e le relazioni sono descritte nello schema usando i meccanismi di astrazione del modello dei dati del DBMS;
- sono **molti**, in assoluto e rispetto ai metadati, e non possono essere gestiti in memoria temporanea;
- sono accessibili mediante **transazioni**, unità di lavoro atomiche che non possono avere effetti parziali;
- sono **protetti** sia da accesso da parte di utenti non autorizzati, sia da corruzione dovuta a malfunzionamenti hardware e software;
- sono utilizzabili **contemporaneamente** da utenti diversi.

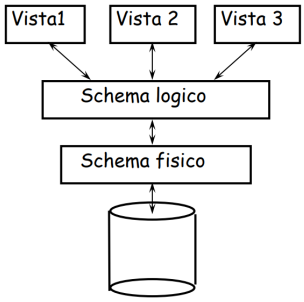
Il **modello relazionale** dei dati è il più diffuso fra i DBMS commerciali. Il meccanismo di astrazione fondamentale è la **relazione** (**tabella**), sostanzialmente un insieme di record con campi elementari; lo schema di una relazione ne definisce il nome e descrive la struttura dei possibili elementi della relazione (insieme di attributi con il loro tipo).

1.4.2 Funzionalità dei DBMS

- Linguaggio per la definizione della base di dati;
- Linguaggi per l'uso dei dati;
- Meccanismi per il controllo dei dati;
- Strumenti per il responsabile della base di dati;
- Strumenti per lo sviluppo delle applicazioni.

Linguaggio per la definizione della base di dati (DDL)

È utile distinguere tre diversi livelli di descrizione dei dati (schemi): il livello di vista logica, il livello logico e quello fisico.



Livello logico: descrive la struttura degli insiemi di dati e delle relazioni fra loro, secondo un certo modello dei dati, senza nessun riferimento alla loro organizzazione fisica nella memoria permanente (schema logico).
Livello fisico: descrive come vanno organizzati fisicamente i dati nelle memorie permanenti e quali strutture dati ausiliarie prevedere per facilitarne l'uso (schema fisico o interno).
Vista logica: descrive come deve apparire la struttura della base di dati ad una certa applicazione (schema esterno o vista).

L'approccio con tre livelli di descrizione dei dati è stato proposto come un modo per garantire le **proprietà di indipendenza logica e fisica dei dati**, che sono fra gli obiettivi più importanti dei DBMS.

- **Indipendenza fisica**: i programmi applicativi non devono essere modificati in seguito a modifiche dell'organizzazione fisica dei dati.
- **Indipendenza logica**: i programmi applicativi non devono essere modificati in seguito a modifiche dello schema logico.

Linguaggi per l'uso dei dati

Un **DBMS** deve prevedere **più modalità d'uso** per soddisfare le esigenze delle diverse categorie di utenti che possono accedere alla base di dati (dati e catalogo):

- Un'interfaccia grafica per accedere ai dati
- Un linguaggio di interrogazione per gli utenti non programmatori
- Un linguaggio di programmazione per gli utenti che sviluppano applicazioni:
  - integrazione DDL e DML nel linguaggio ospite: procedure predefinite, estensione del compilatore, precompilazione

  - comunicazione tra linguaggio e DBMS

- Un linguaggio per lo sviluppo di interfacce per le applicazioni

Strumenti

Strumenti per l'amministratore della base di dati

- Un linguaggio per la definizione e la modifica degli schemi logico, interno ed esterno.
- Strumenti per il controllo e messa a punto del funzionamento del sistema.
- Strumenti per stabilire i diritti di accesso ai dati e per ripristinare la base di dati in caso di malfunzionamenti.

Strumenti per lo sviluppo delle applicazioni

- Produzione di rapporti, grafici, fogli elettronici
- Sviluppo di menù, forme, componenti grafici

Linguaggi per le basi di dati

Disponibilità di vari linguaggi e interfacce diverse: linguaggi testuali interattivi (es. SQL), comandi (come quelli del linguaggio interattivo) immersi in un linguaggio ospite (es. C, Cobol, etc.), comandi (come quelli del linguaggio interattivo) immersi in un linguaggio ad hoc (es. PL/SQL) con altre funzionalità e anche con l'ausilio di strumenti di sviluppo, interfacce amichevoli (senza linguaggio testuale).

Schemi e istanze

In ogni base di dati si distinguono: *lo* ***schema****, sostanzialmente invariante nel tempo, che ne descrive la struttura (aspetto intensionale)*, ovvero le intestazioni delle tabelle, e *l'****istanza****, i valori attuali, che possono cambiare anche molto rapidamente (aspetto estensionale)*, cioè il "corpo" di ciascuna tabella.

Nei linguaggi di interrogazione di basi di dati distinguiamo tra

- **Data Manipulation Language** (**DML**): per l'interrogazione e l'aggiornamento di (istanze di) basi di dati.
- **Data Definition Language** (**DDL**): per la definizione di schemi (logici, esterni, fisici) e altre operazioni generali.

Meccanismi per il controllo dei dati

Una caratteristica molto importante dei DBMS è il tipo di meccanismi offerti per garantire le seguenti proprietà di una base di dati:

- **Integrità**: mantenimento delle proprietà specificate in modo dichiarativo nello schema (vincoli d'integrità).
- **Sicurezza**: protezione dei dati da usi non autorizzati.
- **Affidabilità**: protezione dei dati da malfunzionamenti hardware o software (fallimenti di transazione, di sistema e disastri) e da interferenze indesiderate dovute all'accesso concorrente ai dati da parte di più utenti.

Definizione 1.4.2. Una **transazione** è una sequenza di azioni di lettura e scrittura in memoria permanente e di elaborazioni di dati in memoria temporanea, con le seguenti proprietà:

- **Atomicità**: le transazioni che terminano in modo prematuro (*aborted transactions*) sono trattate dal sistema come se non fossero mai iniziate; pertanto eventuali loro effetti sulla base di dati sono annullati.
- **Serializzabilità**: nel caso di esecuzioni concorrenti di più transazioni, l'effetto complessivo è quello di una esecuzione seriale.
- **Persistenza**: le modifiche sulla base di dati di una transazione terminata normalmente sono permanenti, cioè non sono alterabili da eventuali malfunzionamenti.

1.4.3 Conclusione

Un **DataBase Management System** (**DBMS**) è un sistema (*prodotto software*) in grado di gestire *collezioni di dati* che siano (anche) **grandi**, **persistenti** (con un periodo di vita indipendente dalle singole esecuzioni dei programmi che le utilizzano) e **condivise** (utilizzate da applicazioni diverse), garantendo **affidabilità** (resistenza a malfunzionamenti hardware e software-recovery) e **privatezza** (con una disciplina e un controllo degli accessi). Come ogni prodotto informatico, un DBMS deve essere **efficiente** (utilizzando al meglio le risorse di spazio e tempo del sistema) ed **efficace** (rendendo produttive le attività dei suoi utilizzatori).
Vantaggi

- Indipendenza dei dati

- Recupero efficiente dei dati
- Integrità e sicurezza dei dati
- Accessi interattivi, concorrenti e protetti dai malfunzionamenti
- Amministrazione dei dati
- Riduzione dei tempi di sviluppo delle applicazioni
- La riduzione dei costi della tecnologia e i possibili tipi di DBMS disponibili sul mercato facilitano la loro diffusione.

Svantaggi

- Prima di caricare i dati è necessario definire uno schema
- Possono essere costosi e complessi da installare e mantenere in esercizi
- Possono gestire solo dati strutturati e omogenei
- Sono ottimizzati per le applicazioni OLTP, non per quelle OLAP

Capitolo 2

Progettazione di Basi di Dati

Progettare una basi di dati vuole dire progettare la struttura dei dati e le applicazioni: è l'attività più importante. Per progettare i dati al meglio è necessario che i dati siano un modello fedele del dominio del discorso.

Definizione 2.0.1. Un modello astratto è la rappresentazione formale di idee e conoscenze relative a un fenomeno.

Aspetti di un modello:

- il modello è la rappresentazione di certi fatti;
- la rappresentazione è data con un linguaggio formale;
- il modello è il risultato di un processo di interpretazione, guidato dalle idee e conoscenze possedute dal soggetto che interpreta.

La stessa realtà può utilmente essere modellata in modi diversi e a diversi livelli di astrazione.

Ciascuna di queste fasi è centrata sulla modellazione. La modellazione verrà discussa quindi con riferimento alla problematica della progettazione delle basi di dati.

Per garantire prodotti di buona qualità è opportuno seguire una **metodologia di progetto** con:

- Articolazione delle attività in fasi (**decomposizione**)
- Criteri di scelta (**strategie**)
- **Modelli** di rappresentazione
- **Generalità** rispetto al problema di studio e agli strumenti a disposizione
- **Qualità** del prodotto
- **Facilità** d'uso

Modello dei dati

Insieme di costrutti utilizzati per organizzare i dati di interesse e descriverne la dinamica. Componente fondamentale: **meccanismi di strutturazione** (o costruttori di tipo). Come nei linguaggi di programmazione esistono meccanismi che permettono di definire **nuovi tipi**, così ogni modello dei dati prevede alcuni costruttori. Ad esempio, il modello relazionale prevede il costruttore relazione, che permette di definire insiemi di record omogenei.

Aspetti del problema

Quale conoscenza del dominio del discorso si rappresenta? Aspetto **ontologico** (studio di ciò che esiste): ciò che si suppone esistere nell'universo del discorso e quindi sia da modellare.

Con quali meccanismi di astrazione si modella? Aspetto **logico**.
Con quale linguaggio formale si definisce il modello? Aspetto **linguistico**.
Come si procede per costruire un modello? Aspetto **pragmatico**: metodologia (insieme di regole finalizzate alla costruzione del modello informatico) da seguire nel processo di modellazione

2.1 Aspetto Ontologico: cosa si modella

- Conoscenza concreta: i fatti.
- Conoscenza astratta: struttura e vincoli sulla conoscenza concreta.
- Conoscenza procedurale, comunicazioni:
  - le operazioni di base, le operazioni degli utenti;
  - come si comunicherà con il sistema informatico.

Nel seguito l'attenzione sarà sulla conoscenza concreta e astratta.

2.2 Aspetto Logico: il modello dei dati a oggetti

Un modello dei dati è un insieme di meccanismi di astrazione per descrivere la struttura della conoscenza concreta.

Schema: la descrizione della struttura della conoscenza concreta e dei vincoli di integrità usando un particolare modello dei dati.

Useremo come notazione grafica una **variante** dei cosiddetti diagrammi a oggetti o diagrammi ER (Entità-Relazione). Nozioni fondamentali:

- Oggetto, Tipo di oggetto, Classe
- Ereditarietà, Gerarchia fra tipi, Gerarchia fra classi

2.3 Cosa si modella: la conoscenza concreta

La **conoscenza concreta** riguarda i fatti specifici che si vogliono rappresentare: le entità con le loro proprietà, le collezioni di entità omogenee e le associazioni fra entità.

2.3.1 Conoscenza concreta: entità e proprietà

Le ***entità*** sono ciò di cui interessa rappresentare alcuni fatti (o proprietà), per esempio una descrizione bibliografica di un libro, un libro o documento fisico, un prestito, un utente della biblioteca.

Le ***proprietà*** sono fatti che interessano solo in quanto descrivono caratteristiche di determinate entità, per esempio un indirizzo interessa perché è l'indirizzo di un utente. Classificazione delle proprietà:

- primitiva/strutturata,
- obbligatoria/opzionale,
- univoca/multivalore,
- costante/variabile,
- calcolata/non calcolata.

Conoscenza concreta: collezioni di entità

Una ***proprietà*** è una coppia $\langle$`attributo`, `valore`$\rangle$.
Tipi di entità: ogni entità appartiene ad un tipo che ne specifica la natura. Ad esempio Antonio ha tipo `Persona` con proprietà `Nome: string` e `Indirizzo: string`.
Collezione: un insieme variabile nel tempo di entità omogenee (dello stesso tipo). Ad esempio la collezione di tutte le persone nel dominio del discorso.

Caratteristiche delle proprietà

Ogni **proprietà** ha associato un dominio, ovvero l'insieme dei possibili valori che tale proprietà può assumere:

- proprietà **atomica** se il suo valore non è scomponibile, altrimenti è detta **strutturata**;
- proprietà **univoca** se il suo valore è unico, altrimenti è detta **multivalore**;
- proprietà **totale** (obbligatoria) se ogni entità dell'universo del discorso ha per essa un valore specificato, altrimenti è detta **parziale** (opzionale).

Certi fatti possono essere interpretati come proprietà in certi contesti e come entità in altri.

2.3.2 Conoscenza concreta: collezioni

Una **collezione** è un insieme variabile nel tempo di entità omogenee interessanti dell'universo del discorso.

2.3.3 Modellazione a oggetti: classi e oggetti

Ad ogni entità del dominio corrisponde un oggetto del modello. ***Oggetto***: un'entità software con stato, comportamento e identità che modella un'entità dell'universo:

- lo ***stato*** è modellato da un insieme di costanti o variabili con valori di qualsiasi complessità;
- il ***comportamento*** è un insieme di procedure locali chiamate **metodi** che modellano le operazioni di base che riguardano l'oggetto e le proprietà derivabili da altre.

Un ***oggetto*** può rispondere a dei ***messaggi***, restituendo valori memorizzati nello stato o calcolati con una procedura locale.

Una **classe** è un insieme di oggetti dello stesso tipo, modificabile con operatori per includere o estrarre elementi dall'insieme.

Tipo oggetto

Il primo passo nella costruzione di un modello consiste nella *classificazione delle entità del dominio con la definizione dei tipi degli oggetti che le rappresentano.*

Un ***tipo oggetto*** definisce l'insieme dei messaggi (interfaccia) a cui può rispondere un insieme di possibili oggetti. I **nomi dei messaggi** sono detti anche attributi degli oggetti.

Nei diagrammi ER i tipi oggetti non si rappresentano, l'attenzione è invece sulle collezioni e sulle associazioni; tuttavia, la rappresentazione grafica di una collezione indica anche gli attributi del tipo oggetto associato.

2.3.4 Conoscenza concreta: le associazioni

Un'**istanza di associazione** è un fatto che correla due o più entità, stabilendo un legame logico tra di loro. Un'associazione $R(X, Y)$ fra due collezioni di entità X ed Y è un insieme di istanze di associazione tra elementi di X e Y che in generale varia nel tempo.

Il prodotto cartesiano $(X \times Y)$ è detto dominio dell'associazione.

Tipi di associazione

Un'**associazione** è caratterizzata dalle seguenti proprietà strutturali: **molteplicità** e **totalità**.

Vincolo di univocità: un'associazione $R(X, Y)$ è univoca rispetto a X se per ogni elemento x di X esiste al più un elemento di Y che è associato a x, se non vale questo vincolo, l'associazione è multivalore rispetto ad X.
Cardinalità dell'associazione:

- $R(X, Y)$ è $(1 : N)$ se essa è multivalore su X ed univoca su Y.
- $R(X, Y)$ è $(N : 1)$ se essa è univoca su X e multivalore su Y.
- $R(X, Y)$ è $(N : M)$ se essa è multivalore su X e multivalore su Y.
- $R(X, Y)$ è $(1 : 1)$ se essa è univoca su X e univoca su Y.

Vincolo di totalità: un'associazione $R(X, Y)$ è **totale** (o suriettiva) su X se per ogni elemento x di X esiste almeno un elemento di Y che è associato ad x, se non vale questo vincolo, l'associazione è **parziale** rispetto ad X.

Rappresentazione delle associazioni - Associazione binaria senza proprietà

Un'**associazione** fra due collezioni C_1 e C_2 si rappresenta con una linea che collega le classi che rappresentano le due collezioni. La **linea** è etichettata con il nome dell'associazione che di solito viene scelto utilizzando un predicato ("soggetto predicato complemento"). L'**univocità** di un'associazione, rispetto ad una classe C_1, si rappresenta disegnando una freccia singola sulla linea che esce dalla classe C_1 ed entra nella classe C_2; l'assenza di tale vincolo è indicata da una freccia doppia. Similmente, la **parzialità** è rappresentata con un taglio sulla linea vicino alla freccia, mentre il vincolo di **totalità** è rappresentato dall'assenza di tale taglio.

Questioni terminologiche

Dominio del discorso	*Modello Informatico*
Entità	Oggetto (entity instance)
Tipo entità	Tipo oggetto (entity type)
Collezione	Classe (entity)
Associazione	Associazione (relationship)

Modello a oggetti: le associazioni

Le associazioni si modellano con un costrutto apposito. Possono avere delle proprietà e possono essere ricorsive.

Le **associazioni ricorsive** sono relazioni binarie fra gli elementi di una stessa collezione. Occorre etichettare la linea non solo con il nome dell'associazione, ma anche con dei nomi per specificare il ruolo che hanno i due componenti in un'istanza di associazione.

Per semplicità non daremo una notazione grafica per rappresentare associazioni **non binarie**.

2.3.5 Conoscenza concreta: gerarchie di classi

Spesso le classi di entità sono organizzate in una gerarchia di **specializzazione/generalizzazione**. Una classe della gerarchia minore di altre viene detta **sottoclasse** (le altre sono superclassi). Due importanti caratteristiche delle gerarchie:

- **ereditarietà** delle proprietà;
- gli elementi di una sottoclasse sono un sottoinsieme degli elementi della **superclasse**.

Modello a oggetti: gerarchia tra tipi oggetto

Fra i tipi oggetto è definita una relazione di sottotipo **asimmetrica**, **riflessiva** e **transitiva** (relazione di ordine parziale).

Se T è sottotipo di T', allora gli elementi di T possono essere usati in ogni contesto in cui possano apparire valori di tipo T' (sostituibilità). In particolare:

- gli elementi di T hanno tutte le proprietà degli elementi di T'
- per ogni proprietà p in T', il suo tipo in T è un sottotipo del suo tipo in T'.

La gerarchia può essere semplice o multipla.

Ereditarietà

L'ereditarietà (*inheritance*) permette di definire un tipo oggetto a partire da un altro e l'implementazione di un tipo oggetto a partire da un'altra implementazione. Normalmente l'eredità tra tipi si usa solo per definire sottotipi, e l'ereditarietà tra implementazioni per definire implementazioni di sottotipi

(ereditarietà stretta); in questo caso gli attributi possono essere solo aggiunti e possono essere ridefiniti solo specializzandone il tipo.

Sottoinsiemi

Fra le classi può essere definita una relazione di sottoclasse, detta anche **sottoinsieme**, con le seguenti proprietà:

- È asimmetrica, riflessiva e transitiva.
- Se C è sottoclasse di C', allora il *tipo* degli elementi di C è *sottotipo* del tipo degli elementi di C' (vincolo **intensionale**).
- Se C è sottoclasse di C', allora gli *elementi* di C sono un *sottoinsieme* degli elementi di C' (vincolo **estensionale**).

Vincoli su insiemi di sottoclassi:

- **Disgiunzione**: ogni coppia di sottoclassi in questo insieme è disgiunta, ovvero è priva di elementi comuni (sottoclassi disgiunte).
- **Copertura**: l'unione degli elementi delle sottoclassi coincide con l'insieme degli elementi della superclasse (sottoclassi copertura).

Sottoclassi **scorrelate** non richiedono il vincolo di copertura, né quello di disgiunzione.

Gerarchia multipla

Un tipo può essere definito per ereditarietà a partire da un unico supertipo (ereditarietà singola) o da più supertipi (ereditarietà multipla). Ereditarietà multipla: è molto utile ma può creare alcuni problemi quando lo stesso attributo viene ereditato, con tipi diversi, da più tipi antenato.

2.4 Cosa si modella: la conoscenza astratta

Conoscenza astratta identifica fatti generali che descrivono

- la *struttura* della conoscenza *concreta* (collezioni, tipi entità, associazioni);
- *restrizioni* sui valori possibili della conoscenza concreta e sui modi in cui essi possono evolvere nel tempo (**vincoli d'integrità**): vincoli statici e vincoli dinamici;

- *regole* per derivare nuovi fatti da altri noti.

I vincoli possono essere descritti in modo **dichiarativo** (da preferire), con formule del calcolo dei predicati, oppure mediante **controlli** da eseguire nelle operazioni (di base o degli utenti).

2.5 La costruzione di una Base di Dati

- Analisi dei requisiti: specifica dei requisiti, schemi di settore
- Progettazione concettuale: schema concettuale
- Progettazione logica: schema logico
- Progettazione fisica: schema fisico

Tanto lo schema concettuale che quello logico contengono le viste esterne.

2.5.1 Analisi dei requisiti

In generale, il linguaggio naturale è pieno di ambiguità e fraintendimenti. Si deve, quanto più è possibile, evitare tali ambiguità.
Regole generali:

- Studiare e comprendere il sistema informativo e i bisogni informativi di tutti i settori dell'organizzazione.
- **Scegliere** il corretto livello di **astrazione**.
- **Standardizzare** la struttura delle frasi.
- **Suddividere le frasi** articolate.
- Riorganizzare le frasi per **concetti**, ovvero ottenendo diverse categorie di dati, vincoli e operazioni: **separare** le **frasi sui dati** da quelle sulle **funzioni**.
- Eliminare ambiguità, imprecisioni e disuniformità: individuare **omonimi** e **sinonimi** e unificare i termini; rendere esplicito il **riferimento fra termini**.
- Costruire un **glossario** dei termini.
- Disegnare lo schema.
- Specificare le operazioni.
- Verificare la coerenza fra operazioni e dati.

Specifiche sulle operazioni

Dopo aver definito i dati, occorre stabilire le specifiche sulle operazioni da effettuare sui dati. Bisogna utilizzare la stessa terminologia usata per i dati e inoltre bisognerebbe indicare la frequenza con cui vengono effettuate certe operazioni.

La conoscenza di queste informazioni è indispensabile nella fase di progettazione logica.

2.5.2 Progettazione concettuale di schemi settoriali

- Identificare le classi
- Identificare le associazioni e le loro proprietà strutturali
- Identificare gli attributi delle classi e associazioni e i loro tipi
- Elencare le chiavi
- Individuare le sottoclassi
- Individuare le generalizzazioni

Capitolo 3

Il modello relazionale

Proposto da **E. F. Codd** nel **1970** per favorire l'indipendenza dei dati. Disponibile in DBMS reali nel **1981** (non è facile implementare l'indipendenza con efficienza e affidabilità!). Si basa sul concetto matematico di **relazione** (con una variante). Le relazioni hanno naturale rappresentazione per mezzo di **tabelle**.

Relazione matematica: come nella teoria degli insiemi.

$$D_1, \ldots, D_n \ (n \text{ insiemi anche non distinti})$$

Prodotto cartesiano $D_1 \times \ldots \times D_n$:

l'insieme di tutte le *n-uple* $(d_1, \ldots, d_n)$ tali che $d_1 \in D_1, \ldots, d_n \in D_n$

Relazione matematica su $D_1, \ldots, D_n$:

$$\text{un sottoinsieme di } D_1 \times \ldots \times D_n$$

$D_1, \ldots, D_n$ sono i **domini** della relazione.

Osservazione: una relazione è un insieme quindi

- non c'è ordinamento fra le n-uple;
- le n-uple sono distinte;
- ciascuna n-upla è ordinata, l'i-esimo valore proviene dall'i-esimo dominio.

Tabelle e relazioni

Una tabella rappresenta una **relazione** se i valori di ogni colonna sono fra loro omogenei e se le righe e le intestazioni delle colonne sono diverse fra loro.

In una tabella che rappresenta una relazione l'ordinamento tra le righe e l'ordinamento tra le colonne sono irrilevanti.

Modello basato su valori

Il **modello relazionale è basato su valori**. Ciò significa che i riferimenti fra dati in relazioni diverse sono rappresentati per mezzo di valori dei domini che compaiono nelle ennuple.

Vantaggi della struttura basata su valori

- **Indipendenza dalle strutture fisiche** (si potrebbe avere anche con puntatori di alto livello) che possono cambiare dinamicamente. La rappresentazione logica dei dati (costituita dai soli valori) non fa riferimento a quella fisica.
- Si rappresenta solo ciò che è **rilevante** dal punto di vista dell'applicazione.
- I dati sono **portabili** più facilmente da un sistema ad un altro.
- I **puntatori sono direzionali**.

3.1 Il modello relazionale

Definizione 3.1.1. I meccanismi per definire una base di dati con il modello relazionale sono l'**ennupla** e la **relazione**.

Un **tipo ennupla** T è un insieme finito di coppie

$$\langle \texttt{Attributo}, \texttt{Tipo elementare} \rangle$$

Se T è un tipo ennupla, $R(T)$ è **lo schema della relazione *R***; lo schema di una base di dati è un insieme di schemi di relazione $R_i(T_i)$. Un'**istanza** di uno schema $R(T)$ è un insieme finito di ennuple di tipo T.

Uguaglianza di due tipi ennupla, due ennuple, due tipi relazione.

Informazione incompleta

Il modello relazionale impone ai dati una struttura rigida: le informazioni sono rappresentate per mezzo di ennuple, ma solo alcuni formati di ennuple sono ammessi, cioè quelli che corrispondono agli schemi di relazione.

Valore nullo: denota l'assenza di un valore del dominio (e non è un valore del dominio). `t[A]`, per ogni attributo A, è un valore del dominio $dom(A)$ oppure il valore nullo `NULL`. Si possono (e devono) imporre restrizioni sulla presenza di valori nulli.

3.1.1 Vincoli di integrità

Esistono istanze di basi di dati che, pur sintatticamente corrette, non rappresentano informazioni possibili per l'applicazione di interesse e che quindi generano informazioni senza significato.

Uno **schema relazionale** è costituito da un insieme di **schemi di relazione** e da un **insieme di vincoli** d'integrità sui possibili valori delle estensioni delle relazioni. Un **vincolo d'integrità** è una proprietà che deve essere soddisfatta dalle istanze che rappresentano informazioni corrette per l'applicazione. Un vincolo è espresso mediante una funzione booleana (un predicato): associa ad ogni istanza il valore vero o falso.
Perché usare i vincoli di integrità?

- Descrizione più accurata della realtà.
- Contributo alla "qualità dei dati".
- Utili nella progettazione.
- Usati dai DBMS nella esecuzione delle interrogazioni.
- Non tutte le proprietà di interesse sono rappresentabili per mezzo di vincoli formulabili in modo esplicito.

Tipi di vincoli

Vincoli **intrarelazionali**: sono i vincoli che devono essere rispettati dai valori contenuti nella relazione considerata; vincoli su valori (o di dominio) e vincoli di ennupla.
Vincoli **interrelazionali**: sono i vincoli che devono essere rispettati da valori contenuti in relazioni diverse.

Vincoli di ennupla

I **Vincoli di ennupla** esprimono condizioni sui valori di ciascuna ennupla, indipendentemente dalle altre ennuple. **Vincoli di dominio**: caso particolare, coinvolgono un solo attributo.

3.1.2 Chiave

Informalmente: una **chiave** è un insieme di attributi che identifica le ennuple di una relazione.

Formalmente: un insieme K di attributi è **superchiave** per r se r non contiene due ennuple (distinte) t_1 e t_2 con $t_1[K] = t_2[K]$. K è **chiave** per r se è una superchiave minimale per r (cioè non contiene un'altra superchiave).

Esistenza delle chiavi

Una relazione non può contenere ennuple distinte ma con valori uguali (una relazione è un sottoinsieme del prodotto cartesiano). Ogni relazione ha *sicuramente* come **superchiave** l'insieme di tutti gli attributi su cui è definita e quindi ogni relazione *ha (almeno) una chiave*. L'esistenza delle chiavi garantisce l'accessibilità a ciascun dato della base di dati. Le chiavi permettono di correlare i dati in relazioni diverse:

il modello relazionale è basato su valori

La presenza di valori nulli fra i valori di una chiave non permette di identificare le ennuple e di realizzare facilmente i riferimenti da altre relazioni.

Una **chiave primaria** è una chiave su cui non sono ammessi valori nulli.

3.1.3 Integrità referenziale

Nel modello relazionale le informazioni in relazioni diverse sono correlate attraverso valori comuni, in particolare vengono spesso presi in considerazione i valori delle chiavi (primarie).

Le correlazioni debbono essere "coerenti".

Vincolo di integrità referenziale

Un vincolo di **integrità referenziale** ("**foreign key**") fra gli attributi X di una relazione R_1 e un'altra relazione R_2 impone ai valori su X in R_1 di comparire come valori della chiave primaria di R_2.

3.2 Trasformazione di schemi

3.2.1 Progettazione logica relazionale

Si tratta di "*tradurre*" lo schema concettuale in uno schema logico relazionale che rappresenti gli stessi dati in maniera **corretta** ed **efficiente**; questo richiede una ristrutturazione del modello concettuale.

La trasformazione di uno schema ad oggetti in uno schema relazionale avviene eseguendo i seguenti passi:

1. rappresentazione delle associazioni uno ad uno e uno a molti;
2. rappresentazione delle associazioni molti a molti o non binarie;
3. rappresentazione delle gerarchie di inclusione;
4. identificazione delle chiavi primarie;
5. rappresentazione degli attributi multivalore;
6. appiattimento degli attributi composti.

Obiettivo:

- rappresentare le stesse informazioni;
- **minimizzare la ridondanza**;
- produrre uno schema comprensibile, per facilitare la scrittura e manutenzione delle applicazioni.

3.2.2 Rappresentazione delle associazioni uno a molti

Le associazioni uno a molti si rappresentano aggiungendo agli attributi della relazione rispetto a cui l'associazione è univoca una chiave esterna che riferisce l'altra relazione.

3.2.3 Rappresentazione delle associazioni uno ad uno

Le associazioni uno a uno si rappresentano aggiungendo la chiave esterna ad una qualunque delle due relazioni che riferisce l'altra relazione, preferendo quella rispetto a cui l'associazione è **totale**, nel caso in cui esista un vincolo di totalità.

3.2.4 Vincoli sulla cardinalità delle associazioni uno a molti e uno ad uno

La direzione dell'associazione rappresentata dalla chiave esterna è detta "*la diretta*" dell'associazione. Vincoli sulla cardinalità delle associazioni uno a molti ed uno ad uno:

- **univocità della diretta**;
- **totalità della diretta**: si rappresenta imponendo un vincolo `not null` sulla chiave esterna;

- **univocità dell'inversa** e **totalità della diretta**: si rappresenta imponendo un vincolo not null ed un vincolo di chiave sulla chiave esterna.

3.2.5 Rappresentazione delle associazioni molti a molti

Un'associazione molti a molti tra due classi si rappresenta aggiungendo allo schema una nuova relazione che contiene due chiavi esterne che riferiscono le due relazioni coinvolte; la chiave primaria di questa relazione è costituita dall'insieme di tutti i suoi attributi.

3.2.6 Rappresentazione delle gerarchie fra classi

Il modello relazionale non può rappresentare direttamente le gerarchie, bisogna eliminarle, sostituirle con classi e relazioni:

1. accorpamento delle figlie della gerarchia nel genitore (**relazione unica**);
2. accorpamento del genitore della gerarchia nelle figlie (**partizionamento orizzontale**);
3. sostituzione della gerarchia con relazioni (**partizionamento verticale**).

Accorpamento delle figlie della gerarchia nel genitore

Se A_0 è la classe genitore di A_1 ed A_2, le classi A_1 ed A_2 vengono eliminate ed accorpate ad A_0. Ad A_0 viene aggiunto un **attributo** (*discriminatore*) che indica da quale delle classi figlie deriva una certa istanza, e gli **attributi di A_1 ed A_2** vengono assorbiti dalla classe genitore, e assumono valore nullo sulle istanze provenienti dall'altra classe.

Infine, una **relazione relativa a solo una delle classi figlie** viene acquisita dalla classe genitore e avrà comunque cardinalità minima uguale a 0, in quanto gli elementi dell'altra classe non contribuiscono alla relazione.

Accorpamento del genitore della gerarchia nelle figlie

La classe genitore A_0 viene eliminata e le classi figlie A_1 ed A_2 ereditano le proprietà (attributi, identificatore e relazioni) della classe genitore. Le relazioni che coinvolgono la classe genitore vengono sdoppiate, coinvolgendo ciascuna delle classe figlie.

Il partizionamento orizzontale divide gli elementi della superclasse in più relazioni diverse, per cui *non è possibile mantenere un vincolo referenziale*

verso la superclasse stessa; in conclusione, questa tecnica non si usa se nello schema relazionale grafico c'è una freccia che entra nella superclasse;

Sostituzione della gerarchia con relazioni

La gerarchia si trasforma in due associazioni uno a uno che legano rispettivamente la classe genitore con le classi figlie. In questo caso non c'è un trasferimento di attributi o di associazioni e le classi figlie A_1 ed A_2 sono identificate esternamente dalla classe genitore A_0.

Nello schema ottenuto vanno aggiunti dei vincoli: ogni occorrenza di A_0 non può partecipare contemporaneamente alle due associazioni, e se la gerarchia è totale, deve partecipare ad almeno una delle due.

Capitolo 4

Algebra Relazionale

4.1 Linguaggi per basi di dati

Operazioni sullo schema → **DDL**: **Data Definition Language**

Operazioni di creazione, cancellazione e modifica di schemi di tabelle, creazione viste, creazione indici...

Operazioni sui dati → **DML**: **Data Manipulation Language**

- Data Query language: query o interrogazione della base di dati
- Aggiornamento dati: inserimento, cancellazione e modifica di dati

4.1.1 Linguaggi Relazionali

Algebra *relazionale*: insieme di operatori su relazioni che danno come risultato relazioni. Non si usa come linguaggio di interrogazione dei DBMS ma come rappresentazione interna delle interrogazioni.

Calcolo *relazionale*: linguaggio dichiarativo di tipo logico dal quale è stato derivato l'SQL.

Operatori dell'algebra relazionale

- unione, intersezione, differenza;
- ridenominazione;
- selezione;
- proiezione;

- join (join naturale, prodotto cartesiano, theta-join).

Operatori insiemistici

- le relazioni sono insiemi
- i risultati devono essere relazioni

è possibile applicare **unione**, **intersezione**, **differenza** solo a relazioni definite sugli stessi attributi, cioè possono operare solo su tuple uniformi.

Unione: ∪

Laureati

Matricola	Nome	Età
7274	Rossi	42
7432	Neri	54
9824	Verdi	45

Quadri

Matricola	Nome	Età
9297	Neri	33
7432	Neri	54
9824	Verdi	45

Laureati ∪ Quadri

Matricola	Nome	Età
7274	Rossi	42
7432	Neri	54
9824	Verdi	45
9297	Neri	33

Intersezione: ∩

Laureati

Matricola	Nome	Età
7274	Rossi	42
7432	Neri	54
9824	Verdi	45

Quadri

Matricola	Nome	Età
9297	Neri	33
7432	Neri	54
9824	Verdi	45

Laureati ∩ Quadri

Matricola	Nome	Età
7432	Neri	54
9824	Verdi	45

Algebra Relazionale: Differenza

Laureati

Matricola	Nome	Età
7274	Rossi	42
7432	Neri	54
9824	Verdi	45

Quadri

Matricola	Nome	Età
9297	Neri	33
7432	Neri	54
9824	Verdi	45

Laureati – Quadri

Matricola	Nome	Età
7274	Rossi	42

Ridenominazione

Operatore monadico (con un argomento), "modifica lo schema" lasciando inalterata l'istanza dell'operando.

È indicato con la lettera ρ

Paternità

Padre	Figlio
Adamo	Abele
Adamo	Caino
Abramo	Isacco

$\rho_{\text{Genitore} \leftarrow \text{Padre}}$

Genitore	Figlio
Adamo	Abele
Adamo	Caino
Abramo	Isacco

Maternità

Madre	Figlio
Eva	Abele
Eva	Set
Sara	Isacco

$\rho_{\text{Genitore} \leftarrow \text{Madre}}$

Genitore	Figlio
Eva	Abele
Eva	Set
Sara	Isacco

Proiezione

Operatore monadico, produce un risultato che ha parte degli attributi dell'operando e che contiene ennuple cui contribuiscono tutte le ennuple dell'operando ristrette agli attributi nella lista. Sintassi:

$$\pi_{\text{ListaAttributi}}(\text{Operando})$$

Proiezione: $\pi_{A_1,\ldots,A_N}(R)$
Una proiezione contiene al più tante ennuple quante l'operando, può contenerne di meno.

Se X è una *superchiave* di R, allora $\pi_x(R)$ contiene esattamente tante ennuple quante R. Se X non è superchiave, potrebbero esistere valori ripetuti su quegli attributi, che quindi vengono rappresentati una sola volta.

Selezione (Restrizione)

Operatore monadico, produce un risultato che ha lo stesso schema dell'operando e che contiene un sottoinsieme delle ennuple dell'operando, cioè quelle che soddisfano una condizione espressa combinando, con i connettivi logici $\wedge$ (and), $\vee$ (or), $\neg$ (not), condizioni atomiche del tipo $A\ \Theta\ B$ o $A\ \Theta\ c$, dove Θ è un operatore di confronto, A e B sono attributi su cui l'operatore Θ abbia senso, c una costante compatibile col dominio di A.

È denotata con σ, con la condizione messa a pedice: $\sigma_{\text{Condizione}}(R)$.

Composizione di operatori: $\pi_{\text{Matricola}}(\sigma_{\text{Nome}=\textit{Caio}}(\text{Studenti}))$.

- Cond ::= Espr Θ Espr | Cond $\wedge$ Cond | $\neg$ Cond

- Espr ::= Attributo | Costante | Espr *Op* Espr
- Θ ::= = | < | > | $\neq$ | $\leq$ | $\geq$
- *Op* ::= + | - | * | StringConcat

$$\sigma_{\text{Età}> 30}\ (\text{Persone}) \cup \sigma_{\text{Età}\leq 30}\ (\text{Persone}) \neq \text{Persone}$$

Perché le selezioni vengono valutate separatamente! Ma anche

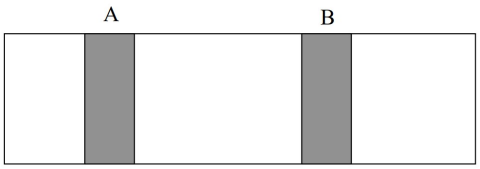
$$\sigma_{\text{Età}> 30\ \vee\ \text{Età}\leq 30}(\text{Persone}) \neq \text{Persone}$$

Perché anche le condizioni atomiche vengono valutate separatamente! La condizione atomica è vera solo per valori **non nulli**, per riferirsi ai valori nulli esistono forme apposite di condizioni: `IS NULL`, `IS NOT NULL`.

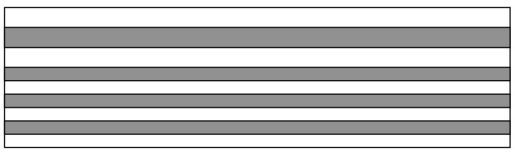
$$\begin{gathered}\sigma_{\text{Età}> 30}\ (\text{Persone}) \cup \sigma_{\text{Età}\leq 30}\ (\text{Persone}) \cup \sigma_{\text{Età IS NULL}}\ (\text{Persone}) \\ = \\ \sigma_{\text{Età}> 30\ \vee\ \text{Età}\leq 30\ \vee\ \text{Età IS NULL}}(\text{Persone}) \\ = \\ \text{Persone}\end{gathered}$$

Combinando selezione e proiezione, possiamo estrarre interessanti informazioni da una relazione.

Proiezione $\pi_{\text{A, B}}(\text{R})$

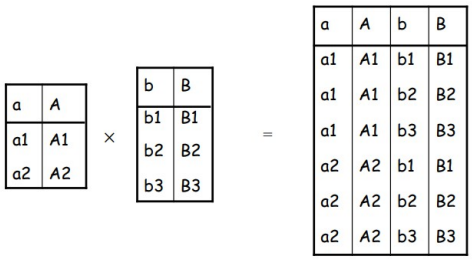


Restrizione $\sigma_{\text{Cond}}(\text{R})$



Prodotto

Prodotto: $R \times S$



a	A
a1	A1
a2	A2

×

b	B
b1	B1
b2	B2
b3	B3

=

a	A	b	B
a1	A1	b1	B1
a1	A1	b2	B2
a1	A1	b3	B3
a2	A2	b1	B1
a2	A2	b2	B2
a2	A2	b3	B3

Join (giunzione)

Combinando selezione e proiezione, possiamo estrarre informazioni da una relazione, ma *non* possiamo *correlare* informazioni presenti in relazioni diverse. Il **join** è l'operatore più interessante dell'algebra relazionale poiché permette di **correlare** dati in relazioni diverse.

Join naturale

Operatore binario (generalizzabile) che correla dati in relazioni diverse, **sulla base di valori uguali in attributi con lo stesso nome**. Produce un risultato:

- sull'unione degli attributi degli operandi;
- con ennuple che sono ottenute combinando le ennuple degli operandi con valori uguali sugli attributi in comune.

$R_1(X_1)$, $R_2(X_2)$
$R_1 \bowtie R_2$ è una relazione su $X_1 \cup X_2$.

$$R_1 \bowtie R_2 =$$

$$\{t \text{ } su \text{ } X_1 \cup X_2 \mid \text{esistono } t_1 \in R_1 \text{ e } t_2 \in R_2 \text{ con } t[X_1] = t_1 \text{ e } t[X_2] = t_2\}$$

Se ogni ennupla contribuisce al risultato: join completo.

Cardinalità del join

Il join di R_1 e R_2 contiene un numero di ennuple compreso fra zero e il prodotto di $|R_1|$ e $|R_2|$:

$$0 \leq |R_1 \bowtie R_2| \leq |R_1| \times |R_2|$$

Se il join fra R_1 ed R_2 è completo, allora contiene un numero di ennuple almeno uguale al massimo fra $|R_1|$ e $|R_2|$. Se il join coinvolge una chiave B di R_2, allora il numero di ennuple è compreso fra zero e $|R_1|$:

$$0 \leq |R_1 \bowtie R_2| \leq |R_1|$$

Se il join coinvolge una chiave B di R_2 e un vincolo di integrità referenziale tra attributi di R_1 ($B \in R_1$) e la chiave di R_2, allora il numero di ennuple è pari a $|\mathrm{R}_1|$:

$$|R_1 \bowtie R_2| = |R_1|$$

Join esterno

Il **join esterno** estende, con valori nulli, le ennuple che verrebbero tagliate fuori da un join (**interno**). Esiste in tre versioni: sinistro, destro, completo.

- Sinistro: mantiene tutte le ennuple del primo operando, estendendole con valori nulli, se necessario $R \overleftarrow{\bowtie} S$
- destro: ... del secondo operando... $R \overrightarrow{\bowtie} S$
- completo: ... di entrambi gli operandi ... $R \overleftrightarrow{\bowtie} S$

Nota bene:

$$\pi_{X_1}(R_1 \bowtie R_2) \subseteq R_1$$
$$R \supseteq (\pi_{X_1}(R)) \bowtie (\pi_{X_2}(R))$$

Prodotto cartesiano

Un join naturale su relazioni senza attributi in comune. Contiene sempre un numero di ennuple pari al prodotto delle cardinalità degli operandi (le ennuple sono tutte combinabili).

Poiché il prodotto cartesiano concatena tuple non necessariamente correlate dal punto di vista semantico, in pratica, ha senso solo se seguito da selezione:

$$\sigma_{\text{Condizione}}(R_1 \bowtie R_2)$$

L'operazione viene chiamata **theta-join** e può essere sintatticamente indicata con

$$R_1 \bowtie_{\text{Condizione}} R_2$$

Le due scritture sono equivalenti.

La condizione C è spesso una congiunzione ($\wedge$) di atomi di confronto $A_1 \ \vartheta \ A_2$, dove ϑ è uno degli operatori di confronto $(=, >, <, \ldots)$. Se l'operatore è sempre l'uguaglianza $(=)$ allora si parla di **equi-join**.

Self Join

Supponiamo di considerare la seguente relazione

Genitore	**Figlio**
Luca	Anna
Maria	Anna
Giorgio	Luca
Silvia	Maria
Enzo	Maria

e di volere ottenere una relazione Nonno-Nipote. È ovvio che in questo caso abbiamo bisogno di utilizzare due volte la stessa tabella. Tuttavia Genitore $\bowtie$ Genitore = Genitore, poiché tutti gli attributi coincidono. In questo caso è utile effettuare una ridenominazione:

$$\rho_{\text{Nonno, Genitore} \leftarrow \text{Genitore, Figlio}}(\text{Genitore})$$

A questo punto effettuando un *natural join* con la tabella Genitore, si ottiene l'informazione cercata.

$\rho_{\text{Nonno, Genitore} \leftarrow \text{Genitore, Figlio}}$ (Genitore) $\bowtie$ $\rho_{\text{Nipote} \leftarrow \text{Figlio}}$ (Genitore)

Nonno	Genitore
Luca	Anna
Maria	Anna
Giorgio	Luca
Silvia	Maria
Enzo	Maria

$\bowtie$

Genitore	Nipote
Luca	Anna
Maria	Anna
Giorgio	Luca
Silvia	Maria
Enzo	Maria

=

Nonno	Genitore	Nipote
Giorgio	Luca	Anna
Silvia	Maria	Anna
Enzo	Maria	Anna

Eventualmente si può effettuare una proiezione

$\pi_{\text{Nonno, Nipote}}$ ($\rho_{\text{Nonno, Genitore} \leftarrow \text{Genitore, Figlio}}$ (Genitore) $\bowtie$ $\rho_{\text{Nipote} \leftarrow \text{Figlio}}$ (Genitore))

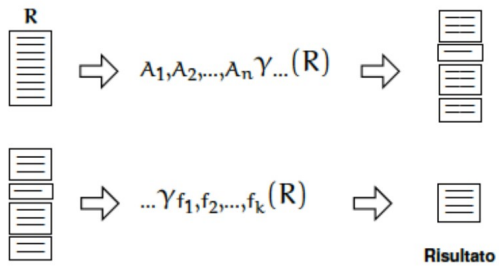
Nonno	Nipote
Giorgio	Anna
Silvia	Anna
Enzo	Anna

Raggruppamento

Raggruppamento: ${}_{\{A_i\}}\gamma_{\{f_i\}}(R)$

Gli A_i sono attributi di R e le f_i sono espressioni che usano funzioni di aggregazione (`min`, `max`, `count`, `sum`, ...). Il valore del raggruppamento è una relazione calcolata come segue:

- Si partizionano le ennuple di R mettendo nello stesso gruppo tutte le ennuple con valori uguali degli A_i.
- Si calcolano le espressioni f_i per ogni gruppo.
- Per ogni gruppo si restituisce una sola ennupla con attributi i valori degli A_i e delle espressioni f_i



4.1.2 Trasformazioni Algebriche

Basate su regole di equivalenza fra espressione algebriche, consentono di scegliere diversi ordini di join e di anticipare proiezioni e restrizioni. Alcuni esempi con la relazione $R(A, B, C, D)$:

$\pi_A(\pi_{A,B}(R)) \equiv \pi_A(R)$
$\sigma_{C_1}(\sigma_{C_2}(R)) \equiv \sigma_{C_1 \wedge C_2}(R)$
$\sigma_{C_1 \wedge C_2}(R \times S) \equiv \sigma_{C_1}(R) \times \sigma_{C_2}(S)$
$R \times (S \times T) \equiv (R \times S) \times T$
$(R \times S) \equiv (S \times R)$
${}_X\gamma_F(\sigma_C(R)) \equiv \sigma_C({}_X\gamma_F(R)$

Non distributività della proiezione rispetto alla differenza

In generale, se R_1 e R_2 sono definite su AB, e contengono tuple uguali su A e diverse su B.

$$\pi_A(R_1 - R_2) \neq \pi_A(R_1) - \pi_A(R_2)$$

Operatori algebrici non insiemistici

$\pi^b_{\{A_i\}}(R)$: proiezione multiinsiemistica (senza eliminazione dei duplicati)
$\tau_{\{Ai\}}(R)$: ordinamento

4.1.3 Calcolo relazionale su ennuple

Il calcolo relazionale è un linguaggio che permette di definire il risultato di un'interrogazione come l'insieme di quelle ennuple che soddisfano una certa condizione ϕ. L'insieme delle matricole degli studenti che hanno superato qualcuno degli esami elencati nella relazione Materie(Materia), si può definire come:

$$\{\text{t.matricola} \mid \text{t} \in \text{Studenti}, \exists \text{m} \in \text{Materie}.\ \exists \text{e} \in \text{ProveEsami}.\ \text{e.Candidato} = \text{t.Matricola} \wedge \text{e.Materia} = \text{m.Materia}\}$$

Che è equivalente a

$$\pi_{\text{Matricola}}(\text{Studenti} \bowtie_{\text{Matricola} = \text{Candidato}} (\text{ProveEsami} \bowtie \text{Materie}))$$

Capitolo 5

SQL

5.1 Interrogazione di una base dati

L'interrogazione di una base di dati è uno degli aspetti più importanti del linguaggio SQL. I comandi di interrogazione, o **QUERY**, permettono di effettuare una ricerca dei dati presenti nel database che soddisfano particolari condizioni richieste dall'utente.

SQL è stato definito nel 1973 ed è oggi il linguaggio universale dei sistemi relazionali. SQL è un calcolo su **multi-insiemi** e il suo comando base è

```
SELECT [DISTINCT] Attributo {Attributo}
  FROM Tabella [Ide] {Tabella [Ide]}
  [WHERE Condizione]
```

Semantica: prodotto + restrizione + proiezione. Un attributo A di una tabella "$R\ x$" si denota come A oppure $R.A$ oppure $x.A$.

Istruzione SELECT per l'interrogazione

Le query vengono effettuate mediante il comando `SELECT`. Sintassi:

```
SELECT ListaAttributi
FROM ListaTabelle
[ WHERE Condizione ]
```

Bisogna specificare:

- la "target list" cioè la lista degli attributi interessati;
- clausola `FROM` per stabilire in quale/quali tabella/e sono contenuti i dati che ci occorrono;

- clausola `WHERE` per esprimere le condizioni che i dati cercati devono soddisfare.

Target List (lista degli attributi)

Specificare la target list corrisponde a scegliere alcuni degli attributi della tabella o delle tabelle considerate. Implementa l'operazione di **proiezione** dell'algebra relazionale.

Se si desidera visualizzare tutti gli attributi della tabella, si può semplificare la target list indicando la lista con un semplice asterisco *.

Where

La clausola `WHERE` serve a scegliere le righe della tabella che soddisfano una certa condizione. In questo modo la clausola `where` implementa la **selezione** dell'algebra relazionale. Ovviamente la clausola `WHERE` è opzionale, e se si omette, si selezionano tutte le righe della tabella specificata.

From

La clausola `FROM` ha lo scopo di scegliere quali sono le tabelle del database da cui vogliamo estrarre le nostre informazioni. Nel caso in cui le tabelle elencate sono due, la clausola `FROM`, insieme con la clausola `WHERE`, che stabilisce quali righe delle due tabelle bisogna accoppiare, implementa il **theta join**.

Select

```
SELECT ListaAttributi
FROM ListaTabelle
[WHERE Condizione]
```

La query considera il prodotto cartesiano tra le tabelle in `ListaTabelle` (*join*). Fra queste seleziona solo le righe che soddisfano la `Condizione` (*selezione*) e infine valuta le espressioni specificate nella target list `ListaAttributi` (*proiezione*). La `SELECT` implementa quindi gli operatori di proiezione, selezione e join dell'algebra relazionale.

La lista degli attributi

Attributi ::= * | Expr [`[AS] nuovoNome`] {Expr [`[AS] nuovoNome`]}
Expr ::= [Ide.]Attributo | Const | (Expr) | [-] Expr [Op Expr] | `COUNT`(*) | AggrFun ([`DISTINCT`] [Ide.]Attributo)
e `AS x`: dà un nome a una colonna.
e AggrFun ::= `SUM` | `COUNT` | `AVG` | `MAX` | `MIN`
AggrFun: o si usano tutte funzioni di aggregazione (e si ottiene un'unica riga) o non se ne usa nessuna.

La lista delle tabelle

Le tabelle si possono combinare usando '`,`' (prodotto): `FROM` $\mathtt{T}_1, \mathtt{T}_2$.
Giunzioni di vario genere:

- Studenti s `JOIN` Esami e `ON` s.Matricola=e.Matricola
- Studenti s `JOIN` Esami e `USING` Matricola
- Studenti s `NATURAL JOIN` Esami e
- Studenti s `LEFT JOIN` Esami e `ON` s.Matricola=e.Matricola
- `LEFT JOIN - USING`
- `NATURAL LEFT JOIN`
- `RIGHT JOIN`
- `FULL JOIN`

La condizione

Combinazione booleana di predicati tra cui:

- Expr Comp Expr
- Expr Comp (Sottoselect che torna un valore)
- `[NOT] EXISTS` (Sottoselect)
- Espr Comp (`ANY` | `ALL`) (Sottoselect)
- Expr `[NOT] IN` (Sottoselect) (oppure `IN`($\mathtt{v}_1, \ldots, \mathtt{v}_n$))

Comp: $<$, $=$, $>$, $\neq$, $\leq$, $\geq$
Connettori logici `AND`, `OR`, `NOT`.

Operatori: `BETWEEN` consente la selezione di righe con attributi in un particolare range, `IN` è usato per selezionare righe che hanno un attributo che assume valori contenuti in una lista, `LIKE` è usato per effettuare ricerche "wildcard" (ossia con un simbolo jolly) di una stringa di valori. Le condizioni di ricerca possono contenere letterali, caratteri o numeri. Esistono due tipi di wildcard:

- '%' denota zero o più caratteri.
- '_' denota un carattere

A volte può succedere che uno dei simboli coinvolti nel pattern matching sia proprio '_' oppure '%'. In questo caso, si sceglie un simbolo che non è ammesso fra i simboli della stringa (supponiamo '#') detto simbolo escape, nell'espressione per la ricerca si fa precedere il simbolo '_' o '%' cercato dal simbolo escape, e poi si specifica che '#' è il simbolo di escape.

Il valore di un campo di un'ennupla può mancare per varie ragioni; SQL fornisce il valore speciale `NULL` per tali situazioni. La presenza del `NULL` introduce dei problemi:

- occorrono dei predicati per controllare se un valore è/non è `NULL`.
- la condizione "reddito > 8" è vera o falsa quando il reddito è uguale a `NULL`? Cosa succede degli operatori `AND`, `OR` e `NOT`?
- Occorre una logica a 3 valori (vero, falso e unknown).
- Va definita opportunamente la semantica dei costrutti. Ad esempio il `WHERE` elimina le ennuple che non rendono vera la condizione.

L'operatore `IS NULL` verifica se il contenuto di un operando è `NULL`.

Tutti gli operatori descritti sono presenti anche in forma negativa, con ovvio significato.

Alias delle colonne

L'alias serve a dare a una colonna un nome diverso rispetto a quello che è utilizzato nella definizione della tabella. Implementa l'operatore ρ (*rideno-minazione*) dell'algebra relazionale.

Può essere usata opzionalmente la parola chiave `AS` tra il nome della colonna e l'alias richiede necessariamente le virgolette se l'alias ha degli spazi.

5.2 Ordinamento, Operatori aggregati e Raggruppamento

5.2.1 Ordinamento del risultato

È possibile dare un ordinamento del risultato di una `Select`. L'ordinamento si può effettuare in base a un attributo e può essere crescente o decrescente. La sintassi è la seguente:

```
SELECT lista_attributi
FROM nome_tabella
WHERE condizioni
ORDER BY Attributo [ASC/DESC]
```

Le righe vengono ordinate in base al campo `Attributo` in maniera crescente o decrescente secondo se è data la specifica `ASC` o `DESC`. `ASC` è il default. Secondo il tipo dell'attributo, l'ordinamento è quello più naturale su quel dominio.

Si può voler ordinare i dati in base a una certa chiave (attributo) e poi ordinare i dati che coincidono su quella chiave in base a un'altra chiave (attributo).

5.2.2 Operatori aggregati

Nella target list possiamo avere anche espressioni che calcolano valori a partire da insiemi di ennuple e che restituiscono una tabella molto particolare, costituita da un singolo valore scalare. SQL-2 prevede 5 possibili operatori aggregati: Conteggio (`COUNT`), Minimo (`MIN`), Massimo(`MAX`), Media (`AVG`), Somma (`SUM`).

`COUNT` restituisce il numero di righe della tabella o il numero di valori di un particolare attributo. L'operatore aggregato viene applicato al risultato dell'interrogazione. Mediante le specifiche `(*)`, `ALL` e `DISTINCT` è possibile contare

- `(*)`: tutte le righe selezionate;
- `ALL`: tutti i valori non nulli delle righe selezionate;
- `DISTINCT`: tutti i valori non nulli distinti delle righe selezionate.

Se la specifica viene omessa, il default è `ALL`.

Le funzioni `MAX` e `MIN` calcolano rispettivamente il maggiore e il minore degli elementi di una colonna. L'argomento delle funzioni `MAX` e `MIN` può anche essere un'espressione aritmetica.

La funzione `SUM` calcola la somma dei valori di una colonna. Le specifiche `ALL` e `DISTINCT` permettono di sommare tutti i valori non nulli o tutti i valori distinti. Il default in mancanza di specifiche è `ALL`.

La funzione `AVG` calcola la media (average) dei valori *non nulli* di una colonna. Le specifiche `ALL` e `DISTINCT` servono a calcolare la media fra tutti i valori o tra i valori distinti. Il default è `ALL`. I valori nulli non vengono considerati nella media.
Non è possibile utilizzare in una stessa `SELECT` una proiezione su alcuni attributi della tabella considerata e operatori aggregati sulla stessa tabella: le funzioni di aggregazione non possono essere usate insieme ad attributi normali.

5.2.3 Raggruppamento

A volte può essere richiesto di calcolare operatori aggregati non per l'intera tabella, ma raggruppando le righe i cui valori coincidono su un certo attributo. In tal caso si può utilizzare la clausola `GROUP BY`. Il raggruppamento considera come gruppo anche le tuple che hanno sull'attributo di raggruppamento il valore `NULL`. La query è innanzitutto eseguita senza operatori aggregati e senza `GROUP BY`, quindi il risultato è diviso in sottoinsiemi aventi gli stessi valori per gli attributi indicati nel `GROUP BY`. Quindi l'operatore aggregato è calcolato su ogni sottoinsieme.

Osservazione: quando si effettua un raggruppamento, questo deve essere effettuato su *tutti gli elementi della target list che non sono operatori aggregati* (ossia sull'insieme degli attributi puri). Questo ha senso perché nel risultato deve apparire una riga per ogni "gruppo".

Gli attributi espressi non aggregati nella `SELECT` devono essere inclusi tra quelli citati nella `GROUP BY`.

Clausola HAVING

Si possono applicare condizioni sul valore aggregato per ogni gruppo. Si può realizzare mediante la clausola `HAVING`.

In generale se la condizione coinvolge un attributo, si usa la clausola `WHERE`, mentre se coinvolge un operatore aggregato si usa la clausola `HAVING`.

```
SELECT ... FROM ... WHERE ...
GROUP BY A1, ..., An [HAVING condizione]
```

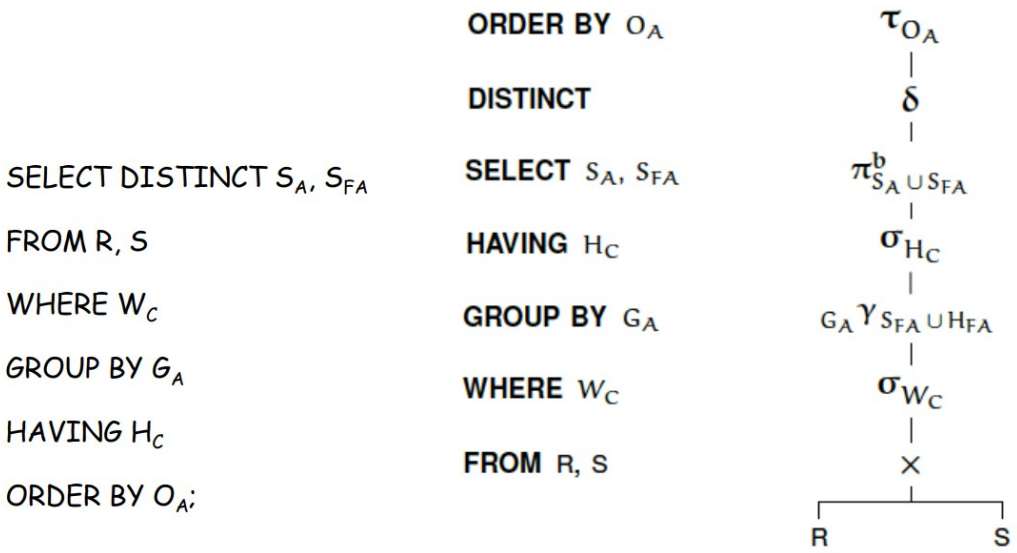
Semantica:

- Esegue le clausole `FROM` - `WHERE`
- Partiziona la tabella risultante rispetto all'uguaglianza su tutti i campi $A_1, \ldots, A_n$ (solo in questo caso, si assume `NULL` = `NULL`)
- Elimina i gruppi che non rispettano la clausola `HAVING`
- Da ogni gruppo estrae una riga usando la clausola `SELECT`

Attenzione: se la `SELECT` contiene sia espressioni aggregate (`MIN`, `COUNT`, ...) che attributi non aggregati, allora **DEVE** essere presente la clausola `GROUP BY`. Le clausole `HAVING` e `SELECT` citano solo:

- espressioni su attributi di raggruppamento;
- funzioni di aggregazione applicate ad attributi non di raggruppamento.

5.3 SQL e Algebra Relazionale



Confronto fra SQL e Algebra relazionale

Riferimento alle colonne

Spesso nel riferimento alle colonne selezionate nel join è necessario specificare da quale delle tabelle la colonna è stata estratta, al fine di evitare ambiguità. La sintassi usata è "`NomeTabella.NomeColonna`".

Ridenominazioni

Di solito in una `select` che definisce una join possono essere necessarie ridenominazioni

- nel prodotto cartesiano (ossia ridenominare le tabelle coinvolte),
- nella target list (ossia ridenominare gli attributi).

5.3.1 Join

Se due tabelle del database contengono dei dati in comune, possono essere correlate mediante un'operazione di *join*. Le colonne delle due tabelle che creano la correlazione rappresentano la stessa entità, ossia i loro valori appartengono allo **stesso dominio**. In genere le colonne delle due tabelle considerate sono legate da un vincolo di chiave esterna (ma non è obbligatorio).

Il **join** (o **equi-join**) fra due tabelle è una terza tabella le cui righe sono tutte e sole quelle ottenute dal prodotto cartesiano delle righe delle due tabelle di partenza i cui valori delle colonne espresse dalla condizione di join sono uguali. In SQL il join viene realizzato mediante una particolare forma del `SELECT`, in cui nella clausola `FROM` vengono indicate le due tabelle coinvolte e nella clausola `WHERE` viene espresso il collegamento fra le due tabelle, mediante la condizione di join.

$\texttt{Tab}_1(\texttt{A}_1, \texttt{A}_2)$ $\texttt{Tab}_2(\texttt{A}_3, \texttt{A}_4)$
  `SELECT` $\texttt{Tab}_1.\texttt{A}_1, \texttt{Tab}_2.\texttt{A}_4$
  `FROM` $\texttt{Tab}_1, \texttt{Tab}_2$
  `WHERE` $\texttt{Tab}_1.\texttt{A}_2 = \texttt{Tab}_2.\texttt{A}_3$

Traduce l'espressione dell'algebra relazionale

$$\pi_{\mathrm{A}_1,\mathrm{A}_4}(\sigma_{\mathrm{A}_2=\mathrm{A}_3}(\mathrm{Tab}_1 \bowtie \mathrm{Tab}_2))$$

Quindi il join consiste di un prodotto cartesiano (`FROM`), di una selezione (`WHERE`) e di una proiezione (`SELECT`).

La condizione di join può essere presente assieme ad altre condizioni mediante il connettore logico `AND`.

Se si vogliono selezionare tutte le colonne delle due tabelle si può sempre usare la notazione "`nome_tabella.*`".

Inner-Join È un'operazione di join in cui la condizione non sia necessariamente una condizione di uguaglianza.

Self-Join

Un caso molto particolare di Join è quello che mette in relazione una tabella con se stessa. Questo si può ottenere ridenominando due volte la tabella con due nomi diversi, trattando le due copie come se si trattasse di due tabelle diverse. In questo caso si parla di **Self Join**.

```
SELECT X.A1, Y.A4
FROM Tab1 X, Tab2 Y, Tab1 Z
WHERE X.A2 = Y.A3 AND X.A2 = Z.A1
```

Cross Join

Talvolta un'interrogazione può coinvolgere più di due tabelle → il **cross-join** implementa il prodotto cartesiano. Si realizza semplicemente mediante una select che utilizza le due (o più) tabelle coinvolte, senza specificare nessuna condizione di join.

Join On

Oltre alla forma vista, nei DBMS più moderni, per effettuare il join di due tabelle è possibile utilizzare una forma più esplicita (standard ANSI). La sintassi è la seguente:

```
SELECT Attributi
FROM Tab1 JOIN Tab2
ON CondizioneDiJoin
```

Equi-Join e Natural Join

Se dobbiamo operare una **equi-join**, ossia un join la cui condizione sia una condizione di uguaglianza, e che sia anche un **Natural Join**, ossia un join creato su *tutte* le colonne che hanno il medesimo nome in entrambe le tabelle, possiamo utilizzare la seguente sintassi:

```
SELECT listaAttributi
FROM Tab1 NATURAL JOIN Tab2
```

Using

Può succedere comunque che nelle tabelle coinvolte ci siano più attributi con lo stesso nome, e col `Natural Join` tutte queste coppie di attributi presi dalla due tabelle vengono identificate. Se invece vogliamo operare una join

in cui la condizione riguarda solo una o alcune di queste coppie, si usa la clausola `USING` seguita dall'elenco degli attributi coinvolti nella condizione:

`SELECT lista attributi`
`FROM` $\texttt{Tab}_1$ `JOIN` $\texttt{Tab}_2 2$
`USING` $(\texttt{attr}_1, \texttt{attr}_2, \ldots)$

Outer Join

Quando vengono correlate mediante una join delle tabelle con colonne contenenti dati in comune, è possibile che un valore sia presente in una delle colonne e non nell'altra. Effettuando un equi-join la riga corrispondente a tale valore viene scartato.

In alcuni casi invece può essere necessario mantenere questi valori. Per fare questo si deve effettuare un **outer join**.

`SELECT lista_attributi`
`FROM` $\texttt{Tab}_1$ `LEFT` [`OUTER`] `JOIN` $\texttt{Tab}_2$

`SELECT lista_attributi`
`FROM` $\texttt{Tab}_1$ `RIGHT` [`OUTER`] `JOIN` $\texttt{Tab}_2$

`SELECT lista_attributi`
`FROM` $\texttt{Tab}_1$ `FULL` [`OUTER`] `JOIN` $\texttt{Tab}_2$

L'operatore di `OUTER JOIN` può essere applicato usando la seguente sintassi:

- {`LEFT` | `RIGHT` | `FULL`} `[OUTER] JOIN` + `ON` ⟨predicato⟩
- {`LEFT` | `RIGHT` | `FULL`} `[OUTER] JOIN` + `USING` ⟨colonne⟩

dove '`OUTER`' è opzionale.

5.4 Sub-Query

Una **subquery** è un comando `Select`, racchiuso tra parentesi tonde, inserito all'interno di un comando SQL, per esempio un'altra `Select`. Le subquery possono essere utilizzate nei seguenti casi:

- in espressioni di confronto,
- in espressioni di confronto quantificato,

- in espressioni `IN`,
- in espressioni `EXISTS`,
- nel calcolo di espressioni.

Tre tipologie: scalare, di colonna, di tabella.
Subquery Scalare: è un comando `Select` che restituisce un solo valore.

```
SELECT Max(Cilindrata) FROM Veicoli
SELECT Cod_Categoria FROM Veicoli Where Targa=‘‘123456’’
```

Subquery di Colonna: è un comando `Select` che restituisce una colonna

```
SELECT cod_categoria FROM Veicoli
```

Subquery di Tabella: è un comando `Select` che restituisce una tabella

```
SELECT Targa, Cod_mod, Posti FROM Veicoli
```

Inoltre si ricorda che nelle `Select` semplici non è possibile utilizzare contemporaneamente funzioni di gruppo e funzioni su singole righe, questo viene reso possibile mediante l'uso delle subquery.

Quantificazione

Tutte le interrogazioni su di una associazione multivalore vanno quantificate.

- Universale negata = esistenziale
- Esistenziale negata = universale

Any, All ed Exists

Le condizioni in SQL permettono anche il confronto fra un attributo e il risultato di una subquery che restituisce una colonna o una tabella.
Operatore **Scalare** (`ANY` | `ALL`) SelectQuery

- `ANY`: il predicato è vero *se almeno uno dei valori restituiti* dalla query soddisfano la condizione.
- `ALL`: il predicato è vero se tutti i valori restituiti dalla query soddisfano la condizione.

In particolare le forme `= ANY` (equivalentemente `IN`) e $\neq$ `ALL` (equivalentemente `NOT IN`), forniscono un modo alternativo per realizzare intersezione e differenza dell'algebra relazionale.

Quantificatore **esistenziale**: `EXISTS` SelectQuery. Il predicato è vero se la SelectQuery *restituisce almeno una* ennupla. Mediante `EXIXST(SELECT*...)` è possibile verificare se il risultato di una subquery resituisce almeno una tupla. Facendo uso di `NOT EXISTS` il predicato è vero se la subquery non restituisce alcuna Tupla.

Occorre un meccanismo per rendere più flessibile la clausola `exists`, rendendo le condizioni della subquery dipendenti dalla tupla della query principale (query esterna), quindi si introduce un legame fra la query e la subquery (query correlate) e si definisce una variabile (un alias) nella query esterna che si utilizza nella subquery.

Semantica delle espressioni "correlate"

La query più interna può usare variabili della query esterna. L'interrogazione interna *viene eseguita una volta per ciascuna ennupla* dell'interrogazione esterna.

Le subquery oltre che essere usate all'interno della clausola `WHERE`, possono anche essere utilizzate nel calcolo di espressioni, dunque per definire colonne.

5.5 Unione, Intersezione, Differenza

A volte può essere utile poter ottenere un'unica tabella contenente alcuni dei dati contenuti in due tabelle omogenee, ossia con attributi definiti sullo stesso dominio. Per esempio: operazioni booleane, calcolando unione, intersezione e differenza.

In SQL la `SELECT` da sola non permette di fare questo tipo di operazioni su tabelle. Esistono per questo dei costrutti espliciti che utilizzano le parole chiave

`UNION`
`INTERSECT`
`EXCEPT` (in oracle `MINUS`)

Tali operatori lavorano sulle tabelle come se fossero insiemi di righe, dunque i duplicati vengono eliminati (a meno che si usi la specifica `ALL`) anche dalle proiezioni!

Unione

L'operatore `UNION` realizza l'operazione di unione definita nell'algebra relazionale. Utilizza come operandi le due tabelle risultanti da comandi `SELECT` e restituisce una terza tabella che contiene **tutte le righe della prima e della seconda tabella**. Nel caso in cui dall'unione e dalla proiezione risultassero delle righe duplicate, l'operatore `UNION` ne mantiene una sola copia, a meno che non sia specificata l'opzione `ALL` che indica la volontà di mantenere i duplicati

```
SELECT ...
UNION [ALL]
SELECT ...
```

Quali nomi per gli attributi del risultato?

- nessuno
- quelli del primo operando
- ...

In SQL quelli del primo operando.

Quanto detto riguardo alla *notazione posizionale* dell'operatore `UNION` vale equivalentemente per gli altri operatori booleani `EXCEPT` (`Minus` in Oracle) e `INTERSECT`.

Differenza

L'operatore `EXCEPT` utilizza come operandi due tabelle ottenute mediante due `select` e ha come risultato una nuova tabella che contiene **tutte le righe della prima che non si trovano nella seconda**: realizza la differenza dell'algebra relazionale. Anche qui si può specificare l'opzione `ALL` per indicare la volontà di mantenere i duplicati. Sintassi:

```
SELECT ...
EXCEPT [ALL]
SELECT ...
```

Nota che la differenza può essere effettuata da un'unica select che utilizza subselect.

Intersezione

L'operatore `INTERSECT` utilizza come operandi due tabelle risultanti dai comandi `SELECT` e restituisce una tabella che contiene le righe comuni alle due tabelle iniziali. Realizza l'intersezione dell'algebra relazionale. Sintassi:

```
SELECT ...
INTERSECT [ALL]
SELECT ...
```

L'opzione `ALL` serve a mantenere gli eventuali duplicati di righe. In sua assenza, si mantiene una sola copia delle righe duplicate.

5.6 SQL per la modifica di basi di dati

Introduciamo ora il **Data Manipulation Language** (**DML**) ossia il linguaggio SQL che serve per **inserire**, **modificare** e **cancellare** i dati del database, ma anche per **interrogare il database**, ossia estrarre i dati dal database.

Inizialmente descriveremo le istruzioni che servono a inserire, cancellare e modificare i dati. In seguito introdurremo le istruzioni per estrarre dal database le informazioni che ci interessano.

Insert

Supponiamo di volere inserire un nuovo dato in una tabella. Tale operazione si realizza mediante l'istruzione

`INSERT INTO ...VALUES`

Sintassi:

```
INSERT INTO nome tabella
[(ListaAttributi)] VALUES (ListaDiValori) |
SQLSelect
```

Ai valori non attribuiti viene assegnato `NULL`, a meno che non sia specificato un diverso valore di default. L'inserimento fallisce se `NULL` non è permesso per gli attributi mancanti. Non specificare gli attributi equivale a specificare tutte le colonne della tabella. **Nota bene**: si deve rispettare l'ordine degli attributi.

È possibile effettuare un insert prendendo i dati da un'altra tabella mediante il comando di interrogazione del database `SELECT`: con questo tipo di insert si possono effettuare tanti inserimenti simultaneamente.

Delete

Per eliminare un elemento bisogna individuare quale. Questo si può stabilire mediante la clausola `WHERE`, dove viene stabilita una condizione che individua l'elemento (o gli elementi) da cancellare. Spesso un particolare elemento può essere individuato mediante il suo valore nella chiave primaria.
Sintassi

```
DELETE FROM nome_tabella
[WHERE Condizione]
```

La condizione del delete può essere una normale condizione di `SELECT`. Questa modalità di delete permette di cancellare più righe con un'unica istruzione, purché le righe soddisfino la condizione.

Update

Inoltre è possibile aggiornare alcuni dati seguendo la seguente sintassi:

```
UPDATE Tabella
SET Attributo = Espr
WHERE Condizione
```

Capitolo 6

Data Definition Language(DDL)

Introduciamo il **Data Definition Language** (**DDL**) SQL, che consiste nell'insieme delle istruzioni SQL che permettono la creazione, modifica e cancellazione delle tabelle, dei domini e degli altri oggetti del database, al fine di definire il suo schema logico.

Le tabelle (corrispondenti alle relazioni dell'algebra relazionale) vengono definite in SQL mediante l'istruzione `CREATE TABLE`. Questa istruzione

- definisce uno schema di relazione e ne crea un'istanza vuota;
- specifica attributi, domini e vincoli.

L'istruzione `CREATE TABLE` è seguita da un nome che la caratterizza e dalla lista delle colonne (attributi) di cui si specificano le caratteristiche. Alla fine si possono anche specificare eventuali vincoli di tabella.

```
CREATE TABLE ⟨nome_tabella⟩
  nomeColonna_1 tipoColonna_1 clausolaDefault_1 vincoloDiColonna_1,
  ...
  nomeColonna_k tipoColonna_k clausolaDefault_k vincoloDiColonna_k,
  vincoli di tabella
```

L'effetto del comando `CREATE TABLE` è quello di definire uno schema di relazione e di crearne un'istanza vuota, specificandone attributi, domini e vincoli. Una volta creata, la tabella è pronta per l'inserimento dei dati che dovranno soddisfare i vincoli imposti.

La visualizzazione dello schema di una tabella, dopo che è stata creata, può essere ottenuta mediante il comando SQL `DESCRIBE`:

`DESCRIBE` ⟨`nomeTabella`⟩

Quindi SQL non è solo un linguaggio di interrogazione (Query Language), ma è anche un linguaggio per la definizione di basi di dati (Data-definition language (DDL)), un linguaggio per stabilire controlli sull'uso dei dati (GRANT) e un linguaggio per modificare i dati.

I tipi

I tipi più comuni per i valori degli attributi sono:

- `CHAR(n)` per stringhe di caratteri di lunghezza fissa `n`;
- `VARCHAR(n)` per stringhe di caratteri di lunghezza variabile di al massimo `n` caratteri;
- `INTEGER` per interi con la dimensione uguale alla parola di memoria standard dell'elaboratore;
- `REAL` per numeri reali con dimensione uguale alla parola di memoria standard dell'elaboratore;
- `NUMBER(p,s)` per numeri con `p` cifre, di cui `s` decimali;
- `FLOAT(p)` per numeri binari in virgola mobile, con almeno `p` cifre significative;
- `DATE` per valori che rappresentano istanti di tempo (in alcuni sistemi, come Oracle), oppure solo date (e quindi insieme ad un tipo `TIME` per indicare ora, minuti e secondi).

Modificare una tabella

Ciò che si crea con un `CREATE` si può eliminare con il comando `DROP` o cambiare con il comando `ALTER`.

```
CREATE TABLE Nome
  (Attributo Tipo [ValoreDefault] [VincoloAttributo]
    {Attributo Tipo [Default] [VincoloAttributo]},
    {VincoloTabella})
```

Default := `DEFAULT` valore | null | username

Nuovi attributi si possono aggiungere con:

```
ALTER TABLE Nome ADD COLUMN NuovoAttr Tipo
```

Con il comando `ALTER TABLE` è possibile (standard SQL):

1. Aggiungere una colonna (`ADD [COLUMN]`)
2. Eliminare una colonna (`DROP [COLUMN]`)
3. Modificare la colonna (`MODIFY`)
4. Aggiungere l'assegnazione di valori di default (`SET DEFAULT`)
5. Eliminare l'assegnazione di valori di default (`DROP DEFAULT`)
6. Aggiungere vincoli di tabella (`ADD CONSTRAINT`)
7. Eliminare vincoli di tabella (`DROP CONSTRAINT`)
8. Altre opzioni sono possibili nei linguaggi specifici

Aggiungere una colonna

Si può aggiungere una colonna in qualsiasi momento se non viene specificato `NOT NULL`. Sintassi:

```
ALTER TABLE nome_tabella
 ADD [COLUMN] nome_col tipo_col default_col vincolo_col
```

In mancanza di altre specifiche, la nuova colonna viene inserita come ultima colonna della tabella. Altrimenti è possibile dare questa specifica:

```
ADD COLUMN ⟨creaDefinizione⟩
  [FIRST/AFTER ⟨nomeColonna⟩]
```

`FIRST` permette di aggiungerla come prima colonna, mentre `AFTER` come colonna subito dopo la colonna indicata.

Eliminare una colonna

```
ALTER TABLE nome_tabella
DROP COLUMN nome_colonna {RESTRICT/CASCADE}
```

In SQL standard le opzioni `RESTRICT/CASCADE` sono alternative ed è obbligatorio specificare l'una o l'altra.

- `RESTRICT`: se un'altra tabella si ha un vincolo di integrità referenziale con questa colonna, l'esecuzione del comando drop fallisce.
- `CASCADE`: eliminando la colonna, vengono eliminate tutte le dipendenze logiche di altre colonne dello schema da questa.

Modificare una colonna

Se si vogliono modificare le caratteristiche di una colonna dopo averla definita, occorre eseguire l'istruzione:

```
ALTER TABLE nome_tabella MODIFY
nome colonna tipo_col default_col vincoli_col
```

Assegnare un valore di default

Nell'SQL standard è possibile imporre un valore di default col comando specifico `SET DEFAULT`, con la seguente sintassi

```
ALTER TABLE nome_tabella
ALTER [COLUMN] nome_colonna
SET DEFAULT valore_default
```

Eliminare un valore di default

In SQL standard è possibile eliminare un vincolo di default da una colonna mediante l'istruzione:

```
ALTER TABLE nome_tabella
ALTER [COLUMN] nome_colonna
DROP DEFAULT
```

Eseguendo questa istruzione il valore di default diventa automaticamente `NULL`.

Aggiungere vincoli di tabella

Se si vuole aggiungere un vincolo di tabella, si esegue il comando

```
ALTER TABLE nome_tabella
ADD CONSTRAINT nome_vincolo vincolo_di_tabella
```

Nota bene: Occorre assegnare un nome al vincolo.

Eliminare vincoli di tabella

Nello standard SQL, se si vuole eliminare un vincolo di tabella si esegue l'istruzione

```
ALTER TABLE nome_tabella
DROP CONSTRAINT nome_vincolo{RESTRICT/CASCADE}
```

L'opzione `RESTRICT` non permette di eliminare vincoli di unicità e di chiave primaria su una colonna se esistono vincoli di chiave esterna che si riferiscono a tale colonna. L'opzione `CASCADE` non opera questa restrizione.

Da notare che per eliminare un vincolo, esso deve essere definito mediante un identificatore.

Drop Table

Si può eliminare una tabella mediante l'istruzione `DROP TABLE`. Nello standard SQL si possono anche specificare le opzioni `RESTRICT/CASCADE`

- `RESTRICT`: se la tabella è utilizzata nella definizione di altri oggetti dello schema, la sua eliminazione viene impedita.
- `CASCADE`: vengono eliminate tutte le dipendenze degli altri oggetti dello schema da questa tabella.

6.1 I vincoli

I **vincoli di integrità** consentono di limitare i valori ammissibili per una determinata colonna della tabella in base a specifici criteri. I vincoli di integrità intrarelazionali (ossia che non fanno riferimento ad altre relazioni) sono:

- `NOT NULL`
- `UNIQUE` definisce chiavi
- `PRIMARY KEY`: chiave primaria (una sola, implica `NOT NULL`)
- `CHECK`

6.1.1 UNIQUE

Può essere espresso in due forme: nella definizione di un attributo, se forma da solo la chiave, oppure come elemento separato. Il vincolo `unique` utilizzato nella definizione dell'attributo indica che non ci possono essere due valori uguali in quella colonna. È una chiave della relazione, ma non una chiave primaria.

Vincolo unique per insiemi di attributi

Il vincolo di unicità può anche essere riferito a coppie o insiemi di attributi. Ciò non significa che per gli attributi dell'insieme considerato ogni singolo valore deve apparire una sola volta, ma che non ci siano due dati (righe) per cui l'insieme dei valori corrispondenti a quegli attributi siano uguali.

In questo caso il vincolo viene dichiarato dopo aver dichiarato tutte le colonne mediante un vincolo di tabella, utilizzando il comando

```
Unique (lista_attributi)
```

6.1.2 Primary key

Due forme: definizione di un attributo, se formato da solo la chiave oppure come elemento separato. Il vincolo `primary key` è simile a `unique`, ma definisce la chiave primaria della relazione, ossia un attributo che individua univocamente un dato.

Implica sia il vincolo `unique` che il vincolo `not null` (non è ammesso che per uno degli elementi della tabella questo valore sia non definito). Serve ad identificare univocamente i soggetti del dominio. Questo vincolo permette spesso il collegamento fra due tabelle.

Chiave primaria con insiemi di attributi

Analogamente al vincolo `unique`, anche il vincolo di chiave primaria può essere definito su un insieme di elementi. In tal caso la sintassi è simile a quella di `unique`:

```
Primary key (lista_attributi)
```

6.1.3 Vincoli interrelazionali

I vincoli interrelazionali sono quei vincoli che vengono imposti quando gli *attributi di due diverse tabelle devono essere messi in relazione*. Questo è fatto per soddisfare l'esigenza di un database di **non essere ridondante** e di avere i **dati sincronizzati**. Se due tabelle gestiscono gli stessi dati, è bene che di essi non ce ne siano più copie, sia allo scopo di non occupare troppa memoria, sia affinché le modifiche fatte su dati uguali utilizzati da due tabelle siano coerenti.

References e **Foreign Key** permettono di definire vincoli di integrità referenziale. Di nuovo **due sintassi**: per singoli attributi (come vincolo di colonna), oppure su più attributi (come vincolo di tabella). È possibile definire

politiche di **reazione alla violazione** (ossia stabilire l'azione che il DBMS deve compiere quando si viola il vincolo).

6.1.4 Check

Un vincolo di `Check` richiede che una colonna, o una combinazione di colonne, soddisfi una condizione per ogni riga della tabella: deve essere una espressione booleana che è valutata usando i valori della colonna che vengono inseriti o aggiornati nella riga. Può essere espresso sia come **vincolo di riga** che come **vincolo di tabella**. Se è espresso come **vincolo di riga**, può coinvolgere solo l'attributo su cui è definito, mentre se serve eseguire un check che coinvolge due o più attributi, si deve definire come **vincolo di tabella**.

6.1.5 Reazione alla violazione

Quando si crea un vincolo `foreign key` in una tabella, in SQL standard si può specificare l'azione da intraprendere quando delle righe nella tabella riferita vengono cancellate o modificate. Tali reazioni alla violazione vengono dichiarate al momento della definizione dei vincoli di `foreign key` rispettivamente mediante i comandi

- `On Delete`
- `On Update`

Reazioni alla violazione On Delete

Impedire il delete (`No Action`): blocca il delete delle righe dalla tabella riferita quando ci sono righe che dipendono da essa. Questa è l'azione che viene attivata per *default*.
Generare un delete a catena (`Cascade`): cancella tutte le righe dipendenti dalla tabella quando la corrispondente riga è cancellata dalla tabella riferita.
Assegnare valore `NULL` (`Set Null`): assegna `NULL` ai valori della colonna che ha il vincolo foreign key nella tabella quando la riga corrispondente viene cancellata dalla tabella riferita.
Assegnare il valore di default (`Set Default`): assegna il valore di default ai valori della colonna che ha il vincolo foreign key nella tabella quando la riga corrispondente viene cancellata dalla tabella riferita.

Reazioni alla violazione On Update

Nello standard SQL la reazione alla violazione può anche essere attivata quando i dati della tabella riferita vengono aggiornati. Viene attivato mediante il comando `ON UPDATE` seguito da:

- `Cascade`: Le righe della tabella referente vengono impostati ai valori della tabella riferita.
- `Set Null`: i valori della tabella referente vengono impostati a `NULL`.
- `Set Default`: i valori della tabella referente vengono impostati al valore di default.
- `No Action`: rifiuta gli aggiornamenti che violino l'integrità referenziale.

6.2 Viste

Le **viste logiche** o **viste** o **view** possono essere definite come delle tabelle virtuali, i cui dati sono riaggregazioni dei dati contenuti nelle tabelle "fisiche" presenti nel database. Le tabelle fisiche sono gli unici veri contenitori di dati. Le viste non contengono dati fisicamente diversi dai dati presenti nelle tabelle, ma forniscono una *diversa visione*, *dinamicamente aggiornata*, di quegli stessi dati. La vista appare all'utente come una normale tabella, in cui può effettuare **interrogazioni** e, limitatamente ai suoi privilegi, anche **modificare dei dati**.
Vantaggi

- Le viste **semplificano la rappresentazione dei dati**. Oltre ad assegnare un nome alla vista, la sintassi dell'istruzione `CREATE VIEW` consente di cambiare i nomi delle colonne.
- Le viste possono essere anche estremamente **convenienti per svolgere una serie di query molto complesse**.
- Le viste consentono di **proteggere i database**: le view ad accesso limitato possono essere utilizzate per controllare le informazioni alle quali accede un certo utente del database.
- Le viste consentono inoltre di **convertire le unità di misura e creare nuovi formati**.

Limitazioni

- Non è possibile utilizzare gli operatori booleani `UNION`, `INTERSECT` ed `EXCEPT`.
- Gli operatori `INTERSECT` ed `EXCEPT` possono essere realizzati mediante una select semplice. La stessa cosa non si può dire dell'operatore `UNION`.
- Non è possibile utilizzare la clausola `ORDER BY`.

Sintassi

Il comando DDL che consente di definire una vista ha la seguente sintassi

```
CREATE VIEW NomeVista [(ListaAttributi)] AS SelectSQL
[with [local | cascaded] checkoption]
```

I nomi delle colonne indicati nella lista attributi sono i nomi assegnati alle colonne della vista, che corrispondono ordinatamente alle colonne elencate nella select. Se questi non sono specificati, le colonne della vista assumono gli stessi nomi di quelli della/e tabella/e a cui si riferisce.

Modifica di una vista

Sebbene il **contenuto** di una vista sia **dinamico**, *la sua struttura non lo è*. Se una vista è definita su una subquery

$$\texttt{Select}^{*}\ \texttt{From}\ \texttt{T}_1$$

e in seguito alla tabella $\texttt{T}_1$ viene aggiunta una colonna, questa nuova definizione non si estende alla vista: la vista conterrà sempre le stesse colonne che aveva prima dell'inserimento della nuova colonna in $\texttt{T}_1$.

Viste di gruppo

Una **vista di gruppo** è una vista di cui una delle colonne è una funzione di gruppo. In questo caso è obbligatorio assegnare un nome alla colonna della vista corrispondente alla funzione.

È una vista di gruppo anche una vista che è definita in base ad una vista di gruppo.

Eliminazione delle vista

Le viste si eliminano col comando `Drop View`. Sintassi:

`Drop View nomeView {Restrict/Cascade}`

`Restrict`: la vista viene eliminata solo se non è riferita nella definizione di altri oggetti.
`Cascade`: oltre che essere eliminata la vista, vengono eliminate tutte le dipendenza da tale vista di altre definizioni dello schema.

Viste modificabili

Le tabelle delle viste si interrogano come le altre, ma in generale non si possono modificare. Deve esistere una corrispondenza biunivoca fra le righe della vista e le righe di una tabella di base, ovvero:

1. `SELECT` senza `DISTINCT` e solo di attributi
2. `FROM` una sola tabella modificabile
3. `WHERE` senza SottoSelect
4. `GROUP BY` e `HAVING` non sono presenti nella definizione.

Possono esistere anche delle restrizioni su `SELECT` su viste definite usando `GROUP BY`.

Aggiornamento delle viste

Le operazioni `INSERT/UPDATE/DELETE` sulle viste non erano permesse nelle prime edizioni di SQL. I nuovi DBMS permettono di farlo con certe limitazioni dovute alla definizione della vista stessa.

Ha senso aggiornare una vista? Dopotutto si potrebbe aggiornare la tabella di base direttamente... tuttavia è molto utile nel caso di accesso dati controllato.

L'opzione `With Check Option` messa alla fine della definizione di una vista assicura che le operazioni di inserimento e di modifica dei dati effettuate utilizzando la vista soddisfino la clausola `Where` della subquery.

Supponiamo che una **vista $\mathbf{V_1}$** sia definita in termini di un'altra vista V_2. Se si crea V_1 specificando la clausola `With Check Option`, il DBMS verifica che la nuova tupla `t` inserita soddisfi **sia la definizione di $\mathbf{V_1}$ che quella di $\mathbf{V_2}$** (e di tutte le altre eventuali viste da cui V_1 dipende), **indipendentemente** dal fatto che V_2 sia stata a sua volta definita `With Check Option`. Questo comportamento di default è equivalente a definire V_1

```
WITH CASCADED CHECK OPTION
```

Lo si può alterare definendo V_1

```
WITH LOCAL CHECK OPTION
```

Ora il DBMS verifica solo che `t` soddisfi la specifica di V_1 e quelle di tutte e **sole le viste da cui V_1 dipende per cui è stata specificata** la clausola `With Check Option`.

6.2.1 Vantaggi delle viste

Facilitazione nell'accesso ai dati

In generale uno dei requisiti per la progettazione di un database relazionale è la **normalizzazione dei dati**. Sebbene la forma normalizzata del database permetta una corretta modellazione della realtà che il DB rappresenta, a volte dal punto di vista dell'utente comporta una *maggiore difficoltà di comprensione* rispetto a una rappresentazione non normalizzata. Le viste permettono di fornire all'utente i dati in una forma *più intuitiva*.

Diverse visioni dei dati

Esistono dei dati che sono presenti nelle tabelle del database, che sono **poco significativi per l'utente** e altri che **devono essere nascosti all'utente**. L'uso delle viste da parte dell'utente permette di **limitare il suo accesso ai dati del database**, eliminando quelli non interessanti per lui e quelli che devono essere tenuti nascosti. L'uso delle viste può essere considerato come una **tecnica per assicurare la sicurezza dei dati**.

Indipendenza logica

Un vantaggio delle viste riguarda l'**indipendenza logica** delle applicazioni e delle operazioni eseguite dagli utenti rispetto alla struttura logica dei dati. Ciò significa che è possibile poter operare *modifiche allo schema senza dover apportare modifiche alle applicazioni* che utilizzano il database.

Utilità delle viste

- Per nascondere certe modifiche all'organizzazione logica dei dati (indipendenza logica)
- Per offrire visioni diverse degli stessi dati senza ricorrere a duplicazioni
- Per rendere più semplici, o per rendere possibili, alcune interrogazioni

6.3 Procedure e Trigger

6.3.1 Trigger

Un **trigger** definisce un'azione che il database deve attivare automaticamente quando si verifica (nel database) un determinato evento. Possono essere utilizzati

- per **migliorare l'integrità** referenziale dichiarativa,
- per **imporre regole complesse** legate all'attività del database,
- per **effettuare revisioni** sulle modifiche dei dati.

L'esecuzione dei trigger è quindi trasparente all'utente. I trigger vengono eseguiti automaticamente dal database quando specifici tipi di comandi (**eventi**) di manipolazione dei dati vengono eseguiti su specifiche tabelle. Tali comandi comprendono i comandi DML `insert`, `update` e `delete`, ma gli ultimi DBMS prevedono anche trigger su istruzioni DDL come `Create View`. Anche gli aggiornamenti di specifiche colonne possono essere utilizzati come trigger di eventi.

Trigger a livello di riga

I trigger a livello di riga vengono eseguiti una volta per ciascuna riga modificata in una transazione; vengono spesso utilizzati in applicazioni di revisione dei dati e si rivelano utili per **operazioni di audit dei dati e per mantenere sincronizzati i dati distribuiti**. Per creare un trigger a livello di riga occorre specificare la clausola

```
FOR EACH ROW
```

nell'istruzione `create trigger`.

Trigger a livello di istruzione

I trigger a livello di istruzione vengono eseguiti **una sola volta per ciascuna transazione**, indipendentemente dal numero di righe che vengono modificate (quindi anche se, ad esempio, in una tabella vengono inserite 100 righe, il trigger verrà eseguito solo una volta). Vengono pertanto utilizzati per **attività correlate ai dati**: vengono utilizzati di solito per **imporre misure aggiuntive di sicurezza sui tipi di transazione che possono essere eseguiti su una tabella**.

È il tipo di trigger *predefinito* nel comando `create trigger` (ossia non occorre specificare che è un trigger al livello di istruzione).

Struttura

I trigger si basano sul paradigma evento-condizione-azione (ECA).
L'istruzione `Create Trigger` seguita dal **nome** assegnato al trigger.
Tipo di trigger, `Before/After`.
Evento che scatena il trigger `Insert/Delete/Update`.
`For each row` se si vuole specificare trigger al livello di riga (altrimenti nulla per trigger al livello di istruzione).
Specificare a quale **tabella** si applica.
Condizione che si deve verificare perché il trigger sia eseguito.
Azione, definita dal codice da eseguire se si verifica la condizione.

Tipi di Trigger

`Before` e `After`: i trigger possono essere eseguiti prima o dopo l'utilizzo dei comandi `insert`, `update` e `delete`; all'interno del trigger è possibile fare riferimento ai vecchi e nuovi valori coinvolti nella transizione. Occorre utilizzare la clausola

`Before/After` ⟨`tipoDiEvento`⟩

Se si tratta di un trigger `before update`:

- per valori **vecchi** intendiamo *i valori che sono nella tabella* e che vogliamo modificare,
- per **nuovi** quelli che *vogliamo inserire* al posto di quelli vecchi.

Se si tratta di un trigger `after update`

- per **vecchi** intendiamo quelli *che c'erano prima* dell'update,
- per **nuovi** quelli *presenti nella tabella alla fine della modifica.*

Un **trigger** è **attivo** quando, in corrispondenza di certi eventi, *modifica lo stato della base di dati.* Un **trigger** è **passivo** se serve a *provocare il fallimento* della transazione corrente sotto certe condizioni.
Instead Of: per specificare che cosa fare invece di eseguire le azioni che hanno attivato il trigger. Ad esempio, è possibile utilizzare un trigger `INSTEAD OF` per reindirizzare le `INSERT` in una tabella verso una tabella differente o per aggiornare con update più tabelle che siano parte di una vista.

I trigger `instead-of` possono essere definiti su viste (relazionali od oggetto). I trigger instead-of devono essere a livello di riga.

6.4 Controllo degli accessi

Ogni componente dello schema (risorsa) può essere protetto (tabelle, attributi, viste, domini, ...). Il possessore della risorsa (colui che la crea) assegna dei privilegi agli altri utenti; un utente predefinito (`_system`) rappresenta l'amministratore della base di dati ed ha completo accesso alle risorse. Ogni privilegio è caratterizzato dalla risorsa a cui si riferisce, dall'utente che concede il privilegio, dall'utente che riceve il privilegio, dall'azione che viene permessa sulla risorsa e se il privilegio può esser trasmesso o meno ad altri utenti. Tipi di privilegi:

- `SELECT`: lettura di dati.
- `INSERT [(Attributi)]`: inserire record (con valori non nulli per gli attributi).
- `DELETE`: cancellazione di record.
- `UPDATE [(Attributi)]`: modificare record (o solo gli attributi).
- `REFERENCES [(Attributi)]`: definire chiavi esterne in altre tabelle che riferiscono gli attributi.
- `WITH GRANT OPTION`: si possono trasferire i privilegi ad altri utenti.

Chi crea lo schema del DB è l'unico che può fare `CREATE`, `ALTER` e `DROP`. Chi crea la tabella stabilisce i modi in cui altri possono farne uso:

```
GRANT Privilegi ON Oggetto TO Utenti [WITH GRANT OPTION]
```

Per revocare il privilegio:

```
revoke Privileges on Resource from Users [restrict | cascade]
```

La revoca deve essere fatta dall'utente che aveva concesso i privilegi: `restrict` (di default) specifica che il comando non deve essere eseguito qualora la revoca dei privilegi all'utente comporti qualche altra revoca (dovuta ad un precedente grant option), `cascade` invece forza l'esecuzione del comando.

Chi definisce una tabella o una view ottiene automaticamente tutti i privilegi su di esse ed è l'unico che può fare `DROP` e può autorizzare altri ad usarla con `GRANT`. Nel caso di viste il "creatore" ha i privilegi che ha sulle tabelle usate nella definizione.

6.5 Indice e catalogo

Creazione di indici

Non è un comando standard dell'SQL e quindi ci sono differenze nei vari sistemi.

```
CREATE INDEX NomeIdx ON Tabella(Attributi)
CREATE INDEX NomeIdx ON Tabella
  WITH STRUCTURE = BTree, KEY = (Attributi)
DROP INDEX NomeIdx
```

Catalogo (dei metadati)

Alcuni esempi di tabelle, delle quali si mostrano solo alcuni attributi, sono:

- Tabella delle password: `PASSWORD(username,password)`.
- Tabella delle basi di dati: `SYSDB(dbname,creator,dbpath,remarks)`.
- Tabella delle tabelle (type = view or table): `SYSTABLES(name,creator,type,colcount,filename,remarks)`.
- Tabella degli attributi: `SYSCOLUMNS(name,tbname,tbcreator,colno,coltype,lenght,default,remarks)`.
- Tabella degli indici: `SYSINDEXES(name,tbname,creator,uniquerule,colcount)`.

E altre ancora sulle viste, vincoli, autorizzazioni, ... (una decina).

Capitolo 7

Programmazione e SQL

7.1 Uso di SQL da programmi

Problemi

- Come collegarsi alla BD?
- Come trattare gli operatori SQL?
- Come trattare il risultato di un comando SQL (relazioni)?
- Come scambiare informazioni sull'esito delle operazioni?

Approcci

Linguaggio integrato (dati e DML): linguaggio disegnato ad-hoc per usare SQL. I comandi SQL sono controllati staticamente dal traduttore ed eseguiti dal DBMS.
Linguaggio convenzionale + API: linguaggio convenzionale che usa delle funzioni di libreria predefinita per usare SQL. I comandi SQL sono stringhe passate come parametri alle funzioni che poi vengono controllate dinamicamente dal DBMS prima di eseguirle.
Linguaggio che ospita l'SQL: linguaggio convenzionale esteso con un nuovo costrutto per marcare i comandi SQL. Occorre un **pre-compilatore** che controlla i comandi SQL, li sostituisce con chiamate a funzioni predefinite e genera un programma nel linguaggio convenzionale + API.

7.1.1 Un linguaggio integrato: PL/SQL

Un linguaggio per manipolare basi di dati che integra DML (SQL) con il linguaggio ospite. Un linguaggio a blocchi con una struttura del controllo completa che contiene l'SQL come sottolinguaggio. Permette:

- di definire variabili di tipo scalare, record (annidato), insieme di scalari, insiemi di record piatti, cursore;
- di definire i tipi delle variabili a partire da quelli della base di dati;
- di eseguire interrogazioni SQL ed esplorarne il risultato;
- di modificare la base di dati;
- di definire procedure e moduli;
- di gestire il flusso del controllo, le transazioni, le eccezioni.

Cursore

È il meccanismo per ottenere uno alla volta gli elementi di una relazione. Un cursore viene definito come un'espressione SQL, poi si apre per far calcolare al DBMS il risultato e infine con un opportuno comando si trasferiscono i campi delle ennuple in opportune variabili del programma.

7.1.2 Linguaggio con interfaccia API

Invece di modificare il compilatore di un linguaggio, si usa una libreria di funzioni/oggetti che operano su basi di dati (API) alle quali si passa come parametro stringhe SQL e ritornano il risultato sul quale si opera con una logica ad iteratori. Esempio:

- Microsoft ODBC è C/C++ standard su Windows
- Sun JDBC è l'equivalente in Java

Dorebbero essere indipendenti dal DBMS: un "driver" gestisce le richieste e le traduce in un codice specifico di un DBMS. Il DB può essere in rete.

7.1.3 SQLJ: Java che ospita l'SQL

Dialetto di SQL che può essere immerso in programmi Java, che poi vengono tradotti da un precompilatore in programmi Java standard sostituendo i comandi SQL in chiamate di una libreria che usano JDBC.

Capitolo 8

La Normalizzazione

8.1 Teoria Relazionale

Due metodi per produrre uno schema relazionale:

- partire da un buon schema a oggetti e tradurlo,
- partire da uno schema relazionale fatto da altri e modificarlo o completarlo.

Teoria della progettazione relazionale: studia cosa sono le "anomalie" e come eliminarle (**normalizzazione**). È particolarmente utile se si usa il secondo metodo. È moderatamente utile anche quando si usa il primo.

Attenzione alle anomalie: ridondanze, potenziali inconsistenze e anomalie nelle inserzioni/eliminazioni.

Obiettivi

Obiettivi della teoria:

- **Equivalenza** di schemi: in che misura si può dire che uno schema ne rappresenta un altro.
- **Qualità** degli schemi (forme normali).
- **Trasformazione** degli schemi (normalizzazione di schemi).

Ipotesi dello **schema di relazione universale**: tutti i fatti sono descritti da attributi di un'unica relazione (**relazione universale**), cioè gli attributi hanno un significato globale. Definizione: lo schema di **relazione universale**

U di una base di dati relazionale ha come attributi l'unione degli attributi di tutte le relazioni della base di dati.

Una **forma normale** è una proprietà di una base di dati relazionale che ne garantisce la "qualità", cioè l'assenza di determinati difetti. Quando una relazione *non è normalizzata*: presenta ridondanze e si presta a comportamenti poco desiderabili durante gli aggiornamenti. La normalizzazione è una procedura che permette di trasformare schemi non normalizzati in schemi che soddisfano una forma normale.

Linee guida per una corretta progettazione

- Semantica degli attributi: si progetti ogni schema relazionale in modo tale che **sia semplice spiegarne il significato**. Non si uniscano attributi provenienti da più tipi di classi e tipi di associazione in una unica relazione.

- Ridondanza: si progettino gli schemi relazionale in modo che nelle relazioni **non siano presenti anomalie di inserimento, cancellazione o modifica**. Se sono presenti delle anomalie (che si vuole mantenere), le si rilevi chiaramente e *ci si assicuri* che i programmi che aggiornano la base di dati operino correttamente.

- Valori nulli: per quanto possibile, **si eviti di porre in relazione di base attributi i cui valori possono essere (frequentemente) nulli**. Se è inevitabile, ci si assicuri che essi si presentino solo in casi eccezionali e che non riguardino una maggioranza di tuple nella relazione.

- Tuple spurie: si progettino schemi di relazione in modo tale che essi possano essere riuniti tramite `JOIN` con condizioni di uguaglianza su attributi che sono o **chiavi primarie** o **chiavi esterne** in modo da garantire che non vengano generate tuple spurie. *Non si abbiano relazioni che contengono attributi di "accoppiamento" diversi dalle combinazioni chiave esterna-chiave primaria.*

8.1.1 Dipendenze funzionali

Per formalizzare la nozione di schema senza anomalie, occorre una descrizione formale della semantica dei fatti rappresentati in uno schema relazionale. **Istanza valida di R**: è una **nozione semantica** che dipende da ciò che sappiamo del dominio del discorso (non estensionale, non deducibile da alcune istanze dello schema).

Istanza valida r su $R(T)$. Siano X e Y due sottoinsiemi non vuoti di T. Esiste in r una **dipendenza funzionale** da X a Y se, per ogni coppia di ennuple t_1 e t_2 di r con gli stessi valori su X, risulta che t_1 e t_2 hanno gli stessi valori anche su Y. La dipendenza funzionale da X a Y si denota con $X \to Y$.

Formalmente

Dato uno schema $R(T)$ e $X, Y \subseteq T$, una **dipendenza funzionale** (DF) fra gli attributi X e Y, è un vincolo su R sulle istanze della relazione, espresso nella forma $X \to Y$, cioè X *determina funzionalmente* Y o Y *è determinato da* X, se per ogni istanza valida r di R un valore di X determina in modo univoco un valore di Y:

$\forall$ r istanza valida di R, $\forall$ t_1, $t_2 \in r$. se $t_1[X] = t_2[X]$ allora $t_1[Y] = t_2[Y]$.

In altre parole: un'istanza r di R(T) soddisfa la dipendenza $X \to Y$ (o $X \to Y$ vale in r), se per ogni coppia di ennuple t_1 e t_2 di r, se $t_1[X] = t_2[X]$ allora $t_1[Y] = t_2[Y]$.

Si dice che un'istanza r_0 di R ***soddisfa*** la DF $X \to Y$ ($r_0 \models X \to Y$) se la proprietà vale per r_0: $\forall$ t_1, $t_2 \in r$. se $t_1[X] = t_2[X]$ allora $t_1[Y] = t_2[Y]$ e che un'istanza r_0 di R ***soddisfa*** un insieme F di DF se per ogni $X \to Y \in F$, vale $r_0 \models X \to Y$:

$r_0 \models X \to Y$ *sse* $\forall$ t_1, $t_2 \in r_0$. se $t_1[X] = t_2[X]$ allora $t_1[Y] = t_2[Y]$.

Dipendenze funzionali atomiche

Ogni dipendenza funzionale $X \to A_1, A_2, \ldots, A_n$, si può scomporre nelle dipendenze funzionali $X \to A_1$, $X \to A_2$, $\ldots$, $X \to A_n$. Le dipendenze funzionali del tipo $X \to A$ si chiamano **dipendenze funzionali atomiche**.

Dipendenze funzionali banali

$Y \to A$ è **non banale** se A non è contenuta in Y. Siamo interessati alle dipendenze funzionali non banali.

Proprietà delle dipendenze funzionali

Notazione: $R\langle T, F\rangle$ denota uno *schema* con **attributi** `T` e **dipendenze funzionali** `F`.

Le DF sono una proprietà semantica, cioè dipendono dai fatti rappresentati e non da come gli attributi sono combinati in schemi di relazione.

Si parla di DF **completa** quando X $\rightarrow$ Y e per ogni W $\subset$ X, non vale W $\rightarrow$ Y. Se X è una **superchiave**, allora X determina ogni altro attributo della relazione, cioè X $\rightarrow$ T. Se X è una **chiave**, allora X $\rightarrow$ T è una DF completa. Da un insieme F di DF, in generale altre DF sono "implicate" da F.

- **Dipendenze implicate**: sia F un insieme di DF sullo schema R, diremo che

  $$\text{F } \textbf{implica logicamente } \text{X} \rightarrow \text{Y } (F \models X \rightarrow Y),$$

  se ogni istanza r di R che soddisfa F soddisfa anche X $\rightarrow Y$.

- **Dipendenze banali**:

  $$\text{implicate dal vuoto, es. } \{\} \models X \rightarrow X$$

8.1.2 Regole di inferenza

Come derivare DF implicate logicamente da F? Usando un insieme di regole di inferenza.

"**Assiomi**" (sono in realtà regole di inferenza) di **Armstrong**:

Se Y $\subseteq$ X, allora X $\rightarrow$ Y	**Riflessività**
Se X $\rightarrow$ Y, Z $\subseteq$ T, allora XZ $\rightarrow$ YZ	**Arricchimento**
Se X $\rightarrow$ Y, Y $\rightarrow$ Z, allora X $\rightarrow$ Z	**Transitività**

Derivazione

Definizione: sia F un insieme di DF, diremo che X $\rightarrow$ Y sia ***derivabile*** da F ($F \vdash X \rightarrow Y$), se X $\rightarrow$ Y può essere inferito da F usando gli assiomi di Armstrong.

Formalmente

Una **derivazione** di f da F è una sequenza finita $f_1, \ldots, f_m$ di dipendenze, dove f_m = f e ogni f_i è un elemento di F oppure è ottenuta dalla precedenti dipendenze $f_1, \ldots, f_{i-1}$ della derivazione usando una regola di inferenza. Si dimostra che valgono anche le regole

$$\begin{array}{lll} \{X \rightarrow Y, X \rightarrow Z\} & \vdash & X \rightarrow YZ \text{ (\textbf{unione U})} \\ Z \subseteq Y,\ \{X \rightarrow Y\} & \vdash & X \rightarrow Z \text{ (\textbf{decomposizione D})} \end{array}$$

Da **Unione** e **Decomposizione** si ricava che se Y = $A_1A_2 \ldots A_n$ allora

$$X \rightarrow Y \Leftrightarrow \{X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n\}.$$

Correttezza e completezza degli assiomi di Armstrong

Teorema Gli assiomi di Armstrong sono corretti e completi. Attraverso gli assiomi di Armstrong, si può mostrare l'equivalenza della nozione di **implicazione logica** ($\models$) e di quella di derivazione ($\vdash$): *se una dipendenza è derivabile con gli assiomi di Armstrong allora è anche implicata logicamente* (**correttezza** degli assiomi) e viceversa *se una dipendenza è implicata logicamente allora è anche derivabile dagli assiomi* (**completezza** degli assiomi).

- **Correttezza** degli assiomi:

$$\forall f, F \vdash f \Rightarrow F \models f$$

- **Completezza** degli assiomi:

$$\forall f, F \models f \Rightarrow F \vdash f$$

Chiusura di un insieme F

Definizione 8.1.1. Dato un insieme F di DF, la **chiusura di F**, denotata con $\mathbf{F}^+$, è:

$$F^+ = \{X \to Y \mid F \vdash X \to Y\}$$

Un problema che si presenta spesso è quello di decidere se una dipendenza funzionale appartiene a F^+ (problema dell'implicazione): controllare se una DF $V \to W \in F^+$. La sua risoluzione con l'algoritmo banale (di generare F^+ applicando ad F ripetutamente gli assiomi di Armstrong) ha una complessità esponenziale rispetto al numero di attributi dello schema.

Definizione 8.1.2. Dato $\mathrm{R}\langle \mathrm{T}, \mathrm{F} \rangle$, e $\mathrm{X} \subseteq \mathrm{T}$, la ***chiusura* di X rispetto ad F**, denotata con $\mathrm{X_F}^+$ (o X^+, se F è chiaro nel contesto) è

$$X_F^+ = \{A_i \in T \mid F \vdash X \to A_i\}$$

Un algoritmo efficiente per risolvere il problema dell'implicazione senza calcolare la chiusura di F scaturisce dal seguente teorema.

Teorema 8.1.1. $F \vdash X \to Y \Leftrightarrow Y \subseteq {X_F}^+$.

Algoritmo per calcolare $\mathbf{X_F}^+$

Chiusura Lenta: sia X un insieme di attributi e F un insieme di dipendenze. Vogliamo calcolare $\mathrm{X_F}^+$.

1. Inizializziamo X^+ con l'insieme X.
2. Se fra le dipendenze di F c'è una dipendenza $\mathrm{Y} \to \mathrm{A}$ con $\mathrm{Y} \subseteq \mathrm{X}^+$ allora si inserisce A in X^+, ossia $X^+ = X^+ \cup \{A\}$.
3. Si ripete il passo 2 fino a quando non ci sono altri attributi da aggiungere a X^+.
4. Si dà in output $\mathrm{X_F}^+ = \mathrm{X}^+$.

Terminazione dell'algoritmo

L'algoritmo termina perché ad ogni passo viene aggiunto un nuovo attributo a X^+. Essendo gli attributi in numero finito, a un certo punto l'algoritmo deve fermarsi. Per dimostrare la correttezza, si dimostra che $\mathrm{X_F}^+ = \mathrm{X}^+$ (per induzione).

8.1.3 Chiavi e attributi primi

Definizione: dato lo schema $\mathrm{R}\langle \mathrm{T}, \mathrm{F} \rangle$, diremo che un insieme di attributi W $\subseteq$ T è una **chiave candidata** di R, se

$$\begin{array}{ll} W \to T \in F^+ & \text{(W superchiave)} \\ \forall\, V \subset W, V \to T \notin F^+ & \text{(se V} \subset \text{W, V non superchiave)} \end{array}$$

Attributo primo: attributo che appartiene ad almeno una chiave.

- Il problema di trovare tutte le chiavi di una relazione richiede un algoritmo di complessità esponenziale nel caso peggiore.
- Il problema di controllare se un attributo è primo è NP-completo.

8.2 Copertura canonica

Definizione Due insiemi di DF, F e G, sullo schema R sono **equivalenti**,

$$\mathrm{F} \equiv \mathrm{G}, \text{ sse } \mathrm{F}^+ = \mathrm{G}^+.$$

Se $\mathrm{F} \equiv \mathrm{G}$, allora F è una **copertura** di G (e G una copertura di F).

Definizione 8.2.1. Sia F un insieme di DF:

1. Data una $X \to Y \in F$, si dice che X contiene un **attributo *estraneo*** A_i sse

   $$(X - \{A_i\}) \to Y \in F^+, \textit{ ovvero } F \vdash (X - \{A_i\}) \to Y$$

   Per verificare se A è estraneo calcoliamo X^+ e verifichiamo se include B, ovvero se basta X a determinare B.

2. X → Y è una **dipendenza *ridondante*** sse

   $$(F - \{X \to Y\})^+ = F^+$$

   Equivalentemente

   $$F - \{X \to Y\} \vdash X \to Y$$

   Per stabilire se una DF del tipo X → A è ridondante la eliminiamo da F, calcoliamo X^+ e verifichiamo se include A, ovvero se con le DF che restano riusciamo ancora a dimostrare che X determina A.

F è detta una **copertura *canonica*** *sse*

- la parte destra di ogni DF in F è un attributo;
- non esistono attributi estranei;
- nessuna dipendenza in F è ridondante.

Teorema 8.2.1. *Per ogni insieme di dipendenze F esiste una copertura canonica.*

Algoritmo per calcolare una copertura canonica. Trasformare le dipendenze nella forma X → A: si sostituisce l'insieme dato con quello equivalente che ha tutti i secondi membri costituiti da singoli attributi (dipendenze atomiche); eliminare gli attributi estranei: per ogni dipendenza si verifica se esitono attributi eliminabili dal primo membro; eliminare le dipendenze ridondanti.

Riepilogo

Le **ridondanze** sui dati possono essere di due tipi: **ridondanza concettuale**, non ci sono duplicazioni dello stesso dato, ma sono memorizzate *informazioni che possono essere ricavate da altre già contenute nel DB*; **ridondanza logica**, esistono *duplicazioni sui dati che possono generare anomalie* nelle operazioni sui dati.

Le dipendenze funzionali sono definite a **livello di schema** e non a livello di istanza ed hanno sempre **un verso**.

Le dipendenze funzionali sono una **generalizzazione del vincolo di chiave e di *superchiave***. Data una tabella con schema R(X) con superchiave, esiste un vincolo di dipendenza funzionale tra K e qualsiasi attributo dello schema r.

$$K \to X_1, \quad X_1 \subseteq X$$

8.3 Decomposizione di Schemi

In generale, per eliminare anomalie da uno schema occorre decomporlo in schemi più piccoli "equivalenti".

Definizione 8.3.1. Dato uno schema R(T), $\rho = \{R_1(T_1), \ldots, R_k(T_k)\}$ è una **decomposizione** di R *sse* $T_1 \cup \ldots \cup T_k = T$

Due proprietà desiderabili di una decomposizione: **conservazione dei dati** (*notazione semantica*) e **conservazione delle dipendenze**.

Definizione 8.3.2. $\rho = \{R_1(T_1), \ldots, R_k(T_k)\}$ è una decomposizione di uno schema R(T) che **preserva i dati** *sse* per ogni ***istanza valida*** r di R:

$$r = (\pi_{T_1} r) \vee (\pi_{T_2} r) \vee \ldots \vee (\pi_{T_k} r)$$

Dalla definizione di giunzione naturale scaturisce il seguente risultato.

Teorema 8.3.1. *Se $\rho = \{R_1(T_1), \ldots, R_k(T_k)\}$ è una decomposizione di R(T), allora per ogni istanza r di R:*

$$r = (\pi_{T_1} r) \vee (\pi_{T_2} r) \vee \ldots \vee (\pi_{T_k} r)$$

Uno schema R(X) si **decompone senza perdita dei dati** negli schemi $R_1(X_1)$ ed $R_2(X_2)$ se, per ogni possibile istanza di R(X), *il join naturale delle proiezioni di r su X_1 ed X_2 produce la tabella di partenza* (cioè non contiene ***ennuple spurie***).

$$\pi_{X_1}(r) \bowtie \pi_{X_2}(r) = r$$

La decomposizione senza perdita è garantita se l'insieme degli attributi comuni alle due relazioni ($X_1 \cap X_2$) è **chiave per almeno una delle due relazioni decomposte**.

Sia r una relazione su un insieme di attributi X e siano X_1 e X_2 due sottoinsiemi di X la cui unione sia pari a X stesso. Inoltre, sia X_0 l'intersezione di X_1 e X_2, allora r si decompone senza perdita su X_1 e X_2 se soddisfa la dipendenza funzionale $X_0 \to X_1$ oppure la dipendenza funzionale $X_0 \to X_2$.

Teorema 8.3.2 (non formale)**.** *Se l'insieme degli attributi comuni alle due relazioni ($X_1 \cap X_2$) è chiave per almeno una delle due relazioni decomposte allora la decomposizione è senza perdita.*

Dimostrazione (non formale). Supponiamo r sia una relazione sugli attributi ABC e consideriamo la sue proiezioni r_1 su AB e r_2 su AC. Supponiamo che r soddisfi la dipendenza funzionale $A \to C$. Allora **A è chiave per r_1 su AC** e quindi non ci sono in tale proiezione due tuple diverse sugli stessi valori di A. Il join costruisce tuple a partire dalle tuple nelle due proiezioni. Sia $t = (a, b, c)$ una tupla del **join di r_1 e r_2**. Mostriamo che appartiene ad r (cioè non è spuria). t è ottenuta mediante il join da $t_1 = (a, b)$ di r_1 e $t_2 = (a, c)$ su r_2. Allora per definizione di proiezione, esistono due tuple in r, $t'_1 = (a, b, *)$ e $t'_2 = (a, *, c)$ (dove * sta per un valore non noto). Poiché $A \to C$, allora esiste un solo valore in C associato al valore a. Dato che (a, c) compare nella proiezione, questo valore è proprio c. Ma allora nella tupla t'_1 il valore incognito deve essere proprio c, quindi $(a, b, c) \in r$. □

Si noti che quella enunciata è una condizione sufficiente ma non necessaria.

Teorema 8.3.3. *Sia* $\mathrm{R}\langle \mathrm{T}, \mathrm{F} \rangle$ *uno schema di relazione, la decomposizione* $\rho = \{R_1(T_1), R_2(T_2)\}$ ***preserva i dati*** *sse*

$$T_1 \cap T_2 \to T_1 \in F^+ \quad \textit{oppure} \quad T_1 \cap T_2 \to T_2 \in F^+$$

Esistono criteri anche per decomposizioni in più di due schemi.

Anche se una decomposizione è senza perdite, può comunque presentare dei problemi di conservazione delle dipendenze.

Proiezione delle dipendenze

Definizione 8.3.3. Dato lo schema $\mathrm{R}\langle \mathrm{T}, \mathrm{F} \rangle$ e $T_1 \subseteq T$, la **proiezione di F su $\mathbf{T_1}$** è

$$\pi_{T_1}(F) = \{X \to Y \in F^+ \mid XY \subseteq T_1\}$$

Algoritmo banale per il calcolo di $\pi_{T_1}(F)$:

$$\text{for each } \mathrm{Y} \subseteq \mathrm{T}_1 \text{ do } (\mathrm{Z}{:=}\mathrm{Y}^+; \text{ output } \mathrm{Y} \to \mathrm{Z} \cap \mathrm{T}_1)$$

Preservazione delle dipendenze

Definizione 8.3.4. Dato lo schema $\mathrm{R}\langle \mathrm{T}, \mathrm{F} \rangle$, la decomposizione

$$\rho = \{R_1, \ldots, R_n\}$$

preserva le dipendenze se e solo se l'unione delle dipendenze in $\pi_{T_i}(F)$ è una ***copertura*** di F.

Proposizione: dato lo schema $\mathrm{R}\langle \mathrm{T}, \mathrm{F} \rangle$, il problema di stabilire se la decomposizione $\rho = \{R_1, \ldots, R_n\}$ preserva le dipendenze ha complessità polinomiale.

Teorema 8.3.4. *Sia* $\rho = \{\mathrm{R}\langle \mathrm{T_i}, \mathrm{F_i} \rangle\}$ *una decomposizione di* $\mathrm{R}\langle \mathrm{T}, \mathrm{F} \rangle$ *che* ***preservi le dipendenze*** *e tale che un* T_j *per qualche* j ***sia una superchiave per R****. Allora* ρ *preserva i dati.*

8.4 Forme normali

Una **forma normale** è una proprietà di una base di dati relazionale che ne garantisce la "qualità", cioè l'assenza di determinati difetti. Quando una relazione **non è normalizzata** presenta ridondanze e si presta a comportamenti poco desiderabili durante gli aggiornamenti. Esistono diversi tipi di forma normale

- 1FN: impone una restrizione sul tipo di relazione, ogni attributo ha un tipo elementare.
- 2FN, 3FN e FNBC: impongono restrizioni sulle dipendenze. FNBC (Boyce-Codd) è la più naturale e la più restrittiva.

8.4.1 Forma normale di Boyce e Codd

Una relazione r è in **forma normale di Boyce e Codd** se, per ogni dipendenza funzionale (non banale) $\mathrm{X} \to \mathrm{Y}$ definita su di essa, **X contiene una chiave K di *r*** (**è una superchiave**). La forma normale richiede che i concetti in una relazione siano omogenei (solo proprietà *direttamente associate alla chiave*).

Intuizione: se esiste in R una dipendenza $X \to A$ non banale ed X **non è chiave**, allora X modella l'identità di un'entità **diversa** da quelle modellate dall'intera R.

Teorema 8.4.1. $\mathrm{R}\langle \mathrm{T}, \mathrm{F} \rangle$ *è in BCNF* $\Leftrightarrow$ *per ogni* $X \to A \in F^+$*, con* $A \notin X$ *(non banale),* X *è una superchiave.*

Algoritmo di analisi

$\mathrm{R}\langle \mathrm{T}, \mathrm{F} \rangle$ è decomposta in $\mathrm{R_1}$(X,Y) e $\mathrm{R_2}$(X,Y) e su di esse si ripete il procedimento (esponenziale).

Input: $\mathrm{R}\langle \mathrm{T}, \mathrm{F} \rangle$, con F copertura canonica.
Output: **decomposizione in BCNF che preserva i dati**.

$\rho = \{\mathrm{R}\langle \mathrm{T}, \mathrm{F}\rangle\}$
while esiste in ρ una $\mathrm{R_i}\langle \mathrm{T_i}, \mathrm{F_i}\rangle$ non in BCNF per la DF $X \to A$ do
  $T_a = X^+$
  $F_a = \pi_{T_a}(F_i)$
  $T_b = T_i - X^+ + X$
  $F_b = \pi_{T_b}(F_i)$
  $\rho = \rho - R_i + \{R_a\langle T_a, F_a\rangle,\ R_b\langle T_b, F_b\rangle\}$
end

Per ogni dipendenza funzionale che non rispetta la forma normale di Boyce-Codd si prende l'insieme di attributi T_a ottenuto dalla chiusura di X e si calcolano le sue dipendenze funzionali tramite la proiezione di F_i su T_a; dopodiché si calcola l'insieme T_b ottenuto dall'insieme degli attributi T_i al quale si toglie la chiusura di X ma si aggiunge X e si calcola la sua proiezione delle dipendenze funzionali. Quindi ad ogni ciclo si va a rimuovere dall'insieme le relazioni che non rispettano la forma normale e si aggiunge R_a e R_b (i due insiemi appena calcolati).

Preserva i dati, ma non necessariamente le dipendenze. Tuttavia, **in alcuni casi** la BCNF "non è raggiungibile": è necessario ricorrere a una forma normale indebolita.

8.4.2 Terza forma normale

Una relazione r è in **terza forma normale** (**3NF**) se, per ogni DF (non banale) $X \to Y$ definita su r, è verificata almeno una delle seguenti condizioni: X contiene una chiave K di r (come nella FNBC), oppure ogni attributo in Y è contenuto in *almeno* una chiave K di r.

La 3FN è **meno restrittiva** della FNBC: tollera alcune ridondanze ed anomalie sui dati e certifica meno la qualità dello schema ottenuto; tuttavia, la 3FN è **sempre ottenibile**, qualsiasi sia lo schema di partenza.

Teorema 8.4.2. $\mathrm{R}\langle \mathrm{T}, \mathrm{F}\rangle$ *è in 3FN se per ogni* $X \to A \in F^+$, *con* $A \notin X$, ***X è una superchiave*** *oppure* ***A è primo***.

Nota:

- la 3FN ammette una dipendenza non banale e non-da-chiave se gli attributi a destra sono primi;
- la NFBC non ammette mai nessuna dipendenza non banale e non-da-chiave.

Algoritmo di Sintesi: versione base

Input: Un insieme R di attributi e un insieme F di dipendenze su R.
Output: Una decomposizione $\rho = \{S_i\}_{i=1...n}$ di R tale che preservi dati e dipendenze e ogni S_i sia in 3NF, rispetto alle proiezioni di F su S_i.

1. Trova una copertura canonica G di F e poni $\rho = \{\}$.
2. Sostituisci in G ogni insieme $X \to A_1, \ldots, X \to A_h$ di dipendenze con lo stesso determinante, con la dipendenza $X \to A_1 \cdots A_h$.
3. Per ogni dipendenza $X \to Y$ in G, metti uno schema con attributi XY in ρ.
4. Elimina ogni schema di ρ contenuto in un altro schema di ρ.
5. Se la decomposizione non contiene alcuno schema i cui attributi costituiscano una superchiave per R, aggiungi ad essa lo schema con attributi W, con W una chiave di R.

Capitolo 9

DBMS

Un DBMS è un sistema software che gestisce grandi quantità di dati persistenti e condivisi. La gestione di **grandi quantità di dati** richiede particolare attenzione ai problemi di *efficienza* (ottimizzazione delle richieste, ma non solo). La **persistenza** e la **condivisione** richiedono che un DBMS fornisca dei meccanismi per garantire l'*affidabilità* dei dati (fault tolerance), per il *controllo degli accessi* e per il *controllo della concorrenza*.

Diverse altre funzionalità vengono messe a disposizione per motivi di **efficacia**, ovvero per semplificare la descrizione dei dati, lo sviluppo delle applicazioni, l'amministrazione di un DB, ecc.

Condivisione dei dati

La **gestione integrata** e la **condivisione dei dati** permettono di evitare ripetizioni (ridondanza dovuta a copie multiple dello stesso dato), e quindi un inutile spreco di risorse (memoria). Inoltre, la **ridondanza** può dar luogo a problemi di **inconsistenza** delle copie e, in ogni caso, comporta la necessità di *propagare* le modifiche, con un ulteriore spreco di risorse (CPU e rete).

Il modello dei dati

Dal punto di vista utente un DB è visto come una collezione di dati che modellano una certa porzione della realtà di interesse. L'**astrazione logica** con cui i dati vengono resi disponibili all'utente definisce un **modello dei dati**, più precisamente:

un modello dei dati è una collezione di concetti che vengono utilizzati per descrivere i dati, le loro associazioni/relazioni e i vincoli che questi devono rispettare.

Un ruolo di primaria importanza nella definizione di un modello dei dati è svolto dai **meccanismi che possono essere usati per strutturare i dati**. Ad esempio esistono modelli in cui i dati sono descritti (solo) sotto forma di alberi (modello *gerarchico*), di grafi (modello *reticolare*), di oggetti complessi (modello *a oggetti*), di relazioni (modello *relazionale*).

Indipendenza fisica e logica

Tra gli obiettivi di un DBMS vi sono quelli di fornire caratteristiche di:

- **Indipendenza fisica**: l'*organizzazione fisica* dei dati dipende da considerazioni legate all'efficienza delle organizzazioni adottate. La riorganizzazione fisica dei dati *non deve comportare effetti collaterali sui programmi applicativi*.
- **Indipendenza logica**: permette di accedere ai dati logici indipendentemente dalla loro rappresentazione fisica.

9.1 Gerarchia delle memorie

La memoria di un sistema di calcolo è organizzata in una gerarchia: al livello più alto memorie di piccola dimensione, molto veloci, costose; scendendo lungo la gerarchia la dimensione aumenta, diminuiscono la velocità e il costo.

Dato un indirizzo di memoria, le prestazioni si misurano in termini di tempo di accesso, determinato dalla somma della latenza (tempo necessario per accedere al primo byte) e del tempo di trasferimento (tempo necessario per muovere i dati).

$$\text{Tempo di accesso} = \text{latenza} + \frac{\text{dim. dati da trasferire}}{\text{veloc. di trasferimento}}$$

Implicazioni per i DBMS

Un DB, a causa della sua dimensione, risiede normalmente su dischi (ed eventualmente anche su altri tipi di dispositivi): i dati devono essere trasferiti in memoria centrale per essere elaborati dal DBMS. *Il trasferimento non avviene in termini di singole tuple, bensì di blocchi* o *pagine*[1]; poiché spesso le operazioni di I/O costituiscono il collo di bottiglia del sistema, si rende necessario ottimizzare l'implementazione fisica del DB, attraverso un'organizzazione efficiente delle tuple su disco, strutture di accesso efficienti, gestione efficiente dei buffer in memoria e delle strategie di esecuzione efficienti per le query.

[1] **Pagine**: termine comunemente usato quando i dati sono in memoria.

9.1.1 Gli Hard Disk

Un hard disk (HD) è un dispositivo elettro-meccanico per la conservazione di informazioni sotto forma magnetica, su supporto rotante a forma di piatto su cui agiscono delle testine di lettura/scrittura.

Il meccanismo del disk drive include organi di registrazione, di posizionamento e di rotazione: un'unità a dischi contiene una pila di dischi metallici che ruota a velocità costante ed alcune testine di letture che si muovono radialmente al disco. Una **traccia** è organizzata in settori di dimensione fissa; i settori sono raggruppati logicamente in **blocchi**, che sono l'unità di trasferimento. Trasferire un blocco richiede un tempo di posizionamento delle testine, un tempo di latenza rotazionale e tempo per il trasferimento (trascurabile).

Pagine

Un blocco (o **pagina**) è una **sequenza contigua di settori su una traccia e costituisce l'unità di I/O per il trasferimento di dati da/per la memoria principale**. La dimensione tipica di una pagina è di qualche KB (4-64 KB): pagine piccole comportano un maggior numero di operazioni di I/O, mentre pagine grandi tendono ad aumentare la frammentazione interna (pagine parzialmente riempite) e richiedono più spazio in memoria per essere caricate. Il *tempo di trasferimento di una pagina* ($\mathbf{T_t}$) da disco a memoria centrale dipende dalla dimensione della pagina ($\mathbf{P}$) e dal transfer rate ($\mathbf{T_r}$).

9.1.2 Gestore della memoria permanente e gestore del buffer

Gestore memoria permanente: fornisce un'astrazione della memoria permanente in termini di insiemi di file logici di pagine fisiche di registrazioni (blocchi), nascondendo le caratteristiche dei dischi e del sistema operativo.
Gestore del buffer: si preoccupa del trasferimento delle pagine tra la memoria temporanea e la memoria permanente, offrendo agli altri livelli una visione della memoria permanente come un insieme di pagine utilizzabili in memoria temporanea, astraendo da quando esse vengano trasferite dalla memoria permanente al buffer e viceversa. Usa una tabella associativa per mantenere la relazione $\langle$`idPage`, `posBuffer`$\rangle$. Inoltre, ad ogni pagina è associato un ***dirty* bit**, indica se la pagina è stata modificata o meno e di conseguenza se deve essere aggiornata sul disco, e un **pin count**, un contatore che viene incrementato/decrementato ogni volta che una pagina viene richiesta/rilasciata.

- `getAndPinPage`: richiede la pagina al buffer manager e vi pone un pin ("spillo"), ad indicarne l'uso.
- `unPinPage`: rilascia la pagina e elimina un pin.
- `setDirty`: indica che la pagina è stata modificata, ovvero è dirty ("sporca").
- `flushPage`: forza la scrittura della pagina su disco, rendendola così "pulita".

Politiche di rimpiazzamento

Nei sistemi operativi una comune politica adottata per decidere quale pagina rimpiazzare è la LRU (*Least Recently Used*), ovvero si rimpiazza la pagina che non è in uso da più tempo. Nei DBMS, LRU non è sempre una buona scelta, in quanto per alcune query il "pattern di accesso" ai dati è noto e può quindi essere utilizzato per operare scelte più accurate in grado di migliorare anche di molto le prestazioni.

L'hit ratio, ovvero la frazione di richieste che non provocano una operazione di I/O, indica sinteticamente quanto buona è una politica di rimpiazzamento; ad esempio esistono algoritmi di `join` che scandiscono N volte le tuple di una relazione: in questo caso la politica migliore sarebbe la MRU (*Most Recently Used*), ovvero rimpiazzare la pagina usata più di recente!

... altro motivo per cui i DBMS non usano (tutti) i servizi offerti dai sistemi operativi...

Struttura di una pagina

Struttura fisica: un insieme, di dimensione fissa, di caratteri.
Struttura logica: informazioni di servizio, area che contiene le stringhe che rappresentano i record.

Una **directory** contiene *un puntatore per ogni record nella pagina*; con questa soluzione l'**identificatore di un record** (**RID**) nel DB è formato da una coppia:

- **PID**: identificatore della pagina.
- **Slot**: posizione all'interno della pagina.

È possibile sia individuare velocemente un record, sia permettere la sua riallocazione nella pagina senza modificare il RID.

9.2 Gestore delle strutture di memorizzazione

9.2.1 Organizzazione seriale e sequenziale

Organizzazione **seriale** (**heap** file): i dati sono memorizzati in modo disordinato uno dopo l'altro; è semplice e a basso costo di memoria, tuttavia è poco efficiente e va bene solo per pochi dati. È l'organizzazione standard di ogni DBMS.

Organizzazione **sequenziale**: i dati sono *ordinati* sul valore di uno o più attributi ottenendo ricerche più veloci; tuttavia le nuove inserzioni fanno perdere l'ordinamento.

La ricerca binaria su file di dati ordinato richiede $\log_2 b$ *accessi a blocco/pagina*. Se il file contiene b_i blocchi, la localizzazione di un record richiede la ricerca binaria nel file e l'accesso al blocco che contiene il record; quindi richiede $\log_2 b_i + 1$ *accessi a blocco/pagina*.

9.2.2 Organizzazione per chiave

Obiettivo: noto il valore di una chiave, trovare il record di una tabella con qualche accesso al disco (idealmente un accesso). **Alternative**:

- Metodo procedurale (hash) o tabellare (indice).
- Organizzazione statica o dinamica.

Hash File

In un file hash *i record vengono allocati in una pagina il cui indirizzo dipende dal valore di chiave del record*:

$$\text{key} \rightarrow \text{H(key)} \rightarrow \text{page address}$$

Una comune funzione hash è il *resto della divisione intera*:

$$H(k) = k \ mod \ NP$$

Si può applicare anche a chiavi alfanumeri che dopo averle convertite. L'insieme delle chiavi è molto più grande dell'insieme dei possibili valori dell'indice.

Metodo procedurale statico

Parametri di progetto:

- la funzione per la trasformazione della chiave;
- il **fattore di caricamento**

$$d = \frac{N}{M \cdot c}$$

  Frazione dello spazio fisico disponibile mediamente utilizzata. Se $\boldsymbol{N}$ **è il numero di tuple** previsto per il file, $\boldsymbol{M}$ **il fattore di pagine** e $\boldsymbol{c}$ **il fattore di caricamento**, il file può prevedere un **numero di blocchi** $\boldsymbol{B}$ pari al numero intero immediatamente superiore a $\boldsymbol{d}$;

- la capacità c delle pagine;
- il metodo per la gestione dei trabocchi.

Le collisioni (overflow) sono di solito gestite con *linked list*, è l'organizzazione più efficiente per l'accesso diretto basato su valori della chiave con condizioni di uguaglianza (**accesso puntuale**); non è efficiente per *ricerche basate su intervalli* (n. per ricerche basate su altri attributi). Funzionano solo con file la cui dimensione non varia molto nel tempo (procedurale **statico**).

Metodo tabellare

Il metodo procedurale (hash) è utile per ricerche per chiave ma non per intervallo. Per entrambi i tipi di ricerche è utile invece il **metodo tabellare**: si usa un *indice*[2], ovvero un *insieme ordinato* di coppie $\langle k, r(k) \rangle$, dove k è un valore della chiave ed $r(k)$ è un riferimento al record con chiave k.

L'indice è gestito di solito con un'opportuna struttura albero detta B^+-*tree*, la struttura più usata e ottimizzata dai DBMS. Gli indici possono essere multi-attributi.

9.2.3 B$^+$-tree

Albero binario di ricerca

Albero binario etichettato in cui per ogni nodo, il sottoalbero sinistro contiene solo etichette minori di quella del nodo e il sottoalbero destro etichette maggiori. Tempo di ricerca (e inserimento), pari alla profondità: logaritmico nel caso "medio" (assumendo un ordine di inserimento casuale).

[2]**Indice**: struttura che contiene *informazioni sulla posizione di memorizzazione delle tuple* sulla base del valore del campo chiave

Albero di ricerca di ordine P

Ogni nodo ha (fino a) P figli e (fino a) $P-1$ etichette ordinate: un albero di ricerca di ordine p è un albero i cui nodi contengono al più $p-1$ *search value* e P *puntatori* nel seguente ordine

$$\langle P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q \rangle \qquad \text{con } q \leq p$$

Ogni P_i è un puntatore ad un nodo figlio (o un puntatore nullo) e ogni K_i è un *search value* appartenente ad un insieme totalmente ordinato. Ogni albero di ricerca deve soddisfare due vincoli fondamentali:

1. ogni nodo $K_1 < K_2 < \ldots < K_{q-1}$;
2. per tutti i valori di X presenti nel sottoalbero puntato da P_i, vale che

$$K_{i-1} < X < K_i \quad \text{per } 1 < i < q$$

$$X < K_i \quad \text{per } i = 1$$

$$K_{i-1} < X \quad \text{per } i = q$$

Un albero di ricerca può essere utilizzato per cercare record memorizzati su disco. **Ogni ricerca/modifica comporta la visita di un cammino radice foglia**. I valori di ricerca (*search value*) possono essere i valori di uno dei campi del file (*search field*) e ad ogni valore di ricerca è associato un puntatore al record avente quel valore (oppure al blocco contenente quel record) nel file di dati.

L'albero stesso può essere memorizzato su disco, assegnando ad ogni nodo dell'albero una **pagina**: quando un nuovo record deve essere inserito, l'albero di ricerca deve essere aggiornato includendo il valore del campo di ricerca del nuovo record, col relativo puntatore al record (o alla pagina che lo contiene) nell'albero di ricerca.

Per inserire/cancellare valori di ricerca nell'albero di ricerca sono necessari specifici algoritmi che garantiscano il rispetto dei due vincoli fondamentali. In generale, tali algoritmi non garantiscono che un albero di ricerca risulti sempre bilanciato (nodi foglia tutti allo stesso livello) $\rightarrow$ soluzione: B-tree e B^+-tree.

B-tree

Un B-tree di ordine p, se usato come struttura di accesso su un campo chiave per la ricerca di record in un file di dati, deve soddisfare le seguenti condizioni:

- ogni nodo interno del B-tree ha la forma

$$\langle P_1, \langle K_1, Pr_1 \rangle, P_2, \langle K_2, Pr_2 \rangle, \ldots, \langle K_{q-1}, Pr_{q-1} \rangle, P_q \rangle \quad \text{con } q \leq p_1$$

  dove P_i è un ***tree pointer*** (puntatore ad un altro nodo del B-tree), K_i è la ***chiave di ricerca*** e Pr_i è un ***data pointer*** (puntatore ad un record il cui campo chiave di ricerca è uguale a K_i o alla pagina che contiene tale record);

- per ogni nodo, si ha che

$$K_1 < K_2 < \ldots < K_{q-1}$$

- ogni nodo ha al più p *tree pointer*;

- per tutti i valori X della chiave di ricerca appartenenti al sottoalbero puntato da P_i, si ha che

$$K_{i-1} < X < K_i \quad \text{per } 1 < i < q$$

$$X < K_i \quad \text{per } i = 1$$

$$K_{i-1} < X \quad \text{per } i = q$$

- ogni **nodo**, esclusa la radice, ha almeno $\lceil \frac{p}{2} \rceil$ *tree pointer*;

- la **radice** ha almeno due *tree pointer*, a meno che non sia l'unico nodo dell'albero;

- un nodo con q *tree pointer*, $q \leq p$, ha $q - 1$ campi chiave di ricerca (e $q - 1$ data pointer);

- **tutti i nodi foglia sono posti allo stesso livello** (i nodi foglia hanno la stessa struttura dei nodi interni, ad eccezione del fatto che tutti i loro tree pointer P_i sono nulli).

B$^+$-tree

Un **B$^+$-tree** è un B-tree in cui *i data pointer sono memorizzati solo nei nodi foglia dell'albero*. La struttura dei nodi foglia differisce da quella dei B-tree e quindi da quella dei nodi interni.

Se il campo di ricerca è un campo chiave, i nodi foglia hanno per ogni valore del campo di ricerca una entry e un puntatore ad un record.

Se un campo di ricerca non è un campo chiave, i puntatori indirizzano un blocco che contiene i puntatori ai record del file di dati, rendendo così necessario un passo aggiuntivo per l'accesso ai dati.

I nodi foglia di un B^+-tree sono generalmente messi fra loro in relazione; ciò viene sfruttato nel caso di rangequery: essi corrispondono al primo livello di un indice, mentre i nodi interni corrispondono agli altri livelli di un indice.

Alcuni valori del campo di ricerca presenti nei nodi foglia sono ripetuti nei nodi interni per guidare la ricerca.

La struttura dei **nodi interni** (di ordine p) di un B^+-tree è la seguente:

1. ogni nodo interno ha la forma

   $$\langle P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q \rangle \quad \text{con } q \leq p$$

   dove ogni P_i è un *tree pointer*(puntatore ad un altro nodo)

2. per ogni nodo interno, si ha che

   $$K_1 < K_2 < \ldots < K_{q-1}$$

3. ogni nodo interno ha al più p tree pointer;

4. per tutti i valori X della *search key* nel sottoalbero puntato da P_i, si ha che

   $$X \leq K_i \quad \text{per } i = 1$$
   $$K_{i-1} < X \leq K_i \quad \text{per } 1 < i < q$$
   $$K_{i-1} < X \quad \text{per } i = q$$

5. ogni nodo interno ha almeno $\lceil \frac{p}{2} \rceil$ tree pointer, mentre la radice ha almeno 2 tree pointer se è un nodo interno;

6. un nodo interno con q tree pointer, con $q \leq p$, ha $q-1$ campi di ricerca.

La struttura dei nodi **foglia** (di ordine p_{leaf}) di un B^+-tree è la seguente:

1. ogni nodo foglia è della forma

   $$\langle \langle K_1, Pr_1 \rangle, \langle K_2, Pr_2 \rangle, \ldots, \langle K_q, Pr_q \rangle P_{next} \rangle$$

   dove $q \leq p_{leaf}$ e per ogni nodo si ha che $K_1 < K_2 < \ldots < K_q$.

- P_{next} è un *tree pointer che punta al successivo nodo foglia* del B^+-tree;
- ogni Pr_i è un *data pointer che punta al record con valore del campo di ricerca uguale a K_i o ad un blocco contenente tale record* (o ad un blocco di puntatori ai record con valore del campo di ricerca uguale a K_i, se il campo di ricerca non è una chiave);

2. ogni nodo foglia ha almeno $\lceil \frac{p_{leaf}}{2} \rceil$ valori;
3. tutti i nodi foglia sono dello stesso livello.

9.2.4 Indici e B^+-tree

Di solito un **indice è organizzato a B^+-tree** per permettere di trovare con pochi accessi, a partire da un valore v, i record di R in cui il valore di A è in una relazione specificata con v. Esistono due tipologie di indici ad albero

- **statici**: *la struttura ad albero viene creata sulla base dei dati presenti nel DB* e non più modificata (o modificata periodicamente);
- **dinamici**: *la struttura ad albero viene aggiornata ad ogni operazione* sulla base di dati di inserimento o di cancellazione, memorizzati, preservando le prestazioni senza bisogno di riorganizzazioni.

Inoltre si possono distinguere indici **primari**, in questo caso la chiave di ordinamento del file sequenziale coincide con la *chiave di ricerca dell'indice*, e **secondari**, in questo caso invece la *chiave di ordinamento* e la chiave di ricerca sono diverse.

Un indice può essere *definito su di un insieme $A_1, \ldots, A_n$ di attributi*. In questo caso l'indice contiene un record per ogni combinazione di valori assunti dagli attributi $A_1, \ldots, A_n$ nella relazione e può essere utilizzato per rispondere in modo efficiente ad interrogazioni che specifichino un valore per ciascuno di questi attributi.

Indice primario

Un indice **primario** è un file ordinato i cui record sono di lunghezza fissa e sono costituiti da due campi: il primo campo è dello stesso tipo del campo *chiave di ordinamento* (chiave primaria), mentre il secondo campo è un *puntatore a un blocco del disco*.

Esiste un record nel file dell'indice per ogni blocco nel file di dati.

Indice secondario

Un indice **secondario** può essere definito su un *campo non chiave* che è una chiave *candidata* e ha valori univoci, o su un campo non chiave con *valori duplicati*. Il primo campo è dello stesso tipo del campo che non viene usato per ordinare il file ed è chiamato *campo di indicizzazione*, il secondo campo è un *puntatore al blocco* oppure un *puntatore al record* (RID).

9.3 Ordinamento di archivi

L'algoritmo più comunemente utilizzato dai DBMS è quello detto di **Merge Sort** a Z vie (Z-way Sort-Merge). Supponiamo di dover ordinare un input che consiste di un file di *NP pagine* e di avere a disposizione solo **NB** $<$ *NP* **buffer** in memoria centrale. L'algoritmo opera in 2 fasi:

- *Sort interno*: si leggono una alla volta le pagine del file; i record di ogni pagina vengono ordinati facendo uso di un algoritmo di sort interno (es. quick-sort); ogni pagina così ordinata, detta anche "*run*", viene scritta su disco in un file temporaneo.
- *Merge*: operando uno o più passi di fusione, le *run* vengono fuse, fino a produrre un'unica *run*.

Caso base

Per semplicità si consideri il caso base a $Z = 2$ vie e si supponga di avere a disposizione solo $NB = 3$ buffer in memoria centrale.

Nel caso base $Z = 2$ si fondono 2 run alla volta; con $NB = 3$, si associa un buffer a ognuna delle run, il terzo buffer serve per produrre l'output, una pagina alla volta: si legge la prima pagina da ciascuna run e si può quindi determinare la prima pagina dell'output; quando tutti i record di una pagina di run sono stati consumati, si legge un'altra pagina della run.

Si consideri per semplicità solo il **numero di operazioni di I/O**; si può osservare che:

- nella fase di sort interno si leggono e si riscrivono *NP* pagine,
- ad ogni passo di merge si leggono e si riscrivono *NP* pagine ($2 \cdot NP$); il numero di passi fusione è pari a $\lceil \log_2 NP \rceil$, in quanto ad ogni passo il numero di run si dimezza.

Il costo complessivo è pertanto pari a $2 \cdot NP \cdot (\lceil \log_2 NP \rceil + 1)$.

In realtà se NP non è una potenza di 2 il numero effettivo di I/O è leggermente minore di quello calcolato, in quanto in uno o più passi di fusione può capitare che una run non venga fusa con un'altra.

Caso generale

Una prima osservazione è che nel passo di sort interno, avendo a disposizione NB buffer, si possono ordinare NP pagine alla volta (anziché una sola), il che porta a un costo di

$$2 \cdot NP \cdot \left(\left\lceil \log_2 \left(\frac{NP}{NB} \right) \right\rceil + 1 \right)$$

Miglioramenti sostanziali si possono ottenere se, avendo $NB > 3$ buffer a disposizione, si fondono $NB - 1$ run alla volta (un buffer è per l'output). In questo caso il numero di passi di fusione è logaritmico in $NB - 1$, ovvero è pari a

$$2 \cdot NP \cdot \left(\left\lceil \log_{NB-1} \left(\frac{NP}{NB} \right) \right\rceil + 1 \right)$$

9.3.1 Realizzazione degli operatori relazionali

Oltre che per ordinare le tuple, il sort può essere utilizzato per le query in cui compare l'opzione `DISTINCT`, ovvero per eliminare i duplicati, e per le query che contengono la clausola `GROUP BY`. Caso di `DISTINCT`:

1. si legge R e si scrive T che contiene solo gli attributi della `SELECT`,
2. si ordina T su tutti gli attributi,
3. si eliminano i duplicati.

Caso di `GROUP BY`: si ordinano i dati sugli attributi del `GROUP BY`, poi si visitano i dati e si calcolano le funzioni di aggregazione per ogni gruppo.

Un'altra situazione in cui il sort può essere usato è per il prodotto cartesiano. $R \times S$ è grande; pertanto, $R \times S$ seguito da una restrizione è inefficiente. Anche se il `Join` è logicamente commutativo, dal punto di vista fisico vi è una chiara distinzione, che influenza anche le prestazioni, tra operando sinistro (o "esterno", "outer") e operando destro (o "interno", "inner").

Nested Loops

```
foreach record r ∈ R do
    foreach record s ∈ S do
        if r_i = s_j then
            aggiungi ⟨r, s⟩ al risultato
```

Il costo di esecuzione dipende dallo spazio a disposizione in memoria centrale. Nel caso base in cui vi sia un buffer per R e un buffer per S, bisogna leggere una volta R e $Nrec(R)$ volte S, ovvero tante volte quante sono le tuple della relazione esterna, per un totale di

$$Npag(R) + Nrec(R) \cdot Npag(S) \text{ I/O}$$

Se è possibile allocare $Npag(S)$ buffer per la relazione interna il costo si riduce a $Npag(R) + Npag(S)$.

Costo con R esterno:

$$Npag(R) + Nrec(R) \cdot Npag(S) \approx Npag(R) \cdot \frac{Nrec(R)}{Npag(R)} \cdot Npag(S)$$

Costo con S esterno:

$$Npag(S) + Nrec(S) \cdot Npag(R) \approx Npag(S) \cdot \frac{Nrec(S)}{Npag(S)} \cdot Npag(R)$$

Si sceglierà R come esterna e S come interna se

$$\frac{Nrec(R)}{Npag(R)} < \frac{Nrec(S)}{Npag(S)}$$

che corrisponde a dire che le tuple di R sono più grandi di quelle di S.

Come esterna conviene la relazione con record più lunghi/grandi.

L'ordine con cui vengono generate le tuple del risultato coincide con l'ordine eventualmente presente nella relazione esterna. Pertanto se l'ordine che si genera è "interessante", ad esempio perché la query contiene `ORDER BY R.A`, la scelta della relazione esterna può risultarne influenzata.

Nested loop a pagine

Molti DBMS usano una variante del Nested Loops, detta Nested loop a pagine (*PageNestedLoop*), che, rinunciando a preservare l'ordine della relazione esterna, risulta più efficiente in quanto esegue il `join` *di tutte le tuple in memoria prima di richiedere nuove pagine della relazione interna.*

```
foreach pagina p_r di R do
    foreach pagina p_s di S do
        esegui il join di tutte le tuple in p_r e p_s
```

Il costo è ora pari a $NP(R) + NP(R) \cdot NP(S)$.

La strategia si estende anche al caso in cui a R siano assegnati più buffer.

Nested loop con indice (IndexNestedLoop)

Data una tupla della relazione esterna R, la scansione completa della relazione interna S può essere sostituita da una scansione basata su un indice costruito sugli attributi di `join` di S, secondo il seguente schema

```
foreach record r ∈ R do
    usa l'indice su j per trovare tutti i record s ∈ S | s_j = r_j
    aggiungi ⟨r, s⟩ al risultato
```

L'accesso alla relazione interna mediante indice porta in generale a ridurre di molto i costi di esecuzione del Nested Loops Join.

Sort Merge

Il Sort-merge Join è applicabile quando entrambi *gli insiemi di tuple in input sono* ***ordinati*** *sugli attributi di join.* Per R(S) ciò è possibile se:

- R(S) è fisicamente ordinata sugli attributi di join,
- esiste un indice sugli attributi di join di R(S).

$r = first(R); s = first(R)$
while $r \in R \wedge s \in S$ do
  if $r_i = s_j$
    avanza r ed s fino a che r_i e s_j non cambiano entrambe, aggiungendo ciascun $\langle r, s \rangle$ al risultato
  else if $r_i < s_j$ avanza r dentro R
  else if $r_i > s_j$ avanza s dentro S

La logica dell'algoritmo (senza considerare il tempo per il sort) sfrutta il fatto che entrambi gli input sono ordinati per evitare di fare inutili confronti, ciò fa sì che il numero di letture sia dell'ordine di $Npag(R) + Npag(S)$ se si accede sequenzialmente alle due relazioni.

9.4 Piani di accesso

L'ottimizzazione delle interrogazione è fondamentale nei DBMS. È necessario conoscere il funzionamento dell'ottimizzatore per una buona progettazione fisica. Obiettivo dell'ottimizzatore: scegliere il piano con costo minimo, fra possibili piani alternativi, usando le statistiche presenti nel catalogo.

Un **piano di accesso** è un albero che descrive l'intero algoritmo che sarà usato per implementare l'interrogazione desiderata. Le foglie sono le tabelle ed i nodi interni specificano le modalità (operatori fisici) con cui gli accessi alle tabelle e le operazioni relazionali sono effettuate.

Ideale: trovare il piano migliore.
Euristica: evitare i piani peggiori!

9.4.1 Operatori fisici

Gli algoritmi per realizzare gli operatori relazionali si codificano in opportuni operatori fisici, ad esempio `TableScan(R)` è l'operatore fisico per la scansione di R. Ogni operatore fisico è un **iteratore**, un *oggetto* con metodi *open*, *next*, *isDone*, *reset* e *close* realizzati usando gli operatori della macchina fisica, con *next* che ritorna un record.

Interfaccia a iteratore

DBMS definiscono gli operatori mediante un'interfaccia a "iteratore", i cui metodi principali sono:

- **open**: *inizializza* lo stato dell'operatore, *alloca il buffer* per gli input e l'output, *richiama ricorsivamente open sugli operatori figli*; viene anche usato per ***passare argomenti*** (ad es. la condizione che un operatore `Filter` deve applicare).
- **next**: usato per richiedere un'altra ennupla del risultato dell'operatore; l'implementazione di questo metodo *include next sugli operatori figli* e codice specifico dell'operatore.
- **close**: usato per *terminare l'esecuzione* dell'operatore, con conseguente *rilascio delle risorse* ad esso allocate.
- **isDone**: indica se vi sono ancora valori da leggere, in genere è booleano.

Un piano di accesso è un algoritmo per eseguire un'interrogazione usando gli operatori fisici disponibili.

Operatori logici e fisici

Operatore logico	Operatore fisico
R	`TableScan(R)`: per la scansione di R.
	`IndexScan(R, Idx)`: per la scansione di R con l'indice Idx.
	`SortScan(R, {A_i})`: per la scansione di R ordinata sugli $\{A_i\}$.
$\pi_{\{A_i\}}$	`Project(O, {A_i})`: per la proiezione dei record di O senza l'eliminazione dei duplicati.
	`Distinct(O)`: per eliminare i duplicati dei record ordinati di O.
σ_{ψ}	`Filter(O, ψ)`: per la restrizione senza indici dei record di O.
	`IndexFilter(R, Idx, ψ)`: per la restrizione con indice dei record di R.
$\tau_{\{A_i\}}$	`Sort(O, {A_i})`: per ordinare i record di O sugli $\{A_i\}$, per valori crescenti.

Operatore logico	Operatore fisico
$\{A_i\}\gamma\{f_i\}$	`GroupBy`(O, {A_i}, {f_i}): per raggruppare i record di O sugli {A_i} usando le funzioni di aggregazione in {f_i}. - Nell'insieme {f_i} vi sono le funzioni di aggregazione presenti nella `SELECT` e nella `HAVING`. - L'operatore restituisce record con attributi gli {A_i} e le funzioni in {f_i}. - I record di O sono ordinati sugli {A_i}.
$\bowtie_{\psi_J}$	`NestedLoop`(O_E, O_I, ψ_J): per la giunzione con il *nested loop* e ψ_J la condizione di giunzione.
	`PageNestedLoop`(O_E, O_I, ψ_J): per la giunzione con il *page nested loop*.
	`IndexNestedLoop`(O_E, O_I, ψ_J): per la giunzione con l'*index nested loop*. L'operando interno O_I è un `IndexFilter`(R, Idx, ψ_J) oppure `Filter`(O, ψ'), con O un `IndexFilter`(R, Idx, ψ_J). Per ogni record r di O_E, la condizione ψ_J dell'`IndexFilter` è quella di giunzione con gli attributi di O_E sostituiti dai valori in r.
	`SortMerge`(O_E, O_I, ψ_J): per la giunzione con il *sort-merge*, con i record di O_E e O_I ordinati sugli attributi di giunzione.

9.5 Transazioni

Cos'è una transazione?

Definizione 9.5.1. Una **transazione** è un'*unità logica di elaborazione che corrisponde a una serie di operazioni fisiche elementari* (letture/scritture) sul DB.

Le **transazioni** rappresentano l'unità di lavoro elementare (insiemi di istruzioni SQL) che *modificano* il contenuto di una base di dati. Sintatticamente una transizione è contornata dai comandi `begin transaction` (e `end transaction`); all'interno possono comparire i comandi di `commit work` e `rollback work`.

Proprietà **ACID** delle transizioni:

- **A**tomicità → la transazione deve essere eseguita con la regola del "tutto o niente".
- **C**onsistenza → la transazione deve lasciare il DB in uno stato consistente, eventuali vincoli di integrità non devono essere violati.
- **I**solamento → l'esecuzione di una transazione deve essere indipendente dalle altre.
- Persistenza (**D**urability) → l'effetto di una transazione che ha fatto `commit work` non deve essere perso.

Una *funzionalità* essenziale di un DBMS è la *protezione dei dati* da malfunzionamenti e da interferenze dovute all'accesso contemporaneo ai dati di più utenti.

Per il programmatore una transazione è un programma sequenziale costituito da operazioni che il sistema deve eseguire garantendo atomicità, consistenza, serializzabilità e persistenza (ACID).
Gestore dell'affidabilità:

- **Atomicità**: le transazioni che terminano in modo prematuro (*aborted transactions*) sono trattate dal sistema come se non fossero mai iniziate; pertanto eventuali loro effetti sulla base di dati sono annullati.
- **Persistenza**: le modifiche sulla base di dati di una transazione terminata normalmente sono permanenti, cioè non sono alterabili da eventuali malfunzionamenti.

Gestore della concorrenza:

- **Serializzabilità**: nel caso di esecuzioni concorrenti di più transazioni, l'effetto complessivo è quello di una esecuzione seriale.

Per aumentare l'efficienza prestazionale, tutti i DBMS utilizzano un buffer temporaneo di informazioni in memoria principale, il quale viene periodicamente scritto su memoria secondaria, quindi è necessaria una comunicazione con il gestore del buffer.

Una transazione può eseguire molte operazioni sui dati recuperati da una base di dati, ma al DBMS interessano solo quelle di *lettura* o *scrittura* della base di dati, indicate con $r_i[x]$ e $w_i[x]$. Un dato letto o scritto può essere un *record*, un *campo* di un record o una pagina. Per semplicità supporremo che sia una pagina. Un'operazione di lettura $r_i[x]$ comporta la lettura di una pagina nel buffer, se non già presente. Un'operazione di scrittura $w_i[x]$ comporta l'eventuale lettura nel buffer di una pagina e la sua modifica nel

buffer, ma non necessariamente la sua scrittura in memoria permanente. Per questa ragione, in caso di malfunzionamento, si potrebbe perdere l'effetto dell'operazione.

Tipi di fallimento

Fallimenti di transazioni: non comportano la perdita di dati in memoria temporanea né persistente (es.: violazione di vincoli, violazione di protezione, stallo).
Fallimenti di sistema: comportano la perdita di dati in memoria temporanea non di dati in memoria persistente (es.: comportamento anomalo del sistema, caduta di corrente, guasti hardware sulla memoria centrale).
Disastri: comportano la perdita di dati in memoria persistente (es.: danneggiamento di periferica).

9.5.1 Gestione delle transazioni

Il gestore dell'affidabilità verifica che siano garantite le proprietà di **atomicità** e **persistenza** delle transazioni; quindi è responsabile dell'implementazione dei comandi di `begin transaction`, `commit` e `rollback`, di ripristinare il sistema dopo *malfunzionamenti software* (**ripresa a caldo**), di ripristinare il sistema dopo *malfunzionamenti hardware* (**ripresa a freddo**).

Il controllore di affidabilità utilizza un log, nel quale sono indicate tutte le **operazioni svolte dal DBMS**.

Convenzioni notazionali: data una transazione T, indicheremo con B(T), C(T) e A(T) i record di `begin`, `commit` e `abort` relativi a T e con U(T, O, BS, AS), I(T, O, AS) e D(T, O, BS) i record di update, insert e delete, rispettivamente su un *oggetto O*, dove BS è *before state* e AS è *after state*.

I record del log associati ad una transazione, consentono di disfare e rifare le corrispondenti azioni sulla base di dati:

- primitiva di **undo**: per *disfare* un'azione su un oggetto O, è sufficiente ricopiare in O il valore BS (l'insert viene disfatto cancellando l'oggetto O);
- primitiva di **redo**: per *rifare* un'azione su un oggetto O, è sufficiente ricopiare in O il valore AS (il delete viene rifatto cancellando l'oggetto O).

Il log si presenta come un file sequenziale suddiviso in record. Esistono due tipi di record:

- **di transazione**: tengono traccia delle operazioni svolte da ciascuna transizione sul DBMS. Per ogni transazione, un record di begin (B), record di insert (I), delete (D) e update (U) e un record di commit (C) o di abort (A).
- **di sistema**: tengono traccia delle operazioni di sistema (dump/checkpoint).

L'operazione di dump

L'operazione di **dump** produce una copia completa della base di dati, effettuata in mutua esclusione con tutte le altre transazioni quando il sistema non è operativo. La copia viene memorizzata in memoria stabile (backup).

Al termine del dump, viene scritto nel log un record di dump, che segnala l'avvenuta esecuzione dell'operazione in un dato istante. Il sistema riprende, quindi, il suo funzionamento normale.

Protezione dei dati da malfuzionamenti

- Copia della BD (dump).
- Giornale/**log** durante l'uso della BD, il sistema registra nel giornale/log la storia delle azioni effettuate sulla BD dal momento in cui ne è stata fatta l'ultima copia.

All'interno del giornale/log si ha $\langle \texttt{T}, \texttt{begin} \rangle$ e per ogni operazione di modifica viene memorizzata la transazione responsabile, il tipo di ogni operazione eseguita, la nuova e vecchia versione del dato modificato:

$$\langle \texttt{T}, \texttt{write}, \texttt{address}, \texttt{oldV}, \texttt{newV} \rangle$$

Segue $\langle \texttt{T}, \texttt{commit} \rangle$ o $\langle \texttt{T}, \texttt{abort} \rangle$.

Regole di scrittura del log

- Regola Write Ahead Log (WAL) $\rightarrow$ la parte ***BS*** (*before state*) di ogni record di log deve essere scritta *prima che la corrispondente operazione* venga effettuata nella base di dati.
- Regola di Commit Precedence $\rightarrow$ la parte ***AS*** (*after state*) di ogni record di log deve essere scritta nel log *prima di effettuare* il `commit` della transazione.

Gli algoritmi si differenziano a seconda del modo in cui si trattano le scritture sulla BD e la terminazione delle transazioni:

- Disfare - Rifare
- Disfare - Non Rifare
- Non Disfare - Rifare
- Non Disfare - Non Rifare

Ipotesi: le scritture nel giornale vengono portate subito nella memoria permanente.

Disfare

Quando si portano le modifiche nella BD? Politica della **modifica *libera***: le modifiche *possono* essere portate nella BD stabile prima che la transazione termini (disfare o *steal*).

Regola per poter disfare: prescrittura nel giornale ("**Log Ahead Rule**" o "**Write Ahead Log**"); *se la nuova versione di una pagina rimpiazza la vecchia sulla BD stabile prima che la transazione abbia raggiunto il punto di commit, allora la vecchia versione della pagina deve essere portata prima sul giornale in modo permanente.*

Rifare

Come si gestisce la terminazione? `commit` ***libero***: una transazione può essere considerata terminata normalmente prima che tutte le modifiche vengano riportate nella BD stabile (occorre rifare).

Regola per poter rifare una transazione: "**Commit Rule**"; *le modifiche (nuove versioni delle pagine) di una T devono essere portate stabilmente nel giornale prima che la transazione raggiunga il commit (condizione per rifare).*

Punto di allineamento

Al momento del ripristino, solo gli aggiornamenti più recenti tra quelli riportati sul giornale/log potrebbero non essere stati ancora riportati sulla base di dati.

Come ottenere la certezza che non sia necessario rieseguire le operazioni più vecchie? Periodicamente si fa un *checkpoint* (`CKP`): si scrive la marca `CKP` sul giornale/log per indicare che tutte le operazioni che la precedono sono state effettivamente effettuate sulla BD.

Un modo (troppo semplice) per fare il `CKP`:

1. si sospende l'attivazione di nuove transazioni,
2. si completano le precedenti, si allinea la base di dati (ovvero si riportano su disco tutte le pagine "sporche" dei buffer),
3. si scrive nel giornale/log la marca `CKP`,
4. si riprende l'esecuzione delle operazioni.

Si scrive sul giornale una marca di inizio checkpoint che riporta l'elenco delle transazioni attive: `BeginCkp` $\{\mathtt{T}_1, \ldots, \mathtt{T}_n\}$. In parallelo alle normali operazioni delle transazioni, il gestore del buffer riporta sul disco tutte le pagine modificate. Si scrive sul giornale una marca di `EndCkp`, la quale certifica che tutte le scritture avvenute prima del `BeginCkp` ora sono sul disco.

Le scritture avvenute tra `BeginCkp` e `EndCkp` forse sono sul disco e forse no.

In caso di fallimento di una **transazione** si scrive nel giornale $\langle \mathtt{T}, \mathtt{abort} \rangle$ e si applica la procedura disfare.

Se si verifica un fallimento di **sistema** la BD viene ripristinata con il comando `Restart` (ripartenza di emergenza), a partire dallo stato al punto di allineamento, procedendo come segue: *le transazioni non terminate devono essere disfatte*, mentre *le transazioni terminate devono essere rifatte*.

Infine, nel caso di **disastro**, si riporta in linea la copia più recente della BD e la si aggiorna rifacendo le modifiche delle transazioni terminate normalmente (ripartenza a freddo).

Ripresa a caldo

Garantisce atomicità e persistenza delle transazioni. Quattro fasi:

1. trovare l'ultimo checkpoint (ripercorrendo il log a ritroso);
2. costruire gli insiemi UNDO (transazioni da disfare) e REDO (transazioni da rifare);
3. ripercorrere il log all'indietro, *fino alla* ***più*** *vecchia azione* delle transazioni in UNDO, disfacendo tutte le azioni delle transazioni in UNDO;
4. ripercorrere il log in avanti, rifacendo tutte le azioni delle transazioni in REDO.

Ripresa a freddo

Risponde ad un guasto che provoca il deterioramento della BD:

1. si ripristinano i dati a partire dal backup,
2. si eseguono le operazioni registrate sul giornale fino all'istante del guasto,
3. si esegue una ripresa a caldo.

9.6 Gestore della concorrenza

Uno schedule $\boldsymbol{S}$ si dice **seriale** se *le azioni di ciascuna transazione appaiono in sequenza*, senza essere inframezzate da azioni di altre transazioni.

$$S = \{T_1, T_2, \ldots, T_n\}$$

Schedule seriale ottenibile se:

- Le transazioni sono *eseguite uno alla volta* (scenario non realistico).
- Le transazioni sono *completamente indipendenti* l'una dall'altra (improbabile).

In un sistema reale, le transazioni vengono eseguite in concorrenza per ragioni di efficienza/scalabilità. Tuttavia, l'esecuzione concorrente determina un insieme di **problematiche** che devono essere gestite; il DBMS deve garantire che l'esecuzione concorrente di transazioni avvenga senza interferenze in caso di accessi agli stessi dati.

Il DBMS transazionale gestisce questi problemi garantendo la proprietà di **isolamento**: garantisce che essa sia eseguita come se non ci fosse concorrenza. Questa proprietà è assicurata facendo in modo che ciascun insieme di transazioni concorrenti sottoposte al sistema sia "**serializzabile**".

Definizione 9.6.1. Un'esecuzione di un insieme di transazioni $\{T_1, \ldots, T_n\}$ si dice **seriale** se, per ogni coppia di transazioni T_i e T_j, tutte le operazioni di T_i vengono eseguite prima di qualsiasi operazione T_j o viceversa.

Definizione 9.6.2. Un'esecuzione di un insieme di transazioni si dice **serializzabile** se produce lo stesso effetto sulla base di dati di quello ottenibile eseguendo serialmente, in un qualche ordine, le sole transazioni terminate normalmente.

Nella pratica i DBMS implementano tecniche di controllo di concorrenza che garantiscono direttamente la serializzabilità delle transazioni concorrenti. Tali tecniche si dividono in due classi principali:

- Protocolli **pessimistici**/conservativi: tendono a "*ritardare*" l'esecuzione di transazioni che potrebbero generare conflitti, e quindi anomalie, rispetto alla transazione corrente. Cercano quindi di prevenire.
- Protocolli **ottimistici**: permettono l'esecuzione sovrapposta e non sincronizzata di transazioni ed effettuano un controllo sui possibili conflitti generati *solo a valle del* `commit`.

Controllo della concorrenza ottimistico

Ogni transazione effettua "liberamente" le proprie operazioni sugli oggetti della base di dati secondo l'ordine temporale con cui le operazioni stesse sono generate. Al `commit` viene effettuato un controllo per stabilire se sono stati riscontrarti eventuali conflitti e nel caso viene effettuato il rollback delle azioni delle transazioni e la relativa riesecuzione. In generale, un protocollo di controllo di concorrenza ottimistico è basato su 3 fasi:

- Fase di **lettura**: ogni transazione legge i valori degli oggetti della BD su cui deve operare e li memorizza in variabili (copie) locali dove sono effettuati eventuali aggiornamenti.
- Fase di **validazione**: vengono effettuati dei controlli sulla serializzabilità nel caso che gli aggiornamenti locali delle transazioni dovessero essere propagati sulla base di dati.
- Fase di **scrittura**: gli aggiornamenti delle transazioni che hanno superato la fase di validazione sono propagati definitivamente sugli oggetti della BD.

Controllo della concorrenza pessimistico

Nella pratica i DBMS implementano tecniche di controllo di concorrenza che garantiscono direttamente la serializzabilità delle transazioni concorrenti. Tali tecniche si dividono in due classi principali: metodi basati su *lock* e metodi basati su *timestamp*.

I DMBS commerciali usano il meccanismo dei lock: blocca l'accesso ai dati ai quali una transazione accede ad altre transazioni.

- Lock a *livello di riga*, *tabella*, *pagina* (multi **granularità**).

- Lock in operazioni di scrittura (*mutua esclusione*) e di lettura (*accesso condiviso*) (**multimodale**).

Su ogni lock possono essere definite due operazioni: **richiesta** del lock in lettura/scrittura e **rilascio** del lock (unlock) acquisito in precedenza.

In generale, quando una risorsa è bloccata, le transazioni che ne richiedono l'accesso vengono in genere messe in coda; quindi devono aspettare (che il lock sia rimosso). In sostanza, questo è un meccanismo efficace, ma influisce sulle prestazioni.

Serializzatore SPL Stretto

Il gestore della concorrenza (serializzatore) dei DBMS ha il compito di stabilire l'ordine secondo il quale vanno eseguite le singole operazioni per rendere serializzabile l'esecuzione di un insieme di transazioni.

Definizione 9.6.3. Il protocollo del blocco a due fasi stretto (**Strict Two Phase Locking**) è definito dalle seguenti regole:

- ogni transazione, prima di effettuare un'operazione acquisisce il blocco corrispondente (chiede il lock);
- transazioni diverse non ottengono blocchi in conflitto;
- i blocchi/lock si rilasciano alla terminazione della transazione (cioè al `commit`/`abort`).

Problema: i protocolli 2PL possono generare schedule con situazioni di **deadlock** (**stallo**).

Per gestire le situazioni di *deadlock* causate dal gestore della concorrenza, si possono usare tre tecniche:

1. Uso dei **timeout**: *ogni operazione di una transazione ha un timeout* entro il quale deve essere completata, *pena annullamento* (`abort`) *della transazione stessa.*
2. **Deadlock avoidance**: prevenire le configurazioni che potrebbero portare ad un deadlock. Come? Eseguendo il **lock/unlock** di tutte le risorse allo stesso tempo, oppure utilizzando dei **time-stamp** o delle **classi di priorità** tra transazioni (può determinare *starvation*[3]).

[3] *Starvation*: quando una transazione è impossibilitata a proseguire la sua esecuzione per un periodo di tempo indefinito, mentre le altre transazioni del sistema proseguono tranquillamente.

3. **Deadlock detection**: utilizzare *algoritmi per identificare eventuali situazioni di deadlock* e prevedere meccanismi di recovery dal deadlock. → Grafo delle richieste/risorse utilizzato per identificare la presenza di cicli (corrispondenti a deadlock); in caso di ciclo, si fa `abort` delle transazioni coinvolte nel ciclo in modo da eliminare la mutua dipendenza.

Time-stamp delle transazioni

Un metodo alternativo al 2PL per la gestione della concorrenza in un DBMS prevede l'utilizzo dei *time-stamp delle transazioni* (metodo **TS**): ad ogni transazione si associa un **time-stamp** che rappresenta il momento di inizio della transazione; ogni transazione *non può leggere o scrivere un dato scritto da una transazione con time-stamp maggiore* e *non può scrivere su un dato già letto da una transazione con time-stamp maggiore.*

Livelli di isolamento/consistenza per ogni transazione

`SERIALIZABLE` assicura che la transazione T legga solo cambiamenti fatti da transazioni concluse (che hanno fatto il `commit`), che nessun valore letto o scritto da T verrà cambiato da altre transazione finché T non è conclusa e che se T legge un insieme di valori acceduti secondo qualche condizione di ricerca, l'insieme non viene modificato da altre transazione finché T non è conclusa.

`REPEATABLE READ` assicura che la transazione T legga solo cambiamenti fatti da transazioni concluse (che hanno fatto il `commit`) e che nessun valore letto o scritto da T verrà cambiato da altre transazione finché T non è conclusa.

`READ COMMITTED` assicura che la transazione T legga solo cambiamenti fatti da transazioni concluse (che hanno fatto il `commit`) e che T non veda nessun cambiamento eventualmente effettuato da transazioni concorrenti non concluse tra i valori letti all'inizio di T.

`READ UNCOMMITTED`: a questo livello di isolamento una transazione T può leggere modifiche fatte ad un oggetto da un transazione in esecuzione; ovviamente l'oggetto può essere cambiato mentre T è in esecuzione. Quindi T è soggetta a effetti fantasma.