Appunti delle lezioni della Vaglini - Basi di dati

Gabriele Frassi

A.A 2019-2020 - Secondo semestre

Indice

Mi auguro che quanto troverete in questo PDF sia utile. Non vi fidate troppo per quanto riguarda gli esercizi e la serializzabilità.

Parte I

Unimap

- **Mer 04/03/2020 08:30-11:30 (3:0 h)** lezione: Introduzione. Modello relazionale. (GIGLIOLA VAGLINI)
- **Gio 05/03/2020 10:30-12:30 (2:0 h)** non tenuta: Sospensione della didattica (GIGLIOLA VAGLINI)
- **Ven 06/03/2020 14:30-16:30 (2:0 h)** non tenuta: Sospensione della didattica (FRANCESCO PISTOLESI)
- **Mer 11/03/2020 08:30-11:30 (3:0 h)** lezione: Algebra relazionale. (GIGLIOLA VAGLINI)
- **Gio 12/03/2020 10:30-12:30 (2:0 h)** lezione: Introduzione e modalità d'esame. Il DBMS MySQL. Sintassi di una query: il SELECT statement. Processing. Condizioni e connettivi logici. Duplicati e keyword DISTINCT. Il valore NULL. Condizioni sui valori NULL. Gestione e formattazione di date. Funzioni DATEDIFF, PERIOD_DIFF e DATE_FORMAT. Lassi di tempo: la keyword INTERVAL. Shift temporale con DATE_ADD/DATE_SUB, e somma diretta. Condizioni temporali legate all'istante di esecuzione: l'impiego di CURRENT_DATE. (FRANCESCO PISTOLESI)
- **Ven 13/03/2020 14:30-16:30 (2:0 h)** lezione: Funzioni di aggregazione: count, count(distinct), sum, avg, min e max. Ridenominazione. Il problema del record connesso a un valore aggregato. Multi-table querying. Inner join. Processing di una query con inner join. Query con join e condizioni sui record. Alias. Natural join. Cross join. Outer join. (FRANCESCO PISTOLESI)
- **Mer 18/03/2020 08:30-11:30 (3:0 h)** lezione: Calcolo relazionale (GIGLIOLA VAGLINI)
- **Gio 19/03/2020 10:30-12:30 (2:0 h)** esercitazione: Risoluzione ragionata degli esercizi per casa. Formulazione delle condizioni, connettivi logici, condizioni temporali e gestione delle date. Commento al codice. (FRANCESCO PISTOLESI)
- **Ven 20/03/2020 11:30-13:30 (2:0 h)** esercitazione: Progetto di interrogazioni con espressioni algebriche o nel calcolo relazionale (GIGLIOLA VAGLINI)
- **Ven 20/03/2020 14:30-16:30 (2:0 h)** lezione: Self join. Uso degli alias. Join multipli. Derived table. Subquery noncorrelated e correlated. Direttiva IN. Subquery scalari. Processazione in MySQL. Risoluzione mista subquery-join. Common Table Expressions (CTE). (FRANCESCO PISTOLESI)
- **Mer 25/03/2020 08:30-11:30 (3:0 h)** lezione: Progetto: modello concettuale (GIGLIOLA VAGLINI)
- **Gio 26/03/2020 10:30-12:30 (2:0 h)** esercitazione: Risoluzione ragionata degli esercizi per casa su join, subquery e funzioni di aggregazione. Esempio di traduzione passo-passo dalla versione con subquery alla versione join-equivalente. (FRANCESCO PISTOLESI)
- **Ven 27/03/2020 14:30-16:30 (2:0 h)** lezione: Raggruppamento. A cosa serve, come funziona e quando usarlo. La clausola GROUP BY. Condizioni sui gruppi: la HAVING clause. Processing in MySQL. Subquery EXISTS. Query con significato insiemistico. Unione di result set. Implementazione della divisione con doppia subquery NOT EXISTS, e con raggruppamento e subquery di conteggio nella having clause. (FRANCESCO PISTOLESI)
- **Mer 01/04/2020 08:30-11:30 (3:0 h)** lezione: La parte DD di SQL. Ristrutturazione di uno schema ER prima della traduzione. (GIGLIOLA VAGLINI)
- **Gio 02/04/2020 10:30-12:30 (2:0 h)** esercitazione: Risoluzione ragionata degli esercizi su query con raggruppamento con join e subquery. Uso delle CTE. (FRANCESCO PISTOLESI)
- **Ven 03/04/2020 11:30-13:30 (2:0 h)** lezione: Traduzione nel modello logico (GIGLIOLA VAGLINI)

- **Ven 03/04/2020 14:30-16:30 (2:0 h)** lezione: Query complesse. Ruolo nell'analisi predittiva, nei modelli decisionali, nel CRM. Strategie risolutive. Modificatori ANY/ALL. Gestire gli ex aequo. Differenza insiemistica. Introduzione alle stored procedure. Sintassi di CREATE PROCEDURE. Gestione degli statement. La keyword DELIMITER. Visualizzazione di result set su standard output. Chiamata a stored procedure. (FRANCESCO PISTOLESI)
- **Mer 08/04/2020 08:30-11:30 (3:0 h)** esercitazione: Esercizi di progetto, di traduzione e di ridondanza. (GIGLIOLA VAGLINI)
- **Gio 16/04/2020 10:30-12:30 (2:0 h)** esercitazione: Esercitazione sulle query complesse. (FRANCESCO PISTOLESI)
- **Ven 17/04/2020 14:30-16:30 (2:0 h)** lezione: Variabili locali e variabili user-defined in una stored procedure. Assegnamento con set e con select into. Parametri di una stored procedure: in, out, inout. Istruzioni condizionali: if-elseif-else, case. Istruzioni iterative: while, repeat e loop. Istruzioni di salto: leave e iterate. Cursori. Sintassi del comando declare cursor. Handler. Tipologie exit e continue. Ciclo di fetch. Exception handling e comando signal. Errori sqlstate. Not found condition. Stored function. Sintassi del comando create function. Esempio di stored function. Esempio di stored procedure che usa una stored function per realizzare un ranking tramite temporary table. (FRANCESCO PISTOLESI)
- **Mer 22/04/2020 08:30-11:30 (3:0 h)** lezione: Dipendenze funzionali e forme normali delle relazioni. (GIGLIOLA VAGLINI)
- **Gio 23/04/2020 10:30-12:30 (2:0 h)** esercitazione: Esercizi su stored procedure e stored function. Svolgimento di due testi d'esame. (FRANCESCO PISTOLESI)
- **Ven 24/04/2020 14:30-16:30 (2:0 h)** lezione: Introduzione ai database attivi. Trigger. Sintassi dell'istruzione create trigger. Trigger after e before. Meccanismo di scatto. Keyword 'new'. Gestione in sync di un attributo ridondante mediante trigger after e before. Business rule e gestione mediante trigger before. Event. Generalità sull'aggiornamento deferred. Pregi e difetti. Sintassi del comando create event. Scheduling di un event. Recurring event. Significato della direttiva on schedule at/every. Introduzione alle materialized view. Utilità nel reporting e nel data analytics. Performance. Politiche di refresh: immediate, deferred e on demand. (FRANCESCO PISTOLESI)
- **Mer 29/04/2020 08:30-11:30 (3:0 h)** lezione: Normalizzazione delle relazioni. (GIGLIOLA VAGLINI)
- **Gio 30/04/2020 10:30-12:30 (2:0 h)** esercitazione: Esercitazione su forme normali e normalizzazione delle relazioni. (GIGLIOLA VAGLINI)
- **Mer 06/05/2020 08:30-11:30 (3:0 h)** lezione: Esecuzione delle transazioni: recovery manager del DBMS (GIGLIOLA VAGLINI)
- **Gio 07/05/2020 10:30-12:30 (2:0 h)** esercitazione: Esercitazione sulle procedure di restart delle transazioni a seguito di guasti soft e hard. (GIGLIOLA VAGLINI)
- **Ven 08/05/2020 14:30-16:30 (2:0 h)** lezione: Esempio di materialized view con implementazione passo-passo del sync e del full refresh in modalità on demand e deferred. Incremental refresh. Log table. Trigger di push. Progettazione di log table efficienti: overhead vs. occupazione di memoria. Implementazione dell'incremental refresh. Processing parziale e totale della log table. Trasferimento delle modifiche nella materialized view. Le modalità partial e complete. (FRANCESCO PISTOLESI)
- **Mer 13/05/2020 08:30-11:30 (3:0 h)** lezione: Controllo della concorrenza nell'esecuzione delle transazioni: il concetto di scheduler (GIGLIOLA VAGLINI)
- **Gio 14/05/2020 10:30-12:30 (2:0 h)** esercitazione: Esercizi sulla serializzabilità degli schedule. (GIGLIOLA VAGLINI)

- **Ven 15/05/2020 11:30-13:30 (2:0 h)** esercitazione: Two-phase locking e time-stamp, differenze. (GIGLIOLA VAGLINI)
- **Ven 15/05/2020 14:30-16:30 (2:0 h)** esercitazione: Esercitazione su materialized view. Implementazione del full refresh mediante stored procedure ed event. Impostazione dell'esercizio per l'implementazione dell'incremental refresh e analisi preliminare della struttura della log table. (FRANCESCO PISTOLESI)
- **Mer 20/05/2020 08:30-11:30 (3:0 h)** lezione: Lo schema fisico delle basi di dati: indici, metodi di esecuzione degli operatori algebrici, in particolare i metodi per il join. Accesso, scansione e ordinamento. Calcolo delle dimensioni dei risultati intermedi. (GIGLIOLA VAGLINI)
- **Gio 21/05/2020 10:30-12:30 (2:0 h)** lezione: Calcolo delle dimensioni dei risultati intermedi: dettagli e esempi per i vari operatori. (GIGLIOLA VAGLINI)
- **Ven 22/05/2020 14:30-16:30 (2:0 h)** lezione: Introduzione alle window functions (analytic functions). Aggregate vs. non-aggregate functions. Clausola over. Definizione della partition e clausola partition by. Non-aggregate functions. Sort della partition. Uso combinato di partition by e order by. Le funzioni row_number, rank, dense_rank. Rank multipli. Funzioni lead e lag. Analisi delle frequenze relative cumulate tramite cume_dist. Window functions su frame. Funzioni first_value e last_value. Definizione di frame mediante rows e range. Funzione moving average. Introduzione alle pivot table. Flat data. Operazione di pivoting. Pivoting statico. SQL dinamico. Prepared statement. Comandi prepare ed execute. Pivoting in SQL dinamico. (FRANCESCO PISTOLESI)
- **Mer 27/05/2020 08:30-11:30 (3:0 h)** esercitazione: Esercizi da esame da provare: algebra, calcolo, dipendenze funzionali e normalizzazione. (GIGLIOLA VAGLINI)
- **Gio 28/05/2020 10:30-12:30 (2:0 h)** esercitazione: Prove di esercizi d'esame: affidabilità, concorrenza e schema fisico. (GIGLIOLA VAGLINI)
- **Ven 29/05/2020 11:30-13:30 (2:0 h)** esercitazione: Simulazione di test di accesso all'orale. (FRANCESCO PISTOLESI,GIGLIOLA VAGLINI)
- **Ven 29/05/2020 14:30-16:30 (2:0 h)** esercitazione: Esercitazione su incremental refresh di materialized view. Tecniche di aggiornamento di attributi aggregati in modalità incremental. Esempio di gestione della media e dei valori massimi tramite concatenazione e parsing. Risoluzione ragionata e commento al codice. (FRANCESCO PISTOLESI)
- **Mer 03/06/2020 14:00-16:00 (2:0 h)** esercitazione: [Recupero del 06/03/2020] Esercizi e tutoring su window functions, tabelle pivot ed SQL dinamico. (FRANCESCO PISTOLESI)

Parte II

Appunti delle lezioni

1 — Mercoledì 04/03/2020

Sistema organizzativo Ogni azienda basa la propra esistenza su un sistema organizzativo, cioè un insieme di risorse e regole atte a perseguire in modo coordinato le esigenze della stessa. Con risorse intendiamo persone, denaro, materiale, ma soprattuto informazioni!

Sistemi informativi La componente del sistema organizzativo che si occupa di ogni singolo processo (aggiunta, manipolazione, ma anche conservazione...) che ha a che fare con l'informazione è detta *sistema informativo*. Il mantenimento dell'informazione è vitale per ogni azienda ed ente! Il concetto è indipendente da qualsiasi forma di automatizzazione: i sistemi informativi esistono da secoli, ancora prima della nascita dell'informatica (le banche hanno sempre conservato moli elevate di informazioni fin dalla loro nascita).

Sistema informativo automatizzato Oggi i sistemi informativi mantengono le informazioni mediante un approccio informatico (*sistema informatico*): è inevitabile, soprattutto nelle grandi aziende, il ricorso a tecnologie informatiche.

1.1 Base di dati

Il cuore del sistema informativo automatizzato è la *base di dati*, un insieme organizzato di dati da cui possiamo trarre le informazioni di nostro interesse.

1.1.1 Differenza tra dati e informazioni

Informazione Una notizia, un dato o un elemento che permette di avere conoscenza di ciò che ci circonda. L'interpretazione dell'informazione, la *decodifica*, ci porta al dato.

Dato Il dato è un concetto generale, la rappresentazione simbolica di un qualunque tipo di informazione all'interno del sistema informatizzato. La codifica dell'informazione è parte sostanziale del progetto della base di dati: decideremo di rappresentare qualcosa mediante stringhe, qualcos'altro mediante interi... e così via.

- I dati non hanno alcun significato da soli, ma se contestualizzati possono portarci a conoscere delle informazioni. Prendiamo una stringa *Mario* e il numero 9. Individualmente non hanno alcun significato, ma in un certo contesto possono portarci a conoscere delle informazioni: il numero 9 all'anagrafe potrebbe essere l'età della persona, a scuola una valutazione didattica....

- I dati sono un qualcosa di tendenzialmente stabile nel tempo. Gli enti pubblici negli ultimi anni hanno adottato l'approccio informatico nella conservazione dell'informazione, adottando nuove proceure: queste hanno ereditato i dati già in possesso dei vari enti.

- I dati costituiscono un patrimonio significativo da sfruttare e proteggere. Basta pensare al grande potere di chi possiede dati: i governi, che possono tassarti; le agenzie di marketing, che possono sfruttare informazioni personali o la tua cronologia per suggerirti prodotti più interessanti; i social, i cui dati possono essere utilizzati da chiunque per poter conoscere le tendenze del momento...

1.1.2 Caratteristiche di una base di dati

Le basi di dati sono

- **grandi**: una base di dati possono contenere una mole elevatissima di dati. La dimensione di una base può essere superiore a quella della memoria centrale e raggiungere milioni di gigabyte. Un sistema deve essere in grado di gestire *memorie secondarie*. Ovviamente una base di dati può essere anche piccola, ma i sistemi devono poter gestire una base indipendentemente dalla loro dimensione.

- **condivise**: la base di dati è una risorsa integrata e condivisa tra più settori di un'organizzazione, ciascuno con un proprio sottosistema informativo. Applicazioni e utenti diversi devono poter accedere a dati comuni: mediante un meccanismo di *controllo di concorrenza* il sistema gestisce le operazioni svolte in contemporanea da più utenti.
  Una base di dati può essere utilizzata anche per scambiare informazioni: se un programma all'interno di questo "ecosistema" modifica i dati della base gli altri programmi visualizzeranno i dati aggiornati (effettivamente uno scambio).

- **persistenti**: le basi di dati non hanno un tempo di vita collegato all'esecuzione dei programmi che le utilizzano.

1.1.3 Perchè utilizziamo una base di dati?

L'approccio della base di dati ci permette di evitare ridondanze ed errori. Prendiamo un esempio: gli uffici amministrativi, fino a qualche anno fa, conservavano le informazioni in modo cartaceo. Ogni ufficio possedeva un "archivio cartaceo" caratterizzato da centinaia di documenti. Non esisteva, ovviamente, un sistema centralizzato.

- L'ufficio anagrafe conserva i certificati di nascita

- L'ufficio che si occupa delle unioni civili conserva gli atti di matrimonio.

Senza un sistema centralizzato non si hanno controlli di consistenza: l'aggiornamento di uno dei due archivi non comporta automaticamente l'aggiornamento di informazioni simili contenute in altri. Presumiamo sia morto un certo *Mario bianchi*: all'anagrafe risulta morto, ma al secondo ufficio risulta sposato a partire da una data successiva alla morte.

1.2 Il *DBMS*

Adottando l'approccio della base di dati otteniamo un sistema integrato privo di ridondanze ed errori. Ciò risulta vantaggioso anche i termini di memoria: invece di avere più archivi contenenti le stesse informazioni ne abbiamo uno solo.

Università di Pisa La nostra università è divisa in tanti dipartimenti e settori: ciascuno di essi può accedere a un vasto sistema informatico che copre tutto l'ateneo. Esso raccoglie i dati di ogni singolo studente e riduce la possibilità di ridondanze e incoerenze mediante una gestione centralizzata. Pensiamo al portale *Valutami*: i voti dell'ateneo o le prenotazioni non sono gestite da ogni singolo dipartimento ma da un sistema utilizzato da ogni singola facoltà. Altro esempio potrebbero essere le mense: i dati da loro utilizzati (i soldi rimasti sulla carta, l'ISEE,...) sono prelevati dallo stesso sistema informatico!

Chi si occupa dei controlli di consistenza (fino ad ora non possibili)? Il *DBMS*!

Definizione (Base di dati - definizione informatica)**.** Collezione di dati grandi, persistenti e condivisi gestita da un *DBMS*

Definizione (*DBMS*)**.** Questa sigla è acronimo di *Data Base Management System*. Esso è un sistema software in grado di gestire collezione di dati.

1.2.1 Ruolo del *DBMS*

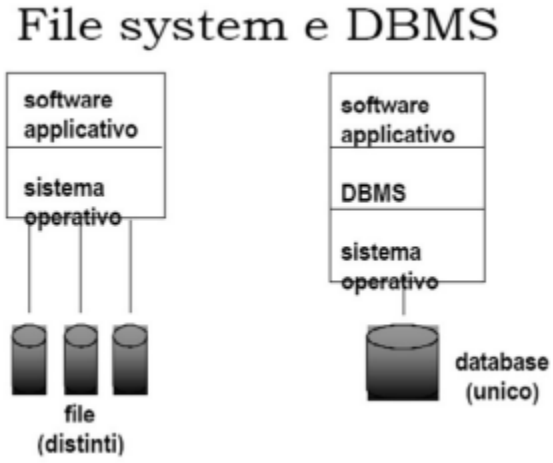
Il *DBMS* "protegge" la base di dati e garantisce

- **affidabilità** (la cosa più importante): i dati, come già detto *persistenti*, devono rimanere inalterati finchè non manipolati da un utente. Il sistema deve prevedere meccanismi di *backup* e *recovery* in caso di malfunzionamenti per non perdere definitivamente i dati salvati.
- **privatezza**: gli utenti devono avere accesso soltanto ai dati di cui hanno bisogno e possono eseguire soltanto le operazioni necessarie. Mediante un sistema di *autorizzazioni* l'amministratore gestisce gli accessi degli utenti stabilendo le operazioni permesse.
- **efficienza**: devono svolgere operazioni in tempi accettabili per l'utente utilizzando al meglio le risorse di memoria a disposizione. I DBMS richiedono molte risorse per il numero elevato di funzionalità disponibili. L'efficienza può essere garantita solo con sistemi informatici adeguati accompagnati da investimenti.
- **efficacia**: devono rendere produttiva l'attività di ogni singolo utente. Il sistema deve essere adeguatamente dimensionato e la base ben progettata.

Ricordiamo la presenza di *controlli di concorrenza* per garantire la possibilità di informazioni simultanee.

1.2.2 Dizionario dei dati, sistema operativo

Parte della base di dati contiene il cosiddetto *dizionario dei dati*: una descrizione unica e centralizzata dei dati utilizzabile dai vari programmi. Il salvataggio dei dati avviene mediante file, ma le funzionalità del file system risultano estese e si hanno più servizi.



1.2.3 Indipendenza dei dati

L'architettura scelta deve garantire l'*indipendenza dei dati*. Questa consiste nella principale proprietà del DBMS. Si permette a utenti e programmi applicativi che utilizzano una base di dati di interagire a un elevato livello di astrazione, che prescinde dai dettagli realizzativi utilizzati nella costruzione della base di dati. L'indipendenza dei dati si articola in

- **indipendenza fisica**: si interagisce col DBMS in modo indipendente dalla struttura fisica dei dati. Posso modificare le strutture fisiche senza influire sulle descrizioni dei dati ad alto livello
- **indipendenza logica**: si interagisce col livello esterno della base in modo indipendente dal livello logico. Posso creare uno schema esterno in base a certe esigenze senza dover modificare lo schema logico. Inoltre, posso modificare il modello logico senza dover modificare le strutture esterne.

Fondamentalmente quanto detto rappresenta la differenza tra interfaccia e implementazione vista a *Fondamenti di programmazione*.

1.2.4 Vantaggi e svantaggi

1.2.4.1 Vantaggi

- Gestione centralizzata (e conseguente riduzione di ridondanze ed errori) con possibilità di "economia di scala"
- Disponibilità di servizi integrati per un'intera organizzazione (maggiore produttività)
- Indipendenza dei dati (favorisce lo sviluppo e la manutenzione delle applicazioni)

1.2.4.2 Svantaggi

- Costo dei prodotti e della transizione verso di essi
- Non scorporabilità (spesso) delle funzionalità (con riduzione di efficienza)

In certe circostanze può risultare utile, soprattutto quando lavoriamo in ambienti piccoli, ricorrere a un approccio più tradizionale nella conservazione dei dati.

1.3 Concetto di *transazione*

Tutti i gestori attualmente in commercio sono detti *transazionali* poichè basati su transazioni (forniscono meccanismi per definire ed eseguire transazioni).
La transazione consiste in un insieme di operazioni (di lettura e scrittura della memoria)

- **indivisibili** (atomicità). Questo significa che se una delle operazioni non può essere eseguita allora la transazione nel suo complesso non può essere eseguita (Esempio: transazione bancaria, compio un prelievo da un conto A e un versamento in un conto B)
- **corrette** anche in presenza di concorrenza (controllo della concorrenza)
- **con effetti definitivi** (permanenza)

Non possiamo vedere lo stato di questa sequenza, ma solo l'inizio e la fine: ciò che avviene nel mezzo non è detto sia corretto, l'importante è che al termine tutto sia effettivamente corretto.
Le transazioni possono essere molto complesse e in alcuni casi provocare errori: lo svolgimento di transazioni deve essere bloccato nel caso in cui possano emergere problemi. Il gestore, ogni volta, stabilisce se l'utente può effettivamente svolgere operazioni.
Quando la transazione termina con successo si ha un risultato di tipo *commit* (impegno) e quanto fatto viene salvato, altrimenti si ha un risultato di tipo *aborted* (abortito) e non vengono apportate modifiche.

1.4 Modello logico

Perchè parliamo di modelli logici? I programmi fanno riferimento ai dati, la cui struttura deve poter essere modificata senza modificare i programmi. Proprio per questo andiamo ad introdurre il *modello logico*, un insieme di costrutti utilizzati per organizzare i dati. Un'organizzazione dei dati, distinta da quella dell'applicazione, permette di dividere l'interfaccia dall'implementazione con un approccio non diverso da quello visto con i moduli a *Fondamenti*: il DBMS nasconde l'implementazione e gli utenti si trovano in un livello astratto da cui possono compiere interrogazioni senza conoscere la struttura.

Esempio Un esempio è il *modello relazionale*, il cui costrutto caratterizzante è la relazione. Essa permette di definire insiemi di record omogeni a struttura fissa: una tabella.

1.4.1 Schema e istanza

Ogni tipo di dato nel modello logico è caratterizzato da

- uno schema, sostanzialmente invariato nel tempo, che descrive la struttura
- un'istanza, i valori, che può variare rapidamente nel tempo.

Nel modello relazionale lo schema consiste nella prima riga della tabella, l'istanza nel corpo della stessa.

1.5 Linguaggi per basi di dati

Il linguaggio utilizzato permette la definizione degli schemi e la manipolazione delle tabelle. Si distinguono due tipi di linguaggio:

- il *Data definition language* (DDL) per la definizione di schemi
- il *Data manipulation language* (DML) per l'interrogazione e l'aggiornamento

Alcuni linguaggi presentano entrambe le funzioni. Il più noto è l'SQL: è detto linguaggio testuale e interrattivo poichè un utente può scrivere l'interrogazione ed eseguirla immediatamente.

1.5.1 Immersione in linguaggio ospite

Caratteristica importante è la possibilità di "immergere" i comandi SQL in un linguaggio ospite: in Java, in C++, in PHP. Questo discorso è affrontato in un capitolo successivo.

1.6 Architettura del DBMS

Con architettura intendiamo il modello di esecuzione utilizzato da un sistema in cui esiste un DBMS. Tutte le realizzazioni di gestori utilizzano un'*architettura distribuita*, caratterizzata dalla presenza di più macchine che lavorano autonomamente e interagiscono mediante la rete.

1.6.1 Architettura a due livelli - *client-server*

L'architettura più diffusa è quella *client-server*, detta anche *a due livelli*. Abbiamo due macchine che comunicano attraverso la rete:

- il server. Ha la base di dati e il DBMS, offre servizi e risponde alle richieste dei client.
- il client. Richiede servizi comunicando con il server attraverso la rete.

In alcune circostanze la macchina potrebbe fungere sia da server che da client, normalmente ciò non avviene. Il server deve avere la potenza sufficiente per svolgere tutte le operazioni richieste dai client in tempi accettabili. Ovviamente possono esserci più server e avere una base di dati distribuita.
Individuiamo due tipi di client:

- **thin client** (il più diffuso): l'elaborazione dei dati (*logica dell'applicazione*) avviene nel server.

- **thick client**: l'elaborazione dei dati avviene nel client. Ciò può risultare svantaggioso: la macchina deve essere potente per trasmettere i dati a tutti i client e ognuna deve avere un rapporto fiduciario col gestore di dati. Inoltre non si possono avere più di poche centinaia di client.

1.6.2 Architettura a tre livelli

Con internet si passa a un'architettura *a tre livelli*. Essa presenta:

- un server contenente la base di dati
- un server applicativo che compie elaborazioni
- un client che compie interrogazioni

Anche in questo caso si possono avere più server, sia del primo tipo che del secondo. Il client, invia le richieste al server applicativo e questo dialoga con il server contenente la base di dati.

Quale vantaggio si cela dietro questa struttura? Il client è di tipo *thin* e si occupa esclusivamente dell'interfacciamento (quindi possibile numero di client maggiore). In un certo senso il client coincide con il browser web!

Sistemi eterogenei Possiamo connettere sistemi non omogenei: posso compiere le solite domande che sono abituato a fare senza rendermi conto, per esempio, di avere un database di diverso tipo.

1.7 Big Data

Per definizione gli oggetti contenuti in base di dati sono di dimensioni enormi. Nel tempo abbiamo visto l'introduzione di strumenti informatici sempre più evoluti e l'aumento di produzione di dati. Descriviamo la cosa mediante le quattro v:

- **Volume**: in un minuto si ottiene già una grande quantità di dati. Milioni di interrogazioni a Google, di like e share su Facebook, di downloads da youtube...
- **Velocità**: i dati si muovono ad alta vellocità. Esistono tecniche che permettono di analizzare i flussi: non salvo i dati ma i risultati dell'analisi dei flussi.
- **Varietà**: i dati non sono più esclusivamente strutturati in tabella, ma prevalentemente non strutturati (testi, immagini, vocali, video)
- **Veridicità**: i dati estratti sono veritieri? Posso estrarre informazioni vere anche in presenza di gravi errori, imprecisioni e incompletezze.

data science La qualità deve essere sempre garantita: ciò ha favorito la nascita della *data science*, una branca dell'informatica che collabora con la matematica (in particolare con la statistica). La data science include i seguenti aspetti:

- *Data cleaning*: costruzione di una raccolta dati con un sufficiente livello di qualità

- *Data integration*: costruzione di una raccolta dati integrata a partire da differenti sorgenti.
- *Data mining*: estrazione di informazioni utili dai dati.
- *Metodi predittivi* Metodi per prevedere mediante osservazioni nel passato dati riguardanti il futuro
- *Machine learning*: metodo predittivo particolare. Vengono forniti alla macchina dei dati iniziali, che ne permettono l'educazione e una progressiva autonomia nell'analisi dei dati.

Data engineers Noi ingegneri ci occuperemo soprattutto delle infrastrutture per analizzare i dati (e non delle tecniche di analisi come il *data scientist*).

1.8 Sistemi non relazionali e potenziamento di un sistema

Negli ultimi anni sono emersi nuovi sistemi non relazionali in cui si rinuncia ad alcuni elementi citati fino ad ora per ottenere maggiore flessibilità. I sistemi NoSQL, per esempio, non hanno una vera e propria struttura dati: semplicemente mi limito ad aggiungere e rimuovere colonne quando necessario. Sia i dati che gli utenti tendono ad aumentare e quindi risulta necessario scalare. Posso adottare uno dei seguenti approcci:

- scalare verticalmente, quindi adottare un approccio centralizzato e aumentare la potenza dei miei server (costi elevati)
- scalare orizzontalmente, approccio distribuito in cui posso aggiungere nuovi server standard e non potentissimi (economico)

L'ultimo approccio è quello adottato con i database *NoSQL*. Ho Mario Bianchi: ogni volta che ho informazioni nuove su di lui continuo ad aggiungerle. Ottengo delle liste. Questi sistemi sono utili in particolare quando abbiamo una crescita molto elevata, quasi incontrollata, di dati. La loro semplicità, tuttavia, è dovuta anche alla mancanza di controlli sull'integrità dei dati: il compito ricade sull'applicazione che dialoga col database. Inoltre l'assenza di una vera e propria struttura dati rende difficile il passaggio a un altro tipo di database. Segue che questi database non domineranno mai il mercato.

1.9 Modello relazionale

Il *modello relazionale* si basa sul concetto matematico di relazione, con alcune differenze. Esso permette di definire insiemi di record omogeni a struttura fissa: una tabella.

Prodotto cartesiano Dati n insiemi $D_1, \ldots, D_n$ (che costituiscono i domini del prodotto cartesiano) non necessariamente distinti il prodotto cartesiano $D_1 \times \cdots \times D_n$ consiste nell'insieme delle *n-uple* ordinate $(v_1, \ldots, v_n)$ dove $v_k \in D_k, k = 1 \ldots n$. Il prodotto cartesiano può essere svolto anche tra insiemi che non presentano la stessa cardinalità.

Relazione matematica Una relazione matematica sugli insiemi $D_1, \ldots, D_n$ consiste in un sottoinsieme del prodotto cartesiano $D_1 \times \cdots \times D_n$ (quindi si considerano solo parte delle *n-uple ordinate* effettivamente ottenute col prodotto cartesiano)

Grado del prodotto cartesiano Il grado del prodotto cartesiano consiste nel numero delle componenti di ogni *n-upla* del prodotto cartesiano.

Cardinalità della relazione La cardinalità della relazione consiste nel numero delle *n-uple* ottenute.
Si ha (nella matematica) una *struttura posizionale* in cui la posizione consiste nell'indice del dominio. Si individua il ruolo di ciascun dominio a partire dalla posizione. L'ordine delle colonne ha un significato ben preciso, mentre l'ordine delle righe non ha importanza (non essendo numerate non si hanno n-uple, ma tuple).

1.9.1 Passaggio a struttura non posizionale

Se associamo ad ogni colonna, e quindi ad ogni dominio, un nome (che chiameremo *attributo*), facciamo venir meno la notazione posizionale introdotta precedentemente: non ho più bisogno di conoscere la posizione ma semplicemente il nome associato agli elementi presenti in una colonna. Segue che non avrà importanza nè l'ordine delle colonne nè l'ordine delle righe.

1.9.2 Tabella

Le relazioni sono rappresentate attraverso tabelle. Si ha una relazione quando:

- i valori di ogni colonna sono omogenei fra loro (quindi in una colonna avrò celle solo di un certo tipo: string, int...)
- le righe sono diverse fra loro (non ho doppioni)
- le intestazioni delle colonne sono diverse fra loro (non esistono più colonne con lo stesso attributo)

Possiamo stabilire un collegamento fra tabelle diverse mediante valori uguali in tuple diverse. Prendiamo per esempio una tabella *Studenti* e una *Esami*. Si stabilisce un collegamento mediante l'uguaglianza tra il campo *matricola* presente in *studenti* e il campo *studente* presente in *esami*!

Istanza L'istanza è un insieme di tuple in cui si ha un valore per ogni attributo[1].

Schema di relazione uno schema di relazione è caratterizzato dalla presenza del nome e di un insieme di attributi. Solitamente presenta la forma $R(A_1, \ldots, A_n)$ dove R è il nome della relazione ed $A_1, \ldots, A_n$ sono gli attributi.

Schema di base di dati Insieme di schemi di relazione. Può essere introdotto mediante la seguente forma: $R = \{R_1(X_1), \ldots, R_k(X_k)\}$.

Tupla su un insieme di attributi Una tupla su insieme di attributi X è una funzione che associa a ciascun attributo A in X il valore del dominio di A. Il valore della tupla t sull'attributo A può essere denotato mediante la forma $t[A]$.

1.9.3 Informazione incompleta

Il modello relazionale impone una struttura rigida per i dati: le tuple devono corrispondere agli schemi di relazione, ma potrebbe succedere che i dati disponibili non corrispondano al formato previsto.

Necessità di un valore nullo Introduciamo a tal proposito il valore $NULL$: di principio, ricordiamo, non esiste il vuoto ma un valore che denota il vuoto. Il NULL può essere utilizzato per denotare

- valori sconosciuti;
- valori inesistenti;
- valori non interessanti.

Il valore NULL si aggiunge a quelli parte del dominio di ciascun attributo. Segue che è necessario restringere la possibilità di inserimento di valori nulli in una relazione (in alcuni casi potrebbe essere necessario richiedere l'inserimento obbligatorio di un valore).

Esempio Aggiunta di una persona all'interno di una tabella: tra gli attributi abbiamo il *ruolo*, ma questa persona non ha ruoli o inizialmente non lo conosciamo. Grazie al valore NULL posso inserire delle tuple parziali aggiornabili successivamente.

1.10 Vincoli di integrità

Esistono istanze di basi di dati che pur essendo sintatticamente corrette non rappresentano informazioni possibili per il contesto. Il gestore, mediante una serie di proprietà (che esprimono la consistenza della base di dati), garantisce la qualità dei dati.

Definizione (Vincoli di integrità)**.** I vincoli corrispondono a proprietà del mondo reale modellato dalla base di dati e interessano tutte le istanze.

I vincoli di integrità

[1] Alle diapositive, 12,13 e 14 viene proposta un'alternativa basata su riferimenti espliciti gestiti dal sistema (puntatore). La cosa presenta dei vantaggi:indipendenza delle strutture fisiche e dati portabili più facilmente da un sistema ad un altro. L'utente finale, inoltre, vede gli stessi dati dei programmatori. I puntatori a livello fisico, tuttavia, non sono visibili a livello logico

- permettono una descrizione più accurata della realtà
- contribuiscono alla qualità dei dati
- sono utili nella progettazione
- sono usati dai DBMS nella esecuzione delle interrogazioni

I vincoli sono associati allo schema e si considerano corrette solo le istanze che soddisfano tutti i vincoli.

Definizione (Vincolo - definizione tecnica)**.** Un vincolo è un predicato che associa ad ogni istanze della base di dati il valore vero o falso. Se il predicato vale vero la proprietà è soddisfatta. Si distinguono **vincoli intrarelazionali** da **vincoli interrelazionali**.

1.10.1 Vincoli intrarelazionali

Un vincolo intrarelazionale è il vincolo di tupla: esprimono condizioni sui valori di ciascuna tupla, indipendentemente dalle altre. Più nello specifico possiamo parlare di vincolo di dominio se la condizione riguarda un solo attributo all'interno di una tupla.

Esempio 1 Ho una relazione *studenti*. Ciascun studente è identificato in modo univoco dal codice di matricola: segue che nella tabella non potranno essere presenti più studenti con la stessa matricoli.

Esempio 2 La lode può essere assegnata solo se il voto è uguale a 30. Non posso farlo con altre valutazioni!

Esempio 3 Lo studente può ottenere solo voti sufficienti: segue che nella tabella *esami* non potrò inserire un valutazione inferiore a 18.
In riferimento al primo esempio parleremo di *identificatori di tupla*: non posso avere due tuple con lo stesso valore dell'attributo matricola. Questo attributo consiste in una *chiave*!

Definizione (Chiave)**.** La chiave è un attributo che permette di identificare in modo univoco le tuple di una relazione. Precisamente:

- Un insieme K di attributi si dirà superchiave se date due tuple t_1 e $_2$ esse non sono distinte con $t_1[k] = t_2[k]$
- Lo stesso insieme sarà definito chiave se consiste in una *superchiave minimale*. Ciò significa che all'interno di K non si trovano altre superchiavi.

Prendiamo i seguenti esempi:

- L'insieme $\{Matricola\}$ è superchiave ed è anche chiave poichè ha un solo attributo e l'insieme vuoto non permette l'identificazione di tuple
- L'insieme $\{Cognome, Nome, Nascita\}$ è superchiave. Nessuno dei sottoinsiemi possibili è superchiave: Cognome e Nome, Cognome e Nascita, Nome e nascita. Segue che l'insieme è anche chiave poichè superchiave minimale.

- L'insieme $\{Matricola, Corso\}$ è superchiave ma non superchiave minimale: esiste un sottoinsieme proprio $\{Matricola\}$, che è superchiave minimale. Segue che il primo insieme non è chiave
- L'insieme $\{Nome, Corso\}$ non è superchiave perchè nella relazione possono comparire tuple uguali sia su nome che su corso.

1.10.1.1 A cosa servono le chiavi?

Ogni relazione contiene tuple distinte tra loro (definizione, è un insieme). Ogni relazione ha come superchiave l'insieme degli attributi su cui è definita, cioè ha almeno una chiave. L'esistenza della chiave permette di identificare in modo univoco ogni tupla garantendo l'accesso ad essa; è altrettanto utile per stabilire correlazioni fra dati di relazioni diverse.

Valori nulli Non è vietato che un attributo chiave assuma valori nulli, ma è sconsigliato generalmente. Presupponimo di avere una solo attributo chiave per identificare le tuple: in quel caso la chiave non può assumere valore nullo, altrimenti non potrei identificarla. Una chiave che non può assumere valori nulli è detta **chiave primaria** (rappresentata con sottolineatura).

1.10.2 Vincoli interrelazionali

Possiamo stabilire correlazioni tra relazioni diverse a partire dai valori delle **chiavi primarie**, in modo da mantenere la consistenza di una relazione in caso di modifica di un'altra.

Esempio Abbiamo una seconda relazione *esami*. Tra gli attributi si ha la matricola identificativa dello studente: non devo poter inserire esami con matricole inesistenti o con matricole esistenti se l'esame è già stato svolto. Ciò vale non solo con l'aggiunta di nuovi esami ma anche in caso di modifica di esami precedenti. Si richiede di intervenire anche quando lo studente viene eliminato (dove vanno a finire le tuple dei suoi esami?)

1.10.2.1 Vincolo di integrità referenziale

Definizione (Vincolo di integrità referenziale)**.** Un vincolo di integrità referenziale fra gli attributi X di una relazione R_1 impone ai valori su X in R_1 di comparire come valori della chiave primaria di R_2.

Integrità referenziali e valori nulli In presenza di valori nulli posso ricorrere a vincoli meno restrittivi. La definizione precedente è valida, semplicemente si impone quanto detto ai valori X diversi da NULL.

1.10.3 Violazioni e interferenze

- Nel caso di violazione di vincoli intrarelazionali si rifiuta l'operazione.
- Nel caso di vincoli interrelazionali si parla di operazioni interferenti che determinano violazioni dei vincoli stabiliti. Abbiamo
  - modifiche di valori (esempio modificare la matricola associata a un esame). L'operazione viene impedita

- azioni sulla tabella *master*: eliminazione di una tupla o modifica dell'attributo riferito. Posso stabilire:
  - l'eliminazione/modificata in cascata delle righe coinvolte;
  - posso porre a NULL il valore dell'attributo referente (nelle tuple rimaste "orfane");
  - assegnare un valore di default all'attributo referente;
  - non consentire la cancellazione.

2 — Mercoledì 11/03/2020

2.1 Interrogazioni

L'interrogazione è un'operazione di lettura che non modifica la base di dati. In certe circostanze un'interrogazione può coinvolgere più tabelle! Abbiamo due tipi di semantiche:

- **dichiarativa**, in cui si esprimono le proprietà del risultato. Essa ricorre al *calcolo relazionale* e consiste nell'effettiva semantica del linguaggio! Le interrogazioni sono espresse ad alto livello.
- **operazionale**, si specificano le modalità di calcolo adottate dal gestore della base di dati per ottenere il risultato. In questo caso si ricorre all'*algebra relazionale*. Poichè si parla di modalità di calcolo possiamo individuare i costi per l'esecuzione di una certa interrogazione

2.1.1 *query processor*

All'interno del gestore abbiamo un modulo detto *query processor*, dove è definito il processo di esecuzione delle interrogazioni. Ciò permette al gestore di definire una strategia di esecuzione ottimale. Si hanno tre fasi:

- **Analisi lessicale, sintattica e semantica** della query scritta: si ottiene una corrispondente espressione algebrica;
- **Ottimizzazione algebrica** dell'espressione: si ottiene una nuova espressione algebrica attraverso ottimizzazioni valide in ogni caso (Esempio: *push selections down* e *push projections down*)
- **Ottimizzazione basata sui costi**: sfruttando quanto contenuto nel cosiddetto *catalogo*, cioè informazioni quantitative (*profili*) riguardanti il database (numero di tuple, dimensione della tupla o dei valori, numero di valori distinti, valori minimi e massimi degli attributi...), possiamo stimare la dimensione dei risultati intermedi ed effettuare ulteriori ottimizzazioni (riparleremo di questa ottimizzazione nell'ultima lezione).

2.1.1.1 Ottimizzazione algebrica

Le due fasi di ottimizzazione si basano sull'euristica: sulla base dei dati a disposizione decidiamo quali procedure adottare per eseguire un'istruzione. Parlare di "euristica", tuttavia, è un po' improprio: il processo si basa soprattutto sulla nozione di equivalenza.

Equivalenza Due espressioni sono equivalenti se producono lo stesso risultato!

2.2 Algebra relazionale

Nell'algebra relazionale abbiamo operatori applicati su una o più relazioni. Gli operatori producono nuove relazioni e possono essere composti. Analizzeremo **operatori su insiemi** (unione, intersezione e differenza) ed **operatori su relazioni** (ridenominazione, selezione, proiezione, join). I primi operatori sono applicati a relazioni che presentano gli stessi attributi: la relazione prodotta avrà gli stessi attributi. I secondi, ad eccezione del join (il più complesso tra quelli che vedremo), sono monadici.

Nelle successive definizioni considereremo X come l'insieme degli attributi.

2.2.1 Unione ($r_1 \cup r_2$)

Date due relazioni $r_1(X), r_2(X)$, l'unione $r_1(X) \cup r_2(X)$ consiste in una nuova relazione che contiene sia le tuple di r_1 che quelle di r_2.

Esempio dalle diapositive Abbiamo le relazioni **laureati_triennali** e **laureati_magistrali**. Ottengo una nuova relazione che elenca le persone che hanno conseguito almeno una laurea. Nei risultati della nuova relazione vediamo Neri e Verdi che hanno conseguito sia la laurea triennale che quella magistrale, un altro Neri che ha solo la laurea magistrale, Rossi che ha solo la laurea triennale.

Laureati triennali

Matricola	Nome	Età
7274	Rossi	32
7432	Neri	24
9824	Verdi	25

Laureati magistrali

Matricola	Nome	Età
9297	Neri	33
7432	Neri	24
9824	Verdi	25

Laureati triennali ∪ Laureati magistrali

Matricola	Nome	Età
7274	Rossi	32
7432	Neri	24
9824	Verdi	25
9297	Neri	33

2.2.2 Intersezione ($r_1 \cap r_2$)

Date due relazioni $r_1(X), r_2(X)$, l'intersezione $r_1(X) \cap r_2(X)$ consiste in una nuova relazione che contiene solo le tuple appartenenti ad entrambe le relazioni.

Esempio dalle diapositive Prendiamo lo stesso esempio di prima. Ottengo una nuova relazione che elenca le persone che hanno preso sia la laurea magistrale che quella triennale. Gli unici che hanno conseguito entrambe le lauree sono Neri e Verdi: nell'istanza avrò soltanto le loro tuple.

Laureati triennali

Matricola	Nome	Età
7274	Rossi	32
7432	Neri	24
9824	Verdi	25

Laureati magistrali

Matricola	Nome	Età
9297	Neri	33
7432	Neri	24
9824	Verdi	25

Laureati triennali ∩ Laureati magistrali

Matricola	Nome	Età
7432	Neri	24
9824	Verdi	25

2.2.3 Differenza ($r_1 - r_2$)

Date due relazioni $r_1(X), r_2(X)$, la differenza $r_1(X) - r_2(X)$ consiste in una nuova relazione che contiene le tuple di r_1 che non appartengono anche ad r_2. L'operatore è rappresentato dal segno meno.

Esempio dalle diapositive Idem con patate. Ottengo una relazione in cui si hanno gli elementi di **laureati_triennali** non presenti in **laureati_magistrali**, cioè coloro che hanno conseguito la laurea triennale ma non quella magistrale.
Nella prima relazione abbiamo Rossi, Neri e Verdi: osserviamo che Neri e Verdi sono presenti anche nella seconda relazione. Segue che l'unica tupla presente nell'istanza sarà quella di Rossi.

Laureati triennali

Matricola	Nome	Età
7274	Rossi	32
7432	Neri	24
9824	Verdi	25

Laureati magistrali

Matricola	Nome	Età
9297	Neri	33
7432	Neri	24
9824	Verdi	25

Laureati triennali – Laureati magistrali

Matricola	Nome	Età
7274	Rossi	32

2.2.4 Ridenominazione ($\rho_{a_1,\ldots,a_n \leftarrow b_1,\ldots,b_n}$)

L'operatore monadico di ridenominazione ci permette di modificare lo schema di una relazione alterando gli attributi. Questa cosa ci permetterà di applicare gli operatori di insieme a relazioni che hanno schema simile ma non equivalente.

Esempio dalle diapositive Abbiamo le relazioni **paternita**($padre, figlio$) e **maternita**($madre, figlio$) a cui voglio applicare l'operatore unione. Essi hanno schema simile, ma un attributo diverso. Mediante l'operatore di ridenominazione modifico l'attributo *padre* in *genitore*. Faccio la stessa cosa con *madre*. A questo punto posso applico l'operatore di insieme.

$\rho_{Genitore \leftarrow Padre}$ (Paternità)

Genitore	Figlio
Adamo	Abele
Adamo	Caino
Abramo	Isacco

$\rho_{Genitore \leftarrow Madre}$ (Maternità)

Genitore	Figlio
Eva	Abele
Eva	Set
Sara	Isacco

$\rho_{Genitore \leftarrow Padre}$ (Paternità) $\cup$ $\rho_{Genitore \leftarrow Madre}$ (Maternità)

Genitore	Figlio
Adamo	Abele
Adamo	Caino
Abramo	Isacco
Eva	Abele
Eva	Set
Sara	Isacco

2.2.5 Selezione (σ_F) - decomposizione orizzontale

L'operatore monadico di selezione restituisce una nuova relazione con lo stesso schema ma con solo una parte delle tuple. Viene mostrato un sottoinsieme i cui elementi sono determinati a partire da un'espressione booleana F: se l'espressione è vera la tupla è considerata appartenente alla nuova relazione, altrimenti viene esclusa.

Esempio dalle diapositive Ho una relazione **Impiegati**($matricola, cognome, filiale, stipendio, eta$). Voglio mostrare soltanto gli impiegati con uno stipendio superiore ai 50.000 euro. Posso rendere la cosa più articolata mostrando soltanto gli impiegati che soddisfano la condizione precedente e che lavorano nella filiale di Milano.

$\sigma_{(Stipendio > 50000)\ AND\ (Filiale = 'Milano')}(Impiegati)$

Matricola	Cognome	Filiale	Stipendio
5998	Neri	Milano	64000

Attenzione ai valori nulli Prendiamo la stessa relazione e consideriamo le persone con $eta > 40$. Le tuple con età nulla vengono automaticamente escluse: ciò può essere evitato indicando che il valore può essere anche nullo (Ho le forme *IS NULL* e *IS NOT NULL*).

$$\sigma_{Eta>40}(Impiegati) \qquad \sigma_{(Eta>40)\ OR\ (Eta\ \textbf{IS NULL})}(Impiegati)$$

2.2.6 Proiezione (π_Y) - decomposizione verticale

L'operatore monadico di proiezione restituisce una nuova relazione che contiene un insieme di tuple ristrette agli attributi Y (ho un sottoinsieme degli attributi, fondamentalmente).

Esempio dalle diapositive Riprendiamo l'esempio utilizzato negli operatori di insieme: creo una nuova relazione in cui mostro soltanto matricole e cognomi.

Matricola	Cognome
7309	Neri
5998	Neri
9553	Rossi
5698	Rossi

$\pi_{Matricola,\ Cognome}(Impiegati)$

Cardinalità delle proiezioni Può capitare che una proiezione produca una relazione con un numero inferiore di tuple. Ciò può avvenire escludendo la chiave *matricola*. Osserviamo che sono presenti nella relazione iniziale due Neri e due Rossi. Nella relazione finale ho due Neri e un *solo* Rossi poichè abbiamo ignorato l'attributo che identifica le tuple in modo univoco: si osserva, inoltre, che entrambi i Rossi lavorano nella filiale di Roma.

Cognome	Filiale
Neri	Napoli
Neri	Milano
Rossi	Roma

$\pi_{Cognome,\ Filiale}(Impiegati)$

2.2.7 Composizione degli operatori

Come già detto posso creare una composizione di operatori: per esempio posso unire l'operatore selezione con l'operatore proiezione. Nella composizione è necessario porre attenzione all'ordine degli operatori: un ordine diverso può portare a risultati diversi.

Matricola	Cognome
7309	Rossi
5998	Neri
5698	Neri

$\pi_{\text{Matricola,Cognome}}\ (\ \sigma_{\text{Stipendio} > 50}\ (\text{Impiegati})\ )$

Adesso spingiamoci oltre: con gli operatori introdotti fino ad ora non possiamo correlare relazioni profondamente diverse tra loro!

2.2.8 Operatore Join

Con l'operatore JOIN ossiamo correlare dati appartenenti a relazione diverse!

2.2.8.1 Prodotto cartesiano ($r_1 \times r_2$)

Operatore binario, di fatto il più costoso per il DBMS. Abbiamo due relazioni: combiniamo le tuple di r_1 con le tuple di r_2. Ottengo una nuova relazione dove

- lo schema consiste nell'unione degli attributi delle due relazioni
- l'istanza è caratterizzata da tuple ottenute mediante il prodotto cartesiano matematico.

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparto	Capo
B	Mori
C	Bruni

R.Impiegato	R.Reparto	Q.Capo	Q.Reparto
Neri	B	Mori	B
Bianchi	B	Mori	B
Neri	B	Bruni	C
Bianchi	B	Bruni	C
Rossi	A	Bruni	C
Rossi	A	Mori	B

2.2.8.2 Join naturale ($r_1 \bowtie r_2$)

Il *join naturale* è un operatore binario che restituisce una relazione con schema uguale all'unione degli attributi degli schemi di $r_1(X_1)$ e $r_2(X_2)$ $(X_1 \cup X_2)$. Si individua che

$$r_1 \bowtie r_2 = \{t \in X_1 \cup X_2 \mid t[X_1] \in r_1, t[X_2] \in r_2\}$$

Il join naturale consiste in un insieme di tuple da cui ottengo tuple appartenenti alla relazione r_1 se applico una proiezione limitata a X_1 e tuple appartenenti alla relazione r_2 se applico una proiezione limitata a X_2. Abbiamo due situazioni:

- **presenza di attributi con lo stesso nome**: le tuple vengono costituite se il valore degli attributi comuni è uguale
- **senza attributi con nome comune**: quello che si ottiene equivale al prodotto cartesiano. Ho un numero di tuple pari al prodotto delle cardinalità degli operandi (le tuple sono tutte combinabili)

Osservazione Solitamente non è possibile tornare alle relazioni originali dopo aver applicato il join.

Esempio dalle diapositive Riprendiamo le relazioni *impiegati* e *caporeparti*. Rossi non è tra le tuple poichè non è combinabile con nessun capo reparto. Si osserva che il capo del reparto C non può essere combinato con nessun impiegato (non si hanno impiegati al reparto C).

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparto	Capo
B	Mori
C	Bruni

Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Mori

Potrebbe succedere che per un reparto (il B in questo caso) si abbiano due capireparto: Neri e Bianchi (del reparto B) vengono combinati due volte!

Impiegato	Reparto
Neri	B
Bianchi	B
Verdi	A

Reparto	Capo
B	Mori
B	Bruni
A	Bini

Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Bruni
Neri	B	Bruni
Bianchi	B	Mori
Verdi	A	Bini

Esempio senza attributi comuni

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Codice	Capo
A	Mori
B	Bruni

Impiegati $\triangleright\triangleleft$ Reparti

Impiegato	Reparto	Codice	Capo
Rossi	A	A	Mori
Rossi	A	B	Bruni
Neri	B	A	Mori
Neri	B	B	Bruni
Bianchi	B	A	Mori
Bianchi	B	B	Bruni

In generale

- Date due relazioni $R_1(X_1), R_2(X_2)$ si osserva che la proiezione di X_1 del join naturale tra R_1 e R_2 è inclusa in R_1.

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

  Se non si hanno attributi comuni avviene il prodotto cartesiano e il risultato dell'espressione coincide con R_1. Se si hanno attributi comuni il risultato non coincide con il prodotto

cartesiano e due tuple vengono associate solo se si hanno le condizioni. Segue che il non-JOIN di alcune tabelle potrebbe comportarmi l'esclusione di alcune tuple di R_1 e quindi avere come risultato un sottoinsieme di R_1.

- Data una relazione $R(X)$, dove $X = X_1 \cup X_2$, il join naturale tra la proiezione di X_1 di R e la proiezione di X_2 di R contiene R

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

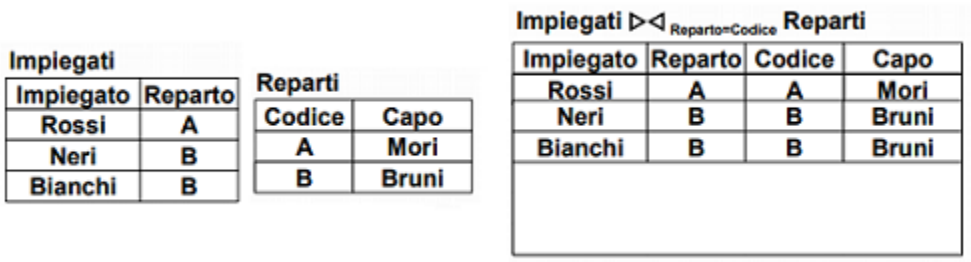
  Se non si hanno attributi comuni tra X_1 e X_2 segue un prodotto cartesiano che mi porta ad avere un numero di tuple maggiore rispetto a prima. Ho quindi un insieme più grande che contiene l'insieme R iniziale. Se ho attributi comuni tra X_1 e X_2 allora può esserci la possibilità che il risultato sia uguale ad R originario.

2.2.8.3 *theta-join* (spesso *equi-join*)

Posso ridurre il prodotto cartesiano attraverso una relazione del tipo $\sigma_F(R_1 \times R_2)$. La stessa operazione può essere fatta attraverso un operatore derivato detto *theta-join*: prendo soltanto le tuple che fanno JOIN e che soddisfano delle condizioni.

$$R_1 \bowtie_F R_2$$

La condizione F è spesso una congiunzioni di atomi di confronto caratterizzati da un operatore di confronto e attributi di relazioni diverse. Se l'operatore di confronto è l'uguale allora si ha l'*equi-join*.
Riprendiamo l'esempio di prima dove non si hanno attributi comuni. Svolgiamo l'equi-join stabilendo un legame valido tra tuple se Reparto=Codice.



Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

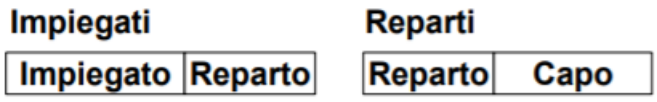
Reparti

Codice	Capo
A	Mori
B	Bruni

Impiegati $\triangleright\triangleleft_{\text{Reparto=Codice}}$ Reparti

Impiegato	Reparto	Codice	Capo
Rossi	A	A	Mori
Neri	B	B	Bruni
Bianchi	B	B	Bruni

2.2.9 JOIN operatore non primitivo



Impiegati

Impiegato	Reparto

Reparti

Reparto	Capo

Il Join non è un operatore primitivo (diciamo, un "prodotto cartesiano schietto"). Individuiamo che

Relazione tra JOIN Naturale e prodotto cartesiano

$Impiegati \bowtie Reparti = \pi_{Impiegato,Reparto,Capo}(\sigma_{I.reparto=reparto}(\rho_{I.reparto \leftarrow Reparto}(Impiegati) \times Reparti))$

- Applico la ridenominazione alla relazione *Impiegati* per poter usare la selezione
- Eseguo il prodotto cartesiano tra la relazione $\rho_{I.Reparto \leftarrow Reparto}(Impiegati)$ e *Reparti*.
- Applico la selezione, scegliendo i record che soddisfano la condizione $I.Reparto = Reparto$
- Proietto gli attributi $Impiegato, Reparto, Capo$. $I.Reparto$ ha lo stesso valore di *Reparto*, non serve proiettarlo nuovamente.

Relazione tra JOIN Naturale ed equi-join

$Impiegati \bowtie Reparti = \pi_{Impiegato,Reparto,Capo}(\rho_{I.reparto \leftarrow Reparto}(Impiegati) \bowtie_{I.Reparto=Reparto} Reparti)$

- Applico la ridenominazione alla relazione *Impiegati* per poter usare la selezione
- Eseguo l'equi-JOIN tra la relazione $\rho_{I.Reparto \leftarrow Reparto}(Impiegati)$ e *Reparti*, scegliendo soltanto le combinazioni di record che soddisfano l'uguaglianza $I.Reparto = Reparto$
- Proietto gli attributi $Impiegato, Reparto, Capo$. $I.Reparto$ ha lo stesso valore di *Reparto*, non serve proiettarlo nuovamente.

2.2.10 Equivalenza di espressioni

Due espressioni sono equivalenti se producono lo stesso risultato qualunque sia l'istanza attuale della base di dati. L'equivalenza è un concetto importante poichè i DBMS, molto spesso, eseguono espressioni equivalenti a quelle date, meno costose. Vediamo due esempi di euristica fondamentale in cui si applica uno dei principi base dell'ottimizzazione: esecuzione di selezioni e proiezioni il più presto possibile per ridurre la dimensioni dei risultati intermedi (e quindi il costo dell'operazione)

2.2.10.1 *Pushing selections down*

Dato un attributo A di R_2, che è attributo di interesse, individuo che

$$\sigma_{A=10}(R_1 \bowtie R_2) = R_1 \bowtie \sigma_{A=10}(R_2)$$

Le condizioni della selezione riguardano un attributo di R_2: eseguo la selezione prima di fare JOIN.

2.2.10.2 *Pushing projections down*

Date due relazioni $R_1(X_1)$ e $R_2(X_2)$ con $Y_2 \subseteq X_2$, individuo che

$$\pi_{X_1 Y_2}(R_1 \bowtie R_2) = R_1 \bowtie \pi_{Y_2}(R_2)$$

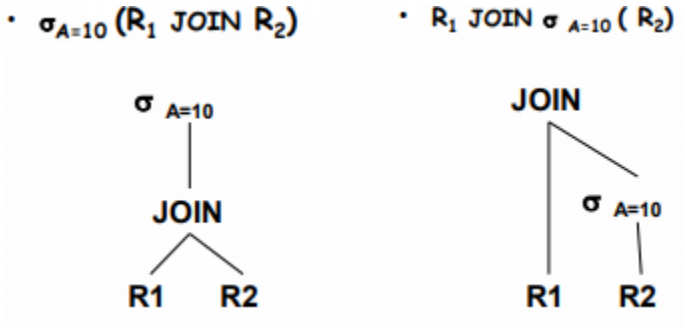
Proietto solo una parte degli attributi di X_2: eseguo la proiezione prima di fare JOIN.

2.2.11 Procedura euristica dell'ottimizzatore

- Decomporre le selezioni congiuntive in successive selezioni atomiche
- Anticipare il più possibile le selezioni
- In una sequenza di selezioni anticipare le più selettive
- Combinare prodotti cartesiani e selezioni per formare JOIN
- Anticipare il più possibile le proiezioni (anche introducendone di nuove)

2.2.12 Alberi per la rappresentazione di interrogazioni

Possiamo utilizzare gli alberi per rappresentare le nostre interrogazioni: le foglie consistono nei dati (relazioni, file), i nodi intermedi negli operatori applicati.



3 — Mercoledì 18/03/2020

La scorsa volta abbiamo intodotto l'algebra relazionale per spiegare le procedure adottate dal DBMS per eseguire le nostre interrogazioni. Adesso analizzeremo la semantica effettiva del linguaggio: il calcolo relazionale. Sappiamo che in SQL non è necessario scrivere una query già ottimizzata: possiamo scrivere versioni meno efficienti che saranno sottoposte a un processo di ottimizzazione.

3.1 Calcolo relazionale

Con **calcolo relazionale** intendiamo una famiglia di linguaggi dichiarativi che permette di specificare le proprietà del risultato di una certa interrogazione. Ne abbiamo due versioni:

- il calcolo relazionale **applicato ai domini** (che presenta in modo naturale le proprietà dei predicati)
- il calcolo **su tuple con dichiarazioni di range** (versione adottata da SQL)

3.1.1 Calcolo su domini

$$\{A_1 : x_1, \ldots, A_n : x_n | f\}$$

- $A_1, \ldots, A_n$ sono nomi di attributi, $x_1, \ldots, x_n$ sono nomi di variabili
- La lista delle coppie $A_i : x_i$ è detta *target list* e definisce la struttura del risultato
- f è una formula. Al suo interno possiamo porre:
  - $R(A_1 : x_1, \ldots, A_n : x_n)$, dove $R(A_1 \ldots A_n)$ è uno schema di relazione $x_1 \ldots x_n$ sono variabili. Vera sui valori $x_1, \ldots, x_n$ che formano una tupla della relazione r sullo schema R, nell'istanza di base di dati a cui l'espressione viene applicata.
  - $x\theta y$, $x\theta c$: x, y sono variabili, c è una costante e θ è un operatore di confronto $(=, \neq, \leq, \geq, >, <)$. Vera sui valori x, y, c che soddisfano il confronto con operatore θ.
  - quantificatori nella forma

    $$\exists x(f) \quad \forall x(f)$$

    Cioè: *Esiste almeno una variabile x che soddisfa la formula f*

  Se f_1 e f_2 sono formule allora lo sono anche $\neg f_1$, $\neg f_2$, $f_1 \wedge f_2$, $f_1 \vee f_2$.

3.1.2 Calcolo su tuple con dichiarazione di range

$$\{x_1.Z_1, \ldots, x_n.Z_n | x_i(R_1), \ldots, x_j(R_m) | f\}$$

- $x_1.Z_1, \ldots, x_n.Z_n$ è la *target list* (con x_i nome di tupla e Z_j insieme di nomi di attributi)
- $x_i(R_1), \ldots, x_j(R_m)$ è la *range list* (campo di variabilità delle variabili)
- f è una formula. Al suo interno possiamo porre:
  - formule atomiche del tipo $x_i.Z_i \, \theta \, x_j.Z_j$ o $x_i.Z_i \, \theta \, c$ (dove c è una costante).
  - connettivi come nel calcolo su domini
  - quantificatori nella seguente forma

$$\exists x(R)(f) \qquad \forall x(R)(f)$$

Cioè: *Esiste nella relazione R almeno una variabile x che soddisfa la formula f*

3.1.3 Quantificatori esistenziali e universali

All'interno dei predicati possiamo introdurre i cosiddetti *quantificatori*. Abbiamo il quantificatore esistenziale $\exists$ e quello universale $\forall$. I due sono intercambiabili, nel senso che ne basta uno per esprimere qualunque cosa. Le *leggi di de Morgan* valgono, *mutatis mutandis*, anche per i quantificatori:

- $\exists x(f) = \neg(\forall x(\neg(f)))$
- $\forall x(f) = \neg(\exists x(\neg(f)))$

Ricordiamo le leggi di De Morgan, già viste a *Fondamenti di programmazione*

$$\boxed{\neg(f \wedge g) = \neg(f) \vee \neg(g)} \qquad \boxed{\neg(f \vee g) = \neg(f) \wedge \neg(g)}$$

Date due condizioni f e g:

1. La negazione dell'AND si ha quando almeno una delle condizioni è negata
2. La negazione dell'OR si ha quando entrambe le condizioni sono negate

Quando sono necessari? I quantificatori possono essere omessi in certe circostanze, ma in altre sono obbligatori: parliamo di interrogazioni più complesse come, per esempio, la differenza.

3.1.4 Esempi

Base di dati per gli esempi

```
IMPIEGATI(Matricola, Nome, Età, Stipendio)
SUPERVISIONE(Capo, Impiegato)
```

Trovare gli impiegati che guadagnano più di 40 milioni

- **Su domini**: { Matricola: m, Nome : n, Età: e, Stipendio: s | Impiegati(Matricola: m, Nome: n, Età: e, Stipendio: s) $\wedge$ s $>$ 40 }
- **Su tuple con dichiarazione di range**: {i.* | i(Impiegati) | i.Stipendio $>$ 40}

Trovare nome e matricola degli impiegati che guadagnano più di 40 milioni

- **Su domini**: { Matricola: m, Nome: n | Impiegati(Matricola: m, Nome: n, Età: e, Stipendio: s) $\wedge$ s $>$ 40}
- **Su tuple con dichiarazione di range**: { i.(Matricola, Nome) | i(Impiegati) | i.Stipendio $>$ 40 }

Trovare matricola e nome dei capi i cui impiegati guadagnano tutti più di 40 milioni

- **Primo metodo con calcolo sui domini**: { Matricola: c, Nome: n | Impiegati(Matricola: c, Nome: n, Età: e, Stipendio: s) $\wedge$ $\forall m'(\forall n'(\forall e'(\forall s'(Impiegati$(Matricola: m', Nome: n', Età: e', Stipendio: s'} $\wedge$ Supervisione(Capo:c, Impiegato:m') $\wedge$ s' $>$ 40))))}
- **Secondo metodo con calcolo sui domini**: { Matricola: c, Nome: n | Impiegati(Matricola: c, Nome: n, Età: e, Stipendio: s) $\wedge$ $\neg(\exists m'(\exists n'(\exists e'(\exists s'(Impiegati$(Matricola: m', Nome: n', Età: e', Stipendio: s'} $\wedge$ Supervisione(Capo:c, Impiegato:m') $\wedge$ s' $\leq$ 40)))))}
- **Terzo metodo con calcolo su tuple**: {i.(Matricola, Nome) | s(Supervisione), i(Impiegati) | i.Matricola = s.Capo $\wedge$ $\neg(\exists i$(Impiegati)(s.Impiegato=i.Matricola $\wedge$ i.Stipendio $\leq$ 40))))}

3.1.5 Discussione sul calcolo su domini

Il calcolo su domini è dichiarativo, ma eccessivamente verboso con possibilità di scrivere espressioni sensa senso dipendenti dal dominio e aventi risultati di grandi dimensione. Nell'algebra, invece, tutte le espressioni hanno un senso e sono indipendenti dal dominio.

Indipendenza dal dominio Un'espressione si dice indipendente dal dominio se il suo risultato, su ciascuna istanza di base di dati, non varia al variare del dominio rispetto al quale l'espressione è valutata.

Esempio di espressione dipendente dal dominio Un esempio di espressione dipendente dal dominio è la seguente:

$$\{A_1 : x_1, A_2 : x_2 \mid R(A_1 : x_1, A_2 : x_2) \wedge x_2 = x_2\}$$

$x_2 = x_2$ è una condizione sempre vera: se il dominio include gli interi da 0 a 99 avremo 100 tuple, se invece il dominio va da 0 a 999 avremo 1000 tuple!

3.1.6 Discussione sul calcolo su tuple

Le variabili rappresentano tuple e si ha minore verbosità. Alcune interrogazioni importanti non possono essere espresse, in particolare le unioni:

$$R_1(AB) \cup R_2(AB)$$

Ogni variabile nel risultato ha un solo range, mentre vorremmo tuple sia della prima relazione che della seconda. Intersezione e differenza, comunque sia, sono esprimibili. Per questa motivazione SQL prevede esplicitamente un operatore unione, ma non tutte le versioni prevedono intersezione e differenza.

3.1.7 Calcolo e algebra relazionale

Calcolo e algebra relazionale sono "equivalenti":

- Per ogni espressione del calcolo relazionale che sia indipendente dal dominio esiste un'espressione dell'algebra relazionale equivalente a essa
- Per ogni espressione dell'algebra relazionale esiste un'espressione del calcolo relazionale equivalente a essa (e di conseguenza indipendente dal dominio)

Possono espresse buona parte delle interrogazioni, però ci sono cose non esprimibili come la *chiusura transitiva*.

3.1.8 Chiusura transitiva di una relazione

La chiusura transitiva non può essere definita poichè dovrei fare il JOIN innumerevoli volte per arrivare al risultato, alla tabella definitiva.

Esempio Consideriamo la seguente relazione

`Supervisione(Impiegato, Capo)`

Voglio trovare, per ogni impiegato, tutti i superiori (cioè il capo, il capo del capo, e così via). Nel seguente esempio (teniamo conto che vi è la ridenominazione dell'attributo *Campo*) la cosa pare semplice

Impiegato	Capo
Rossi	Lupi
Neri	Bruni
Lupi	Falchi

Impiegato	Superiore
Rossi	Lupi
Neri	Bruni
Lupi	Falchi
Rossi	Falchi

ma se abbiamo una nuova n-upla come nel secondo esempio risulta necessario stabilire ulteriori legami. Se il capo di Rossi è Lupi e il capo di Lupi è Falchi allora il capo di Rossi è Falchi.

Impiegato	Capo
Rossi	Lupi
Neri	Bruni
Lupi	Falchi
Falchi	Leoni

Impiegato	Superiore
Rossi	Lupi
Neri	Bruni
Lupi	Falchi
Rossi	Falchi
Lupi	Leoni
Rossi	Leoni

Conclusione Non possiamo calcolare la chiusura transitiva di una relazione qualunque. In algebra relazionale l'operazione si simulerebbe con un numero di JOIN illimitato.

3.2 Estensione dell'algebra relazionale

Il modello relazionale può essere facilmente estesso a comprendere operatori SQL non direttamente riconducibili agli operatori algebrici introdotti. Ciò non comporta una modifica del modello.

3.2.1 JOIN Esterno

Ho tre tipi di OUTER JOIN: left, right e full. Questi permettono di combinare tuple che normalmente non fanno JOIN.

- Il LEFT JOIN: mantiene tutte le tuple della prima relazione
- Il RIGHT OUTER JOIN: mantiene tutte le tuple della seconda relazione
- Il FULL OUTER JOIN: mantiene tutte le tuple

Esempio di JOIN *Trovare la matricola dei capi degli impiegati che guadagnano tutti più di 40.000 euro*

Impiegati

Matricola	Nome	Età	Stipendio
7309	Rossi	34	45000
5998	Bianchi	37	38000
9553	Neri	42	35000
5698	Bruni	43	42000
4076	Mori	45	50000
8123	Lupi	46	60000

Supervisione

Impiegato	Capo
7309	5698
5998	5698
9553	4076
5698	4076
4076	8123

Si individuano impiegati non subordinati a nessuno, cioè i capi stessi. Con un JOIN normale (con I.Matricola e S.Impiegato) i record riguardanti questi impiegati sparirebbero. Facciamo il JOIN sinistro

$$\pi_{Capo}(\sigma_{MatricolaISNULL}(Supervisione =\bowtie_{Impiegato=Matricola} \sigma_{Stipendio\leq 40.000}(Impiegati)))$$

ottenendo

Matricola	Nome	Età	Stipendio	Impiegato	Capo
5998	Bianchi	37	38000	5998	5698
9553	Neri	42	35000	9553	4076
NULL	NULL	NULL	NULL	4076	8123

Ho due impiegati che soddisfano le condizioni e un capo che non ha impiegati che guadagnano più di 40.000 euro. Ho tuple che fanno JOIN e tuple che non hanno fatto JOIN. Quanto fatto permette di svolgere l'operazione di differenza, che vedremo più avanti

3.2.2 Proiezione generalizzata

$$\pi_{F_1,F_2,F_3}(E)$$

Dove F_1, F_2, F_3 sono espressioni aritmetiche su attributi di E e costanti. Creo "un nuovo" attributo derivante da un'espressione algebrica dipendente da altri attributi e da costanti. Vediamo un esempio con la seguente tabella

Conto

Cliente	Credito	Spese
Andrea	6000	1000
Andrea	4000	500
Maria	10000	2000
Anna	3000	1500
Filippo	3000	1000
Luigi	5000	1800
Franco	5000	2000
Maria	6000	2000
Andrea	10000	5000
Anna	5000	1000

Posso scrivere, per esempio $\pi_{Cliente,Credito-Spese}(Conto)$, ottenendo

Conto

Cliente	Credito	Spese
Andrea	6000	1000
Andrea	4000	500
Maria	10000	2000
Anna	3000	1500
Filippo	3000	1000
Luigi	5000	1800
Franco	5000	2000
Maria	6000	2000
Andrea	10000	5000
Anna	5000	1000

3.2.3 Funzioni aggregate

Si possono usare nelle espressioni dei nomi di funzione (operatori) che si applicano a insiemi e producono un valore scalare come risultato. Gli operatori aggregati sono:

- **Somma**: $sum_{Spese}(Conto)$
- **Massimo**: $max_{Credito}(Conto)$
- **Minimo**: $min_{Credito}(Conto)$
- **Conteggio**: $count_{Cliente}(Conto)$
- **Conteggio di elementi distinti**: count-distinct$_{Cliente}(Conto)$

3.2.4 Raggruppamento

Posso raggruppare gli elementi di una relazione attraverso un apposito operatore. Posso raggruppare dei conti in base al codice cliente ravvicinando i conti correnti dei vari clienti. Posso applicare a questi gruppi degli operatori aggregati e ottenere un result set con una riga per ogni cliente.

$$_{\textbf{Cliente}}G_{sum(Credito)}(Conto)$$

Posso avere più attributi con cui compiere il raggruppamento (a sinistra), ma anche più funzioni di aggregazione (a destra).

3.2.5 Divisione

[1] Operatore di tipo universale (parola chiave: tutti). Posso porlo sintatticamente all'interno di una query, ovviamente non è primitivo perchè corrisponde a un'espressione estremamente complessa.

Esempio *Trovare i nomi dei clienti che hanno un conto corrente in tutte le filiali di banca di Pisa.*
Abbiamo le seguenti relazioni:

```
Branch(bank_name,branch_name, branch_city)
Account(branch_name, bank_name, account_number, branch_city)
Depositor(account_number, customer_name)
```

Possiamo risolvere l'esercizio attraverso la seguente espressione algebrica:

$$\Pi_{CN,BrN}(depositor \bowtie account) \div \Pi_{BrN}(\sigma_{BC='Pisa'}(branch))$$

Il risultato sono i clienti che hanno un conto in tutte le filiali di Pisa. L'espressione precedente, derivata, corrisponde alla seguente

$$\begin{aligned}&\Pi_{CN}(depositor \bowtie \sigma_{BC='Pisa'}(account)) - \Pi_{CN}(\\ &((\Pi_{CN}(depositor \bowtie \sigma_{BC='Pisa}(account)) \bowtie \Pi_{BrN}(\sigma_{BC='Pisa}(branch))\\ &-\\ &\Pi_{CN,BrN}(depositor \bowtie account)\\ &)\end{aligned}$$

- Prendo i clienti che esistono e hanno almeno un conto in una filiale di Pisa.
- Attraverso un prodotto cartesiano (osservo dalle proiezioni che non metto insieme tuple con attributi comuni) genero tutte le coppie possibili tra i clienti che hanno conti a Pisa e le sedi di Pisa. Il risultato intermedio è caratterizzato da coppie veritiere (coloro che hanno effettivamente un conto in quella sede) e coppie false (persone che non hanno un conto in quella sede)
- Da questo prodotto cartesiano sottraggo le coppie veritiere (tutte, incluse quelle non di Pisa)

[1] Sezione da imparare come l'ave maria. La Vaglini chiede molto spesso di spiegare la divisione. Ringrazio Giovanni Ligato per avermi spiegato la cosa il giorno prima dell'orale (mi ha letteralmente salvato la vita).

- La differenza più esterna mi porta ad ottenere coloro che hanno conti solo a Pisa.
- **Persona che ha conti solo a Pisa**: le coppie ottenute attraverso il prodotto cartesiano sono tutte veritiere. Segue che la differenza più interna le rimuoverà. Quindi la differenza più esterna non mi va a rimuovere questa persona.
- **Persona che non ha conti in tutte le filiali di Pisa**: le coppie ottenute attraverso il prodotto cartesiano sono in parte veritiere e in parte false. Attraverso la differenza più interna rimuovo le coppie veritiere. Mi rimangono le coppie false. Se ci sono coppie false significa che l'utente non ha conti in tutte le filiali di Pisa

3.3 Relazioni derivate

Relazioni di base Presentano un contenuto autonomo

Relazioni derivate Relazioni il cui contenuto è funzione del contenuto di altre relazioni. Quando compiamo un'interrogazione costruiamo dei risultati intermedi senza intaccare la base di dati. Potrei avere il bisogno di recuperare un risultato (anche più di una volta) assegnandogli un nome. Ciò consiste in una *vista*. Essa può essere materializzata (salvata e usata più volte) o virtuale (usata una volta sola).

- **Viste materializzate**: La memorizzazione risulta vantaggiosa perchè non hai da fare il calcolo tutte le volte (si hanno delle tabelle in carne ed ossa, ripensare a quanto visto con Pistolesi). I risultati sono immediatamente disponibili, ma potrebbero essere ridondanti e appesantire. Le viste materializzate non sono supportate da tutti i DBMS
- **Viste virtuali**: La vista virtuale è sempre ricalcolata ed è supportata da tutti i DBMS (nel db salviamo non il result set, ma lo snippet - il testo dell'interrogazione).

Colleghiamo a quanto fatto fino ad ora Ottenere una vista significare dare un nome a un espressione. Esempio:

$$Supervisione = \pi_{Impiegato,Capo}(Afferenza \bowtie Direzione)$$

Posso adottare questa vista anche in altre espressioni:

$$\sigma_{Capo='Leoni'}(Supervisioni)$$

viene eseguita come

$$\sigma_{Capo='Leoni'}(\pi_{Impiegato,Capo}(Afferenza \bowtie Direzione))$$

Vantaggi per il programmatore Possiamo semplificare la scrittura di interrogazioni seguendo il metodo del *divide et impera* (divisione di un problema in sottoproblemi)

4 — Mercoledì 25/03/2020

4.1 Ciclo di vita

Il progetto è un'attività essenziale nella realizzazione di un software. Si parte da una serie di esgienze e si arriva a una base di dati che le soddisfa. Tutte le attività svolte per arrivare all'oggetto desiderato vanno a formare il **ciclo di vita**.

Definizione (Ciclo di vita)**.** Sequenza di attività, anche ripetute ciclicamente, svolte da analisti, progettisti, utenti, nello sviluppo e nell'uso dei sistemi software.

Le fasi tipiche del ciclo di vita sono:

- **Raccolta e analisi dei requisiti**: si cerca di capire cosa bisogna fare, quali proprietà dovrà avere il software da progettare;
- **Progettazione**: individuazione delle funzionalità richieste dal sistema e dei dati necessari;
- **Realizzazione** vera e propria;
- **Collaudo**: sperimentazione, verifica che quanto progettato sia corretto;
- **Operatività**: il sistema diventa operativo.

4.2 Concentriamoci sulla progettazione

La progettazione riguarda sia i dati che le funzionalità del sistema. Nel caso dei sistemi informativi (questi) la parte fondamentale è quella riguardante la progettazione dei dati.

Buon progetto Un buon progetto presenta un'adeguata documentazione di quanto fatto. Si usano dei modelli per rappresentare i dati in modo formale, preciso e comprensibile: utilizzo quindi dei linguaggi che permettano di seguire tutta la fase di progettazione e verificare durante il progetto il modo in cui i requisiti sono rispettati.

Modello per il ciclo di vita Esistono moltissimi modelli, il più vecchio e utilizzato è il *Waterfall model*. Le fasi di vita sono attraversate in sequenza una dopo l'altra. ogni volta che si passa a una fase viene chiusa la precedente e questa non viene più ripetuta.
Qualunque sia il modello adottato si hanno delle sottofasi:

- **Acquisizione dei requisiti**: dipende molto dai rapporti umani tra cliente e sviluppatore. Si tiene conto anche di normative, regolamenti e procedure aziendali, realizzazioni già esistenti

- **Analisi dei requisiti**: presenti una serie di linguaggi necessari per definire i requisiti da più punti di vista

Come si acquisiscono i requisiti? I requisiti possono essere acquisiti mediante documentazione (normative, regolamenti e procedure aziendali, realizzazioni già esistenti) e direttamente dall'utente. Questo aspetto è il più complesso nel processo di acquisizione: gli utenti potrebbero fornire informazioni diverse (chi sta in alto potrebbe avere visione più ampia, ma meno dettagliata). Segue la necessità, nel corso della progettazione, di parlare più volte con chi di dovuto *raffinando* le informazioni già in nostro possesso.

Ranking dei requisiti Nella fase di progettazione è necessario tenere conto del *ranking* dei requisiti: alcuni sono più importanti di altri.

Quando siamo in possesso della documentazione necessaria? La regola fondamentale è leggere ogni singola cosa in nostro possesso: una singola parola può fare la differenza! Raggruppiamo le frasi generali e le frasi relative ai vari concetti. Per chiarire il termine di concetto solitamente si inserisce nel progetto un *Glossario* (per ciascun elemento si indica una descrizione, eventuali sinonimi e collegamenti).

4.2.1 Esempio concreto degli ultimi passi spiegati

Si vuole realizzare una base di dati per una società che eroga corsi: di ogni corso vogliamo rappresentare i dati dei partecipanti e dei docenti. Per gli studenti (circa 5000), identificati da un codice, si vuole memorizzare il codice fiscale, il cognome, l'età, il sesso, il luogo di nascita, il nome dei loro attuali datori di lavoro, i posti dove hanno lavorato in precedenza insieme al periodo, l'indirizzo e il numero di telefono, i corsi che hanno già frequentato (le materie sono in tutto circa 200) e il giudizio finale.

4.2.1.1 Glossario dei termini

Termine	Descrizione	Sinonimi	Collegamenti
Partecipante	Persona che partecipa ai corsi	Studente	Corso, Società
Docente	Docenti dei corsi, che può essere esterno.	Insegnate	Corso
Corso	Corso organizzato dalla società. Può avere più edizioni.	Materia	Docente
Società	Ente presso cui i partecipanti lavorano o hanno lavorato	Posti	Partecipante

4.2.1.2 Frasi di carattere generale

Si vuole realizzare una base di dati per una società che eroga corsi: di ogni corso vogliamo rappresentare i dati dei partecipanti e dei docenti.

4.2.1.3 Frase relative ai partecipanti

Per i partecipanti (circa 5000), identificati da un codice, rappresentiamo il codice fiscale, il cognome, l'età, il sesso, la città di nascita, i nomi dei loro attuali datori di lavoro e di quelli

precedenti (insieme alle date di inizio e fine rapporto), le edizioni dei corsi che stanno attualmente frequentando e quelli che hanno frequentato nel passato, con la relativa votazione finale in decimi.

4.2.1.4 Frasi relative ai datori di lavoro

Relativamente ai datori di lavoro presenti e passati dei partecipanti, rappresentiamo il nome, l'indirizzo e il numero di telefono.

4.2.1.5 Frasi relative ai corsi

Per i corsi (circa 200), rappresentiamo il titolo e il codice, le varie edizioni con date di inizio e fine e, per ogni edizione, rappresentiamo il numero di partecipanti e il giorno della settimana, le aule e le ore dove sono tenute le lezioni.

4.2.1.6 Frasi relative a tipi specifici di partecipanti - sottoclassi di partecipanti a cui si vuole dare una rappresentazione particolare

Per i partecipanti che sono liberi professionisti, rappresentiamo l'area di interesse e, se lo possiedono, il titolo professionale. Per i partecipanti che sono dipendenti, rappresentiamo invece il loro livello e la posizione ricoperta.

4.2.1.7 Frasi relative ai docenti

Per i docenti (circa 300), rappresentiamo il cognome, l'età, la città di nascita, tutti i numeri di telefono, il titolo del corso che insegnano, di quelli che hanno insegnato nel passato e di quelli che possono insegnare. I docenti possono essere dipendenti interni della società di formazione o collaboratori esterni.

4.2.1.8 Morale della favola

Non si inizia la progettazione finchè non si è letto tutto in modo adeguato.

4.2.2 Progettazione per livelli di astrazione

Si hanno vari livelli di astrazione per la progettazione:

- **Livello concettuale**, il più esterno: esprime come deve essere fatto il sistema per rispettare i requisiti. Permette di vedere, nel nostro caso, l'organizzazione della base di dati
- **Livello logico**: si dice qualcosa sull'organizzazione dei dati. Le relazioni utilizzate, come sono collegate tra di loro...
- **Livello fisico**: vediamo dei blocchi in memoria secondaria. Si decide l'allocazione dei dati e le modalità di accesso nei momenti di lettura.[1]

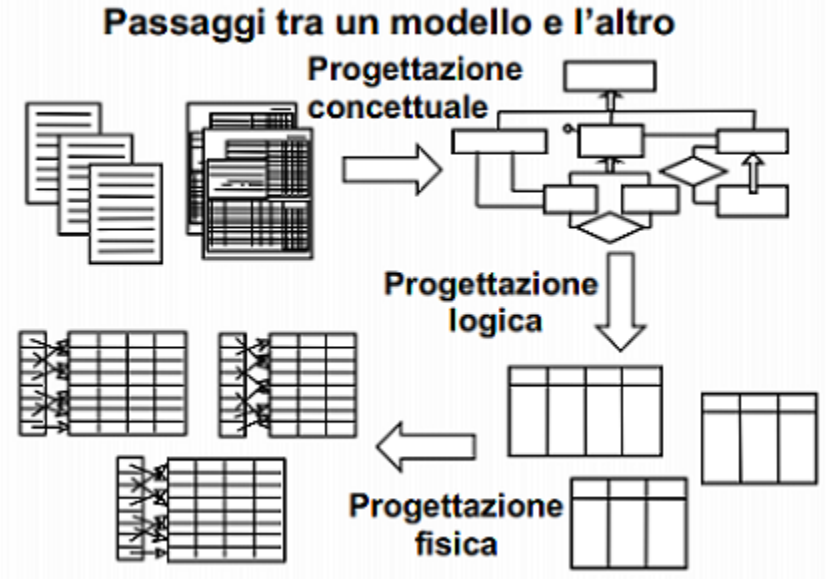
[1] Un esempio di cosa da fare potrebbe essere uno studio sulle relazioni per valutare la creazione di uno o più indici, in modo tale da velocizzare le operazioni più frequenti

Questi livelli di astrazione corrispondono a degli schemi in cui si usano modelli e linguaggi differenti per descrivere ciò che si rappresenta in ogni livello.

Come si passa da un livello a un altro? Facciamo una progettazione concettuale, li mettiamo nella rappresentazione, li verifichiamo e se tutto è valido compiamo un passaggio automatico da progettazione concettuale a progettazione logica. Se il progetto concettuale è ben fatto il progetto logico presenta esattamente le stesse proprietà. Il passaggio da progetto logico a fisico, invece, non è così automatico. Dopo la conclusione di una fase di progettazione si passa a quella successiva e non si ritorna più su quelle precedenti.

4.2.3 Modelli concettuali

Abbiamo bisogno di un buon linguaggio per descrivere lo schema concettuale. Abbiamo bisogno di un modello astratto il più vicino possibile all'essere umano: generalmente sono grafici semplici da usare e utili per interagire col cliente (quindi facilmente comprensibili anche da non-tecnici). I modelli esistenti sono formali (definiti in modo non ambiguo, descrivono in modo chiaro le caratteristiche di una base di dati) e integrati (descrivono tutte le caratteristiche della base di dati).



4.3 Modello E-R

Il modello più utilizzato è quello E-R (*Entity-Relationship*): esso è il modello standard per descrivere una base di dati a livello concettuale.

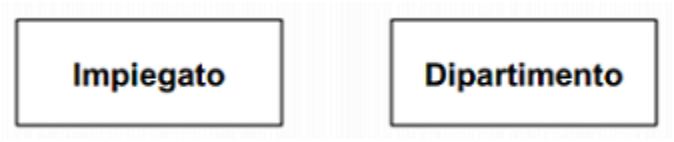
4.3.1 Costrutti del modello E-R

Abbiamo tre costrutti di base: entità, relazioni e attributi. Si aggiungono i seguenti: identificatore e generalizzazione.

4.3.1.1 Entita (*entity*)

Un'entità è una classe di oggetti esistenti. Intendiamo fatti, persone, cose, impiegati, città, fatture, conti correnti... Ad ogni elemento sono associate delle proprietà comuni e con esistenza autonoma.

Occorrenza di entità Un'occorrenza è un elemento appartenente alla classe. A questo livello di progettazione so che un impiegato, per esempio, avrà certe proprietà: non mi interessa sapere i valori esatti ma so che possiede certe proprietà.
Rappresentazione grafica L'entità si rappresenta con un rettangolo avente un nome al suo interno.



Convenzioni sui nomi I nomi devono essere significativi (cioè che alludono in modo chiaro a cosa intendiamo) e possibilmente singolari.

4.3.1.2 Associazioni (*relationship*)

L'associazione è un legame fra due o più entità. Prendiamo per esempio studenti, esami e insegnamenti. Lo studente svolge un esame per un certo corso!

Rappresentazione grafica Le associazioni si rappresentano mediante rombi con all'interno il nome dell'associazione. Il rombo è collegato alle due entità collegate. Si osserva che non sono presenti frecce: non si ha un verso, chi viene prima e chi viene dopo non ha importanza.

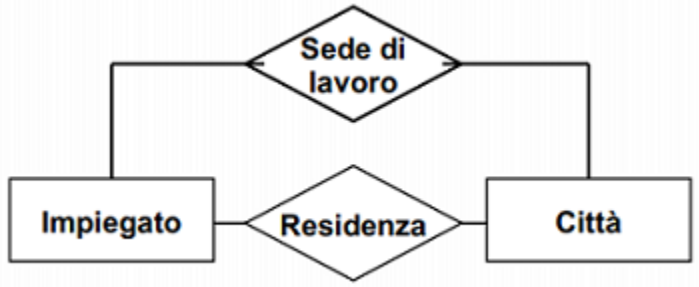
Convenzioni sui nomi Anche qui i nomi devono essere espressivi e singolari. Importante usare sostantivi e non verbi: il verbo attribuisce un verso all'associazione e abbiamo detto che non ci sono frecce.
Occorrenza di associazione Un insieme di tuple, di occorrenze di entità. Se l'associazione è fra t entità avrò t tuple, ciascuna per ogni entità coinvolta. Non posso avere occorrenze ripetute, se ne ha una sola. L'occorrenza è definita come una coppia.

Domanda Questa base di dati consisterà in una rappresentazione del libretto elettronico o in un supporto a statini per la creazione di statistiche? Un uso diverso può comportare una progettazione diversa.

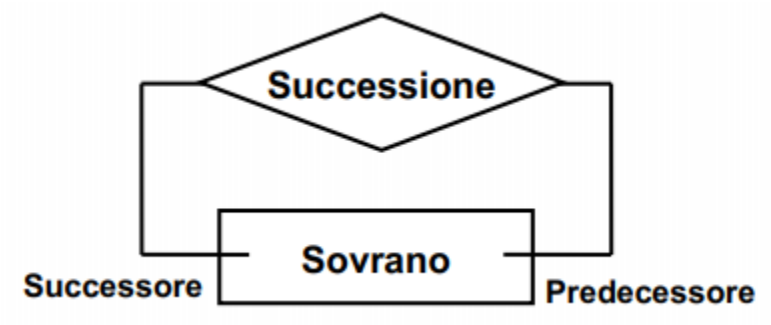
- Se voglio realizzare semplicemente un libretto elettronico mi limito a indicare l'esame passato con la data e la valutazione
- Se voglio fare un supporto a statini allora ho bisogno di ulteriori informazioni: non soltanto l'ultimo esame passato ma anche i tentativi. Se progetto una base di dati come un libretto elettronico non posso trovare queste informazioni! Si deve organizzare la base di dati in modo che queste operazioni siano realizzabili: includeremo a tal proposito una nuova entità *DataEsame*.

Passaggio da una rappresentazione per libretto a rappresentazione per supporto a statini

Relationship diverse sulle stesse entità Possono esistere più relazioni tra le stesse entità. L'impiegato può essere associato, per esempio, a due città: quella di residenza e quella di lavoro. Le due città possono coincidere ma possono essere anche diverse!



Relationship ricorsiva Un'entità può comparire più volte nell'associazione. Lego una persona, per esempio, a un'altra persona: ho l'occorrenza conoscenza che riguarda due persone che si conoscono. Posso stabilire, in alcuni casi, ruoli nella relazione ricorsiva: ho l'entità sovrano e la relazione successione, dove distinguo successore da predecessore.

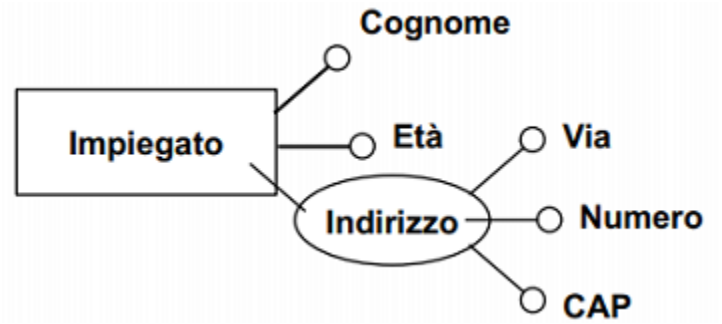


Relationship mista Posso mescolare ricorsione con ricorsione.

4.3.1.3 Attributo

Proprietà elementare di un'entità o di un'associazione. L'attributo ha un certo dominio (o stringa, o intero, o data...) e il valore è associato ad occorrenze di entità o associazione.

Rappresentazione grafica Gli attributi si rappresentano attraverso grafi.



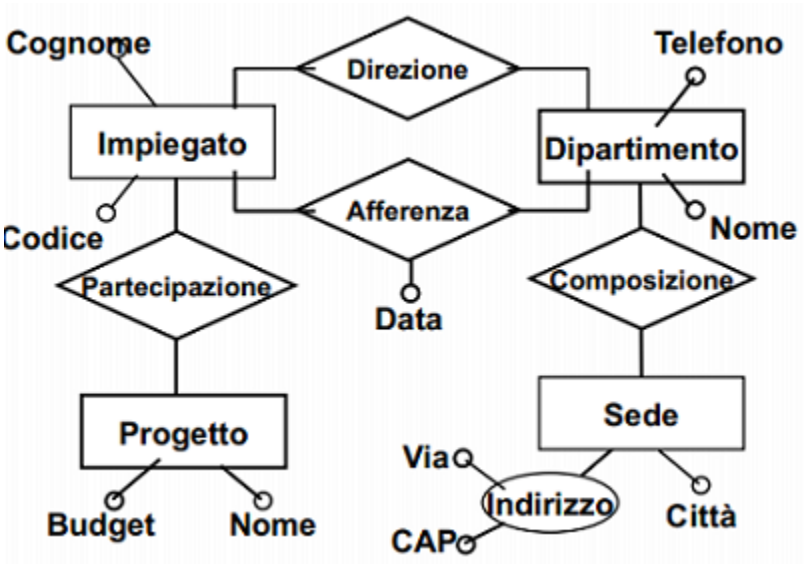
Risulta possibile strutturare gli attributi in modo più dettagliato. Posso raggruppare parte degli attributi in un unico gruppo. La cosa è soprattutto grafica! Il nome dell'attributo composto è racchiuso all'interno di un "cerchio schiacciato".

4.3.2 Esempio concreto

Testo *Si vuole descrivere l'organizzazione di un'azienda*

- *Con sedi diverse*
- *Ogni sede è composta di vari dipartimenti*
- *Gli impiegati dell'azienda afferiscono ai vari dipartimenti e un'impiegato li dirige*
- *Gli impiegati lavorano su progetti*
- *Ogni entità o relationship può avere vari attributi*

Immagine



- Le entità presenti (concrete) sono Impiegato, Dipartimento, Progetto e Sede.
- Individuo gli attributi che descrivono le varie entità.
- Gli impiegati lavorano sui progetti, quindi stabilisco una relazione tra impiegato e progetto.
- L'impiegato afferisce a un dipartimento, cioè può decidere in quale dipartimento stare. Uno degli impiegati, inoltre, dirige un dipartimento
- In ogni sede ci sono più dipartimenti, ogni dipartimento ha una sede in cui è collocato.

Torno dal cliente e gli faccio vedere lo schema Il cliente muove delle osservazioni, dettagli nuovi da aggiungere:

- un dipartimento ha un solo direttore
- un impiegato può afferire ad un solo dipartimento
- il direttore di dipartimento afferisce a quel dipartimento

Lo schema è incompleto e non possiamo aggiungere i dettagli richiesti utilizzando solo i costrutti base.

4.3.3 Cardinalità

La cardinalità mi permette di stabilire, per esempio, quante coppie posso stabilire in un'associazione. Generalmente consiste in una coppia di valori associati a ogni entità che partecipa a una relazione: specifica il numero minimo e massimo di occorrenze della relazione cui ciascuna occorrenza di entità può partecipare.

Rappresentazione grafica La cardinalità si rappresenta etichettando le entità.
Simbologia I simboli adottabili nella cardinalità sono

- numeri, ponibili sia nella cardinalità minima che in quella massima
- N, ponibile solo nella cardinalità massima, che attesta un numero formalmente illimitato di associazioni.

Esempio grafico

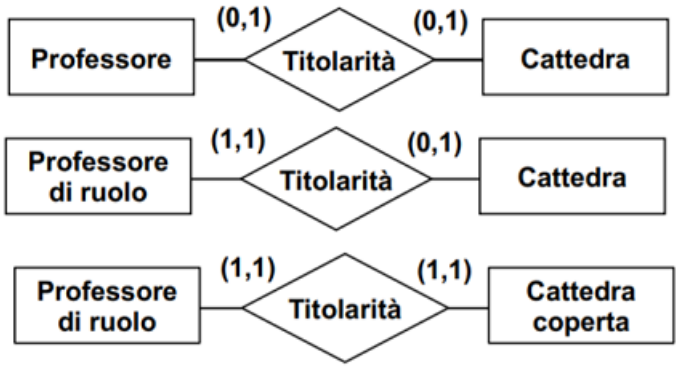
- Ogni impiegato ha un incarico ma non ne può avere più di cinque
- L'incarico può essere assegnato a un massimo di cinquanta persone, ma può essere anche non assegnato.

Esempio grafico 2

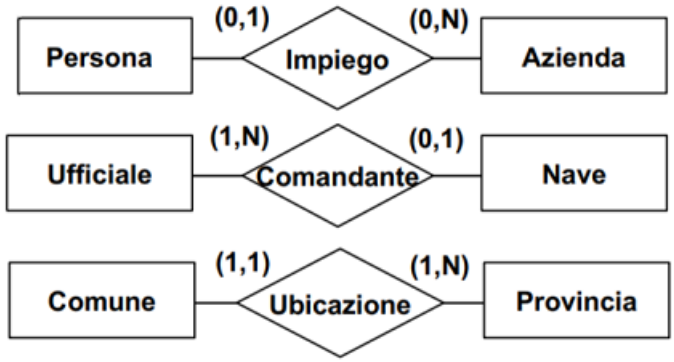
- L'impiegato è associato a una e una sola città di residenza
- La città può essere associata a un numero infinito di impiegati, ma può essere anche priva di relazioni con gli impiegati

Tipi di relationship Attraverso le cardinalità massime possiamo definire i seguenti tipi di relationship:

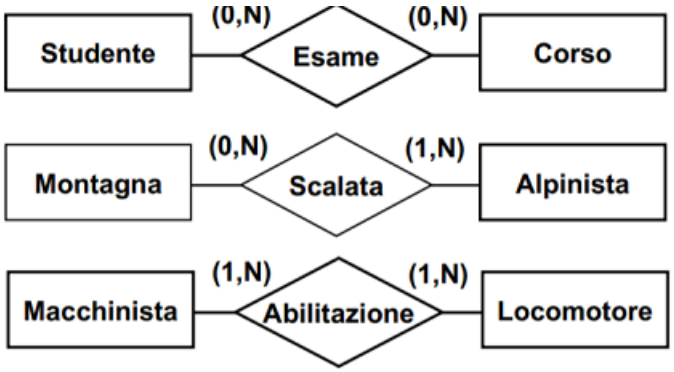
- uno a uno (la cardinalità massima è sempre 1)



- uno a molti (una delle due cardinalità massime è 1, l'altra N)

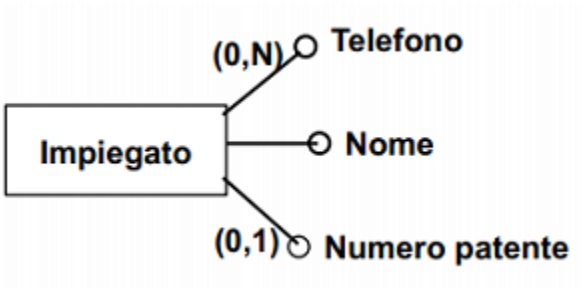


- molti a molti (entrambe le cardinalità massime sono N)



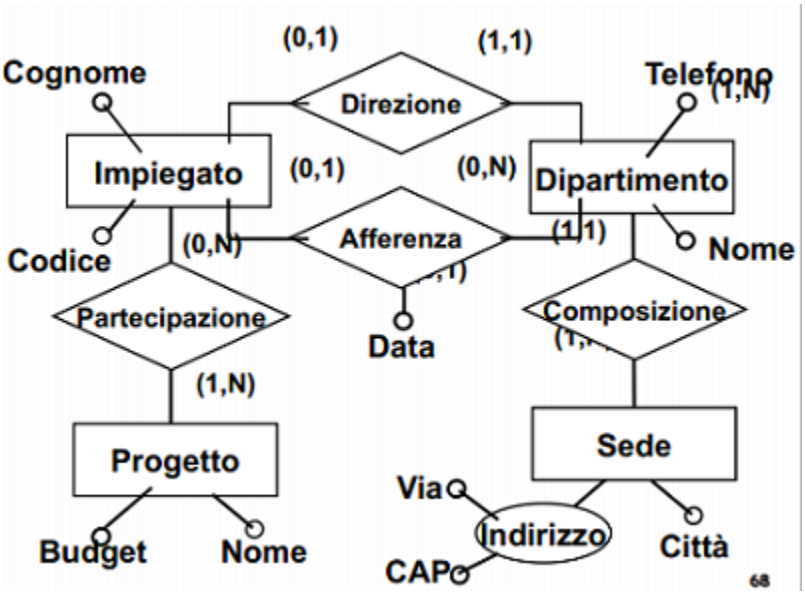
Cardinalità di attributi Posso associare cardinalità anche agli attributi per indicare opzionalità dell'informazione o attributi multivalore.

Esempio cardinalità di attributi



- L'impiegato può essere associato a un numero infinito di telefoni (non solo quello personale, ma anche quello aziendale...) o a nessun telefono
- L'impiegato può essere privo di patente o avere una sola patente.

4.3.4 Riprendiamo lo schema E-R dell'azienda e poniamo le cardinalità



- Il dipartimento ha uno e un solo direttore (c'è sempre, quando aggiungo un nuovo dipartimento conosco già il direttore). L'impiegato può essere direttore ma può anche non esserlo
- Il dipartimento può essere privo di afferenze ma ne può avere un numero illimitato. L'impiegato può non avere afferenza, ma ne ha al massimo una (quindi l'impiegato può decidere di porsi in un certo dipartimento anche dopo l'assunzione). Segue che la data di afferenza può essere assente, e che ce ne sia al massimo una.
- Il dipartimento può avere infiniti numeri di telefono, ma almeno uno.
- L'impiegato può essere assegnato a un numero infinito di progetti, ma potrebbe anche essere privo di incarichi. Un progetto deve avere almeno un impiegato incaricato (ovviamente non si hanno limiti nel numero di persone assegnabili a un progetto).

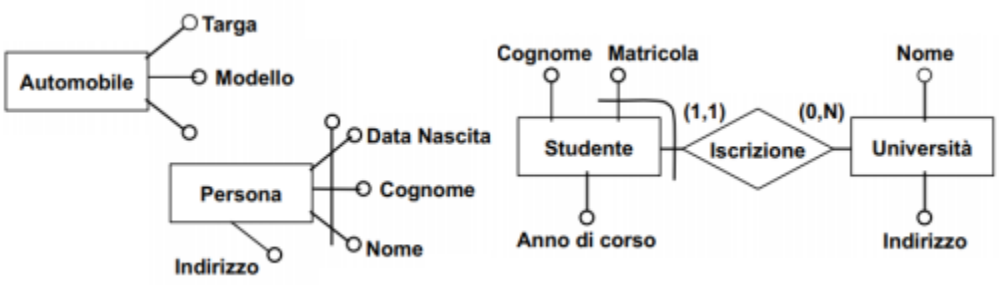
- Il dipartimento sta in una sola sede. La sede ha almeno un dipartimento, ma può ospitare più di un dipartimento

Ritorno nuovamente dal cliente I primi due dettagli sono sistemati, l'ultimo (quello di afferenza del direttore di dipartimento) no! Per il controllo di questa cosa, basato su valori, dobbiamo aggiungere la data di inizio direzione in modo da poter verificare che il direttore abbia assunto l'incarico dopo aver afferito a un certo dipartimento.

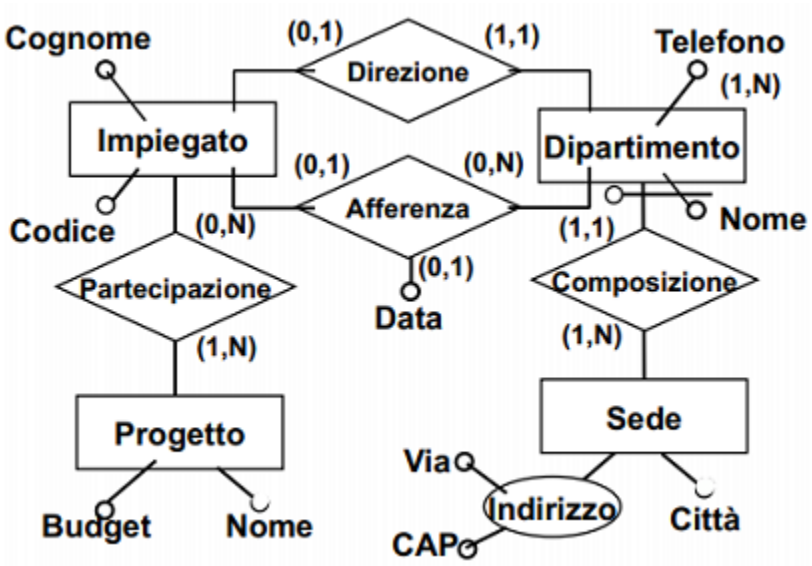
4.3.5 Identificatore di un'entità

Con le cardinalità si impongono vincoli generali sulle occorrenze delle associazioni. La scelta dell'identificatore permette di arricchire ulteriormente lo schema: ne può avere più di uno, ma ne deve avere per forza uno.
L'identificatore può essere, per esempio, la targa di un automobile (se abbiamo un'entità Automobile) o tre attributi (Cognome, Nome e DataNascita) per l'entità Persona.

Identificatori esterni: ho una base di dati con gli studenti di tutta Italia. Uno studente di un certo ateneo è identificato da una matricola: tuttavia potrei avere uno studente, in un altro ateneo, avente la stessa matricola. Quindi devo utilizzare come identificatore non solo la matricola, ma anche il nome dell'università: questo identificatore lo devo prendere da un'altra tabella per evitare che vengano inseriti atenei inesistenti (Per esempio l'*ateneo di Fornacette*). Segue l'uso di un vincolo di integrità referenziale.
L'identificatore esterno può essere utilizzato solo con cardinalità $(1,1)$: se avessi uno studente iscritto a più università in Italia quanto detto non funzionerebbe.



4.3.6 Mettiamo gli identificatori nello schema E-R

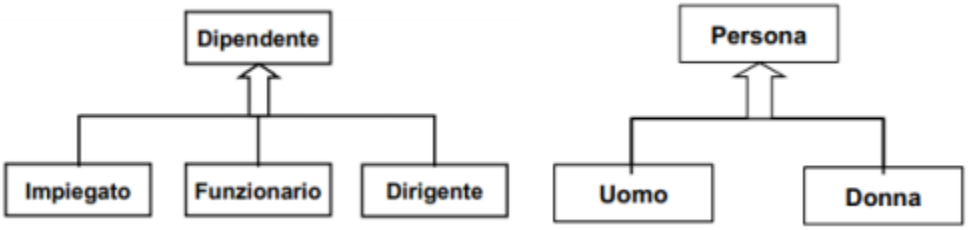


- Ogni impiegato ha un codice univoco identificativo
- Ogni progetto è identificato dal nome
- La sede è identificata dalla città (quindi una sola sede per città)
- Il dipartimento è identificato dal nome e dalla città (identificatore esterno)

Nelle associazioni non ho identificatori poichè l'identificatore è costituito dalla coppia associata.

4.3.7 Generalizzazione

La generalizzazione è un costrutto utile quando si stabilisce il modello concettuale ma che viene perso col passaggio ai modelli successivi. Si ha un concetto gerarchico. Ho l'entità Dipendente, posso stabilire l'esistenza di sue sottoclassi: impiegato, funzionario e dirigente. Sono distinti da attributi specifici che rendono diversa una sottoclasse dall'altra (lo stipendio, la residenza...).



Tutte le proprietà dell'entità genitore vengono ereditate dalle figlie e non rappresentate esplicitamente.
Tipi di generalizzazione La generalizzazione può essere

- **Totale**: freccia piena, se ogni occorrenza dell'entità genitore è occorrenza di almeno una delle entità figlie (Esempio: le occorrenze di persona sono uomini o donne, nient'altro).

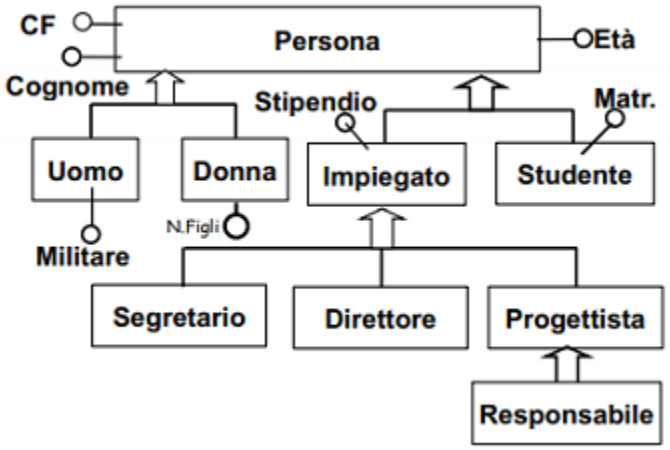
- **Parziale**: freccia vuota, esistono occorrenze dell'entità genitore che non sono occorrenze di entità figlie (Esempio: esistono occorrenze di persona che non sono nè studenti nè lavoratori).

La generalizzazione si dice:

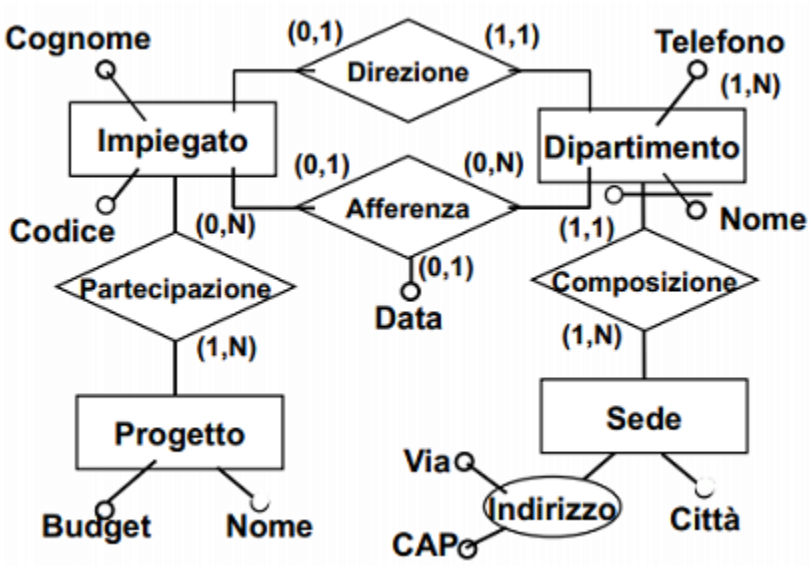
- **Esclusiva**: se ogni occorrenza dell'entità genitore è occorrenza di al più una delle entità figlie (Esempio: gli studenti lavoratori sono persone ma esistono occorrenze di persona che non sono ne studenti nè lavoratori).

- **Sovrapposta**: esistono occorrenze dell'entità genitore che sono occorrenze di più di una delle entità figlie (Esempio: ho uno studente lavoratore, un'occorrenza dell'entità persona è occorrenza sia di studente che di lavoratore).

Gerarchia con una sottoclasse Posso avere una gerarchia con una sola sottoclasse. Segue che avrò una generalizzazione di tipo parziale: è ovvio che se pongo solo una sottoclasse allora esisteranno occorrenze dell'entità genitore che non sono occorrenze dell'entità figlia (altrimenti farei prima a togliere il costruttore chiamando l'entità genitore come l'entità figlia).
Alberi paralleli di gerarchia Sono possibili alberi di gerarchia parallela, cioè fare partire più alberi dall'entità genitore. In questi alberi si distingue generalizzazione di tipo totale da generalizzazione di tipo parziale.



4.3.8 Documentazione associata agli schemi E-R



Lo schema e basta non è mai sufficiente per rappresentare tutti i dettagli di un'applicazione. Ci sono vincoli non esprimibili (Esempio: il direttore che afferisce al dipartimento che dirige). Si pone in forma cartacea ciò che non è visibile dal grafico e anche ciò che lo è.

Dizionari dei dati (entità)

Entità	Descrizione	Attributi	Identificatore
Impiegato	Dipendente dell'azienda	Codice, Cognome	Codice
Progetto	Progetti aziendali	Nome, Budget	Nome
Dipartimento	Struttura aziendale	Nome, Telefono	Nome, Sede
Sede	Sede dell'azienda	Città, Indirizzo	Città

Dizionari dei dati (relationship)

Relazioni	Descrizione	Componenti	Attributi
Direzion	Direzione di un dipartimento	Impiegato, Dipartimento	
Afferenza	Afferenza a un dipartimento	Impiegato, Dipartimento	Data
Partecipazione	Partecipazione a un progetto	Impiegato, Progetto	
Composizione	Composizione dell'azienda	Dipartimento, Sede	

Regole di vincolo

1. Il direttore di un dipartimento deve afferire a tale dipartimento
2. Un impiegato non deve avere uno stipendio maggiore del diretto del dipartimento al quale afferisce
3. Un dipartimento con sede a Roma deve essere diretto da un impiegato con più di dieci anni di anzianità

4. Un impiegato che non afferisce a nessun dipartimento non deve partecipare a nessun progetto

Regole di derivazione

1. Il numero di impiegati di un dipartimento si ottiene contando gli impiegati che afferiscono a tale dipartimento
2. Il budget di un progetto si ottiene moltiplicando per 3 la somma degli stipendi degli impiegati che vi partecipano

I vincoli espressi in precedenza e le regole di derivazione riguardano i valori che vengono immessi nelle entità e associazioni in esecuzione.

4.3.9 Quali costrutti scelgo?

Per la scelta dei costrutti mi baso sulle loro definizioni

- **Entità**: Se ha proprietà significativa e descrive oggetti con esistenza autonoma
- **Attributo**: Se è semplice e non ha proprietà
- **Associazione**: Se correla due o più concetti
- **Generalizzazione**: Se è caso particolare di un altro

4.3.10 Quali caratteristiche deve avere il mio schema?

Uno schema deve essere

- Corretto
- Completo
- Leggibile
- Minimale

4.3.11 Strategie di progetto

- **top-down**: parto da uno schema iniziale, lo raffino e integro mediante primitive che lo trasformano in una serie di schemi intermedi. Collego questi schemi e arrivo allo schema E-R finale. Le primitive di raffinamento sono:
  - Da entità ad associazione tra entità
  - Da entità a generalizzazione
  - Da associazione a insiemi di associazioni
  - Da associazione a entità con associazioni
  - Introduzione di attributi su entità e associazioni
- **bottom-up**: al contrario. Parto da specifiche iniziali, le divido fino a ottenere una specifica componente minima da cui nasce lo schema E-R. Gli schemi prodotti vengono fusi e integrati fino ad ottenere lo schema finale. Le primitive sono:
  - Generazione di entità
  - Generazione di associazione

- Generazione di generalizzazione

Solitamente la strategia adottata è mista:

- inizialmente individuo i concetti principali realizzando uno schema scheletro
- lo decompongo se necessario
- poi lo raffino, espando ed integro.

Definizione dello schema scheletro Schema che contiene i concetti più importanti, i più citati o indicati esplicitamente come cruciali.

5 — Mercoledì 01/04/2020

5.1 Esercitazione su progettazione

5.1.1 Fasi del progetto

Ricapitoliamo le varie fasi introdotte nella lezione precedente

- **Analisi dettagliata delle specifiche fornite dal committente**
  Questa fase è fondamentale per capire a fondo quali informazioni devono essere mantenute all'interno della base di dati. Scoprire in fase avanzata di progettazione che devono essere aggiunte nuove informazioni alla base di dati può essere molto costoso
- **Progettazione concettuale della base di dati**
  Questa fase va dall'individuazione delle entità e delle relazioni, durante l'analisi delle specifiche, alla creazione dello schema concettuale (schema EntityRelationship)
- **Progettazione logica della base di dati**
  Questa fase prende in pasto lo schema concettuale prodotto dalle fasi precedenti, e produce seguendo regole precise che richiedono poche scelte ulteriori lo schema logico (tabelle).
- **Specifica dei vincoli**
  Durante questa fase vengono specificati sia i vincoli di integrità referenziale (che derivano dalla traduzione in schema logico prodotta durante la fase precedente) che altri vincoli (necessari per mantenere la base di dati in uno stato consistente).
- **Realizzazione delle query**
  Durante questa fase vengono create le query SQL che possono essere utilizzate per svolgere sulla base di dati le operazioni predefinite.

5.1.2 Specifiche di progetto

- Vogliamo progettare il sistema informativo di supporto ad un sito per l'acquisto di prodotti on-line.
- Il sito potrà essere utilizzato solamente da utenti registrati, quindi anche gli acquisti possono essere effettuati solamente da utenti che si sono preventivamente registrati sul sito. Ovviamente è possibile che un utente sia registrato sul sito senza aver mai effettuato nessun acquisto.
- I fornitori possono vendere i propri prodotti tramite il sito solo nel caso in cui siano preventivamente registrati.
- I prodotti acquistati mediante il sito possono essere consegnati all'acquirente solamente da corrieri che siano preventivamente registrati sul sito.
- I fornitori offrono i propri prodotti, specificando prezzi e condizioni di vendita e di spedizione.
- Uno stesso prodotto potrà essere venduto da fornitori diversi con prezzi e condizioni di vendita e spedizione diverse.

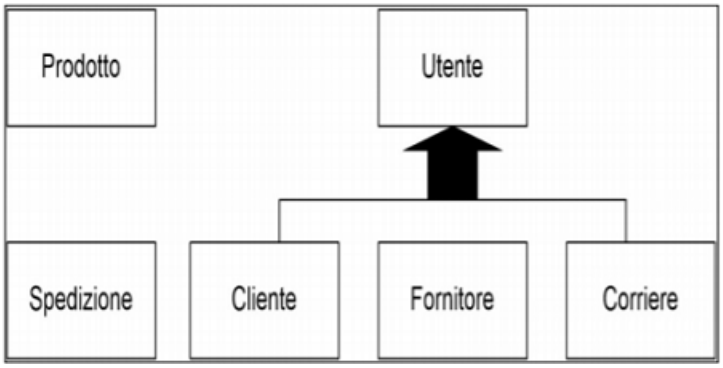
- Il cliente sceglierà il fornitore da cui comprare un prodotto in base al suo criterio di scelta (a causa del prezzo, delle condizioni di vendita, ecc).
- Ciascun corriere metterà a disposizione della clientela vari tipi d consegna.
- Ciascun fornitore registrato sul sito effettuerà accordi commerciali con alcuni dei corrieri registrati sul sito.
- Nel momento in cui un fornitore decida di servirsi di un dato corriere, questi permetterà l'utilizzo di tutti i tipi di consegna messi a disposizione.
- Nel momento in cui un cliente fa un ordine ad un fornitore potrà scegliere per la consegna solamente uno dei corrieri convenzionati con il fornitore. Ovviamente il cliente sceglierà il corriere in base a propri criteri (prezzo, tempi di consegna, garanzie offerte sulla consegna, ecc).
- Il sistema informativo dovrà mantenere anche informazioni sull'abbinamento di prodotti appartenenti a categorie diverse
  - Ad esempio, se un cliente acquista un prodotto di categoria X, il sistema informativo dovrà essere in grado di consigliare i prodotti della categoria Y che sono abbinabili con il prodotto appena acquistato.
- Nel caso in cui il cliente effettui un ordine tramite il sito potrà specificare il proprio grado di soddisfazione sull'operato del fornitore, inserendo nella base di dati un voto compreso tra 0 e 10.
- Analogamente il cliente potrà specificare il proprio grado di soddisfazione sull'operato del corriere.
- Il voto che un cliente dà ad un fornitore o ad un corriere è legato ad un ordine e quindi ad una data.
- Sulla base di dati progettata dovranno essere effettuate delle query in grado di recuperare almeno le seguenti informazioni:
  - Elenco di tutti i prodotti di una data categoria X che sono consigliati in seguito all'acquisto di un dato prodotto di categoria Y.
  - Elenco dei fornitori che vendono il maggior numero di prodotti diversi di una data categoria X.
  - Elenco dei fornitori che vendono un dato prodotto x a prezzo più basso.
  - Elenco dei fornitori con voto medio più alto tra quelli che hanno un dato prodotto x presente in magazzino.
- Inoltre sulla base di dati dovranno essere effettuate le seguenti operazioni:
  - Inserimento di un nuovo ordine (con grado di soddisfazione del cliente nei confronti dell'operato del fornitore e del corriere nullo).
  - Inserimento del grado di soddisfazione del cliente nei confronti dell'operato del fornitore e/o del corriere relativamente ad un ordine già effettuato.
  - Inserimento di un nuovo prodotto nel magazzino di un fornitore (supponendo che tale prodotto fosse già presente all'interno della base di dati).

5.1.3 Individuazione entità

- Iniziamo la progettazione individuando le entità della base di dati:
  - Gli Utenti sono registrati e costituiscono un'entità
  - Un Prodotto è un'entità, la Categoria a cui appartiene può essere considerata come suo attributo

  - La Spedizione (il tipo)

- Possiamo specializzare Utente con Cliente, Fornitore e Corriere
- Otteniamo, quindi, la seguente situazione dove gli attributi in comune (Telefono, Indirizzo, e-mail, ecc) di Cliente, Fornitore e Corriere saranno attributi di Utente.



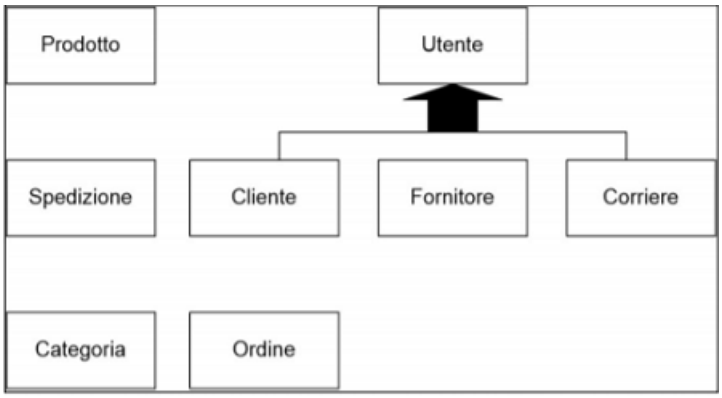
- La base di dati deve tenere memoria degli ordini effettuati in modo che un voto possa essere assegnato sia ai fornitori che ai corrieri.
- Ciascun ordine consisterà delle seguenti informazioni:
  - Cliente che ha effettuato l'ordine
  - Fornitore che ha ricevuto l'ordine
  - Corriere utilizzato per la consegna dell'ordine
  - Tipo di spedizione utilizzato per recapitare l'ordine
  - Prodotti acquistati tramite l'ordine
  - Voto al corriere
  - Voto al fornitore
- Poiché un ordine non ha un'esistenza indipendente dalle altre entità (ad esempio un ordine non può esistere se non c'è un cliente che lo genera), possiamo classificare Ordine come una relazione.
- Ordine stabilisce un'associazione tra 5 entità diverse:
  - Cliente
  - Fornitore
  - Corriere
  - Spedizione
  - Prodotti
- Ed ha alcuni attributi propri: coto al corriere e voto al fornitore
- Nel ragionamento fin qui fatto abbiamo stabilito due punti che possono risultare difficili da gestire
  1. una relazione connessa a 5 entità diverse pone problemi soprattutto per quanto riguarda l'individuazione delle cardinalità
     - Possiamo pensare di raffinare (approccio topdown) la relazione Ordine in un'entità Ordine correlata alle 5 entità Cliente, Fornitore,.. da 5 relazioni diverse.
  2. I prodotti hanno un attributo Categoria. Questa soluzione potrebbe dare origine a inconsistenze in fase di inserimento. Ad esempio, un fornitore potrebbe inserire

  - Prodotto = “x”, Categoria = “RAM”
  - Prodotto = “y”, Categoria = “ram”

  Come conseguenza nella base di dati i due prodotti risulterebbero appartenenti a categorie diverse

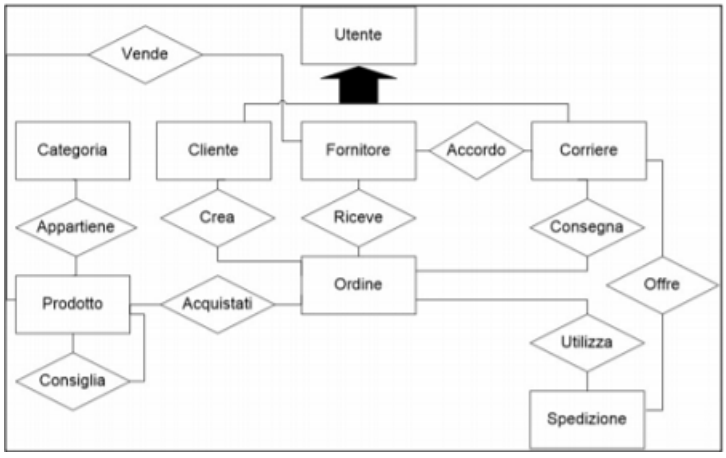
  - Per evitare problemi di questo tipo possiamo effettuare ancora un raffinamento e trasformare Categoria in un’entità collegata a Prodotto da una relazione di appartenenza.

- A questo punto lo schema delle entità risulterà il seguente: abbiamo una generalizzazione e alcune associazione già individuate



5.1.4 Individuazione relazioni

- Occorre mettere in relazione l’entità Fornitore e l’entità Prodotto: introduciamo la relazione Vende
- Occorre mettere in relazione l’entità Fornitore e l’entità Corriere: introduciamo la relazione Accordo
- Occorre mettere in relazione l’entità Corriere e l’entità Spedizione: introduciamo la relazione Offre
- Occorre mettere in relazione l’entità Prodotto con se stessa per poter memorizzare l’abbinamento tra prodotti di categoria diversa: introduciamo la relazione Consiglia

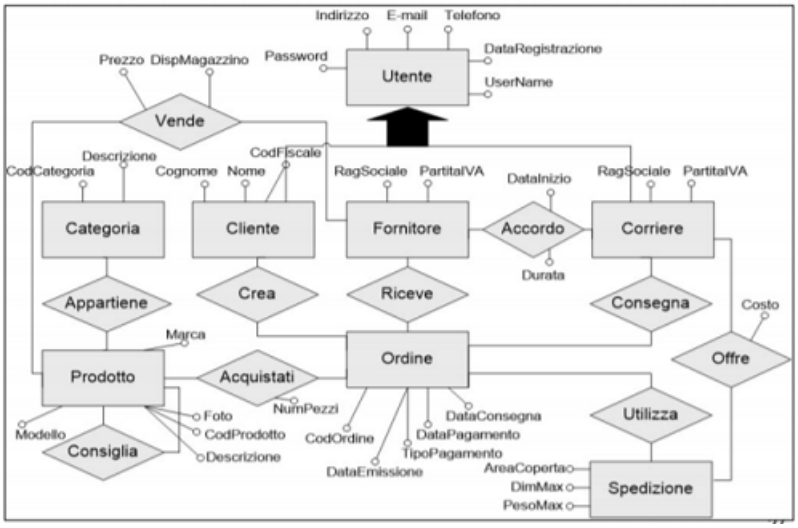


5.1.5 Individuazione attributi

- Vediamo quali attributi associare alle varie entità:
  - Utente: Indirizzo, Telefono, E-mail, DataRegistrazione, UserName, Password

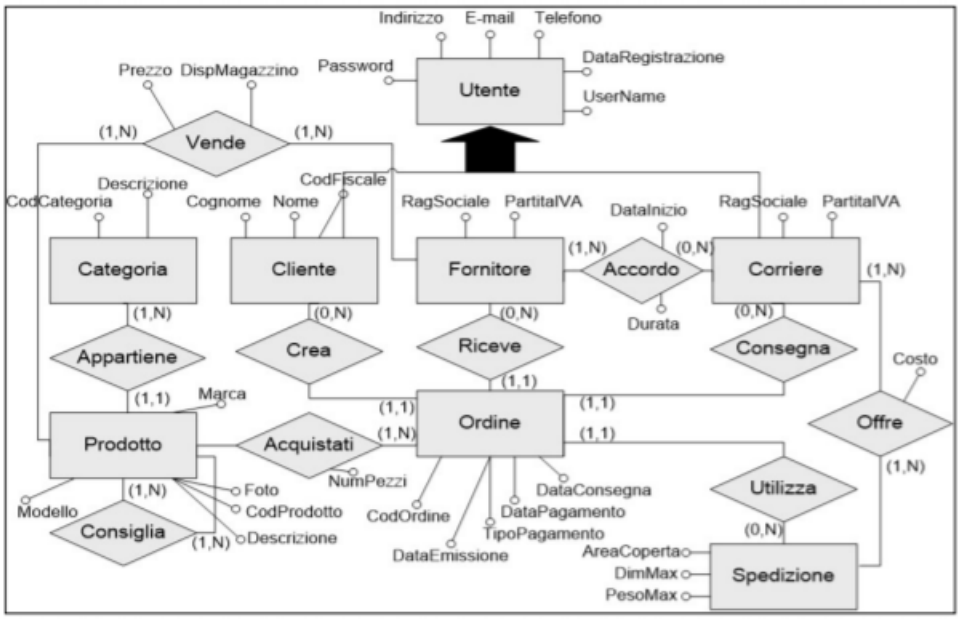
  - Cliente: Nome, Cognome, CodFiscale
  - Fornitore: RagSociale, PartitaIVA
  - Corriere: RagioneSociale, PartitaIVA
  - Categoria: CodCategoria, Descrizione
  - Prodotto: CodProdotto, Marca, Modello, Descrizione, Foto
  - Ordine: CodOrdine, DataEmissione, DataPagamento, TipoPagamento, DataConsegna, VotoFornitore, VotoCorriere
  - Spedizione: AreaCoperta, DimMax, PesoMax

- Non abbiamo associato il prezzo all'entità Prodotto perché un medesimo prodotto potrebbe essere venduto da fornitori diversi a prezzi diversi.
- Analogamente non abbiamo associato il prezzo all'entità Spedizione perché supponiamo che uno stesso tipo si spedizione possa essere offerto da vari corrieri a prezzi diversi.
- Vediamo quali attributi potremmo associare alle varie relazioni:
  - Vende: Prezzo, DispMagazzino
  - Appartiene: nessun attributo
  - Crea: nessun attributo
  - Riceve: nessun attributo
  - Acquistati: NumPezzi
  - Consiglia: nessun attributo
  - Accordo: DataInizio, Durata
  - Consegna: nessun attributo
  - Utilizza: nessun attributo
  - Offre: Costo
- Alla relazione Accordo abbiamo associato l'attributo DataInizio e Durata, ma supponiamo di non dover fare nessun controllo perché manterremo nella base di dati solamente gli accordi attualmente in vigore.
- Nella relazione Acquistati manteniamo memoria del numero di pezzi di ciascun prodotto richiesti in ciascun ordine



5.1.6 Individuazione cardinalità

- Entità Ordine - Relazione Riceve: (1,1) in quanto un ordine viene ricevuto da un solo fornitore.
- Entità Ordine - Relazione Consegna: (1,1) in quanto un ordine viene recapitato da un solo corriere.
- Entità Ordine - Relazione Crea: (1,1) in quanto un ordine viene eseguito da un solo cliente.
- Entità Ordine - Relazione Utilizza: (1,1) in quanto per la consegna di un ordine si utilizza un solo tipo di spedizione.
- Entità Ordine - Relazione Acquistati: (1,N) in quanto un ordine può contenere più prodotti diversi.
- Entità Fornitore - Relazione Riceve: (0,N) in quanto un fornitore può aver ricevuto più ordini o nessun ordine (se ad esempio il fornitore si è appena iscritto al sito).
- Entità Fornitore - Relazione Accordo: (1,N) si suppone che, contemporaneamente all'iscrizione al sito, ogni fornitore si accordi con almeno un corriere.
- Entità Fornitore - Relazione Vende: (1,N) un fornitore può vendere più di un prodotto. Si suppone che, nel momento in cui si iscrive al sito, ogni fornitore sia in grado di vendere almeno un prodotto.
- Entità Corriere - Relazione Accordo: (0,N) un corriere può essere accordato con più fornitori o con nessun fornitore (ad esempio se si è appena iscritto al sito).
- Entità Corriere - Relazione Consegna: (0,N) un corriere può aver consegnato più ordini o nessun ordine (ad esempio se si è appena iscritto al sito).
- Entità Spedizione - Relazione Utilizza: (0,N) un tipo di spedizione può essere utilizzato per consegnare più ordini. È possibile però che un tipo di spedizione non sia mai stato utilizzato (ad esempio se è stato appena introdotto nel sito o se è particolarmente svantaggioso).
- Entità Spedizione - Relazione Offre: (1,N) un tipo di spedizione può essere offerto da più corrieri. Ciascun tipo di spedizione, però, deve essere offerto da almeno un corriere, altrimenti non avrebbe motivo di essere inserito nella base di dati del sito
- Entità Cliente - Relazione Ordine: (0,N) ciascun cliente può aver effettuato più ordini. È possibile che un cliente non abbia effettuato alcun ordine.
- Entità Prodotto - Relazione Vende: (1,N) un prodotto può essere venduto da più fornitori, e deve essere venduto da almeno un fornitore (altrimenti non avrebbe senso la presenza del prodotto nella base di dati del sito).
- Entità Corriere - Relazione Offre: (1,N) un corriere può offrire tipi di spedizione diversi. Si suppone che un corriere offra almeno un tipo di spedizione.
- Entità Prodotto - Relazione Consiglia: (1,N) ciascun prodotto ha almeno un altro prodotto consigliato.
- Entità Categoria - Relazione Appartiene: (1,N) la base di dati deve contenere almeno un prodotto per ciascuna categoria, infatti l'esistenza di una categoria vuota non avrebbe senso.
- Entità Prodotto - Relazione Appartiene: (1,1) ciascun prodotto appartiene ad una ed una sola categoria.



5.1.7 Introduzione di nuovi attributi

- Se analizziamo le operazioni che devono essere compiute sulla base di dati ci rendiamo conto che potrebbe convenire introdurre dei nuovi attributi al fine di facilitarle. Questi attributi possono anche anche rappresentare informazione già presente nella base di dati.
- Consideriamo, ad esempio: elenco dei fornitori con voto medio più alto tra quelli che hanno un dato prodotto X presente in magazzino.
- In questo caso per recuperare il voto medio di un fornitore dobbiamo andare a leggere tutti gli ordini ricevuti da tale fornitore.
- Per facilitare questa operazione potremmo introdurre i seguenti attributi sull'entità Fornitore (perché non il voto medio direttamente?): TotaleVoti, NumeroVoti.
- In questo modo, ogni volta che introduciamo un ordine dovremo anche aggiornare questi attributi per il fornitore che ha ricevuto l'ordine.
- Quindi le operazioni coinvolte dall'introduzione di questi nuovi attributi sono le seguenti:
  - Elenco dei fornitori con voto medio più alto tra quelli che hanno un dato prodotto x presente in magazzino.
  - Inserimento del grado di soddisfazione del cliente nei confronti dell'operato del fornitore e/o del corriere in un ordine di cui si conosce il codice.
- Per capire se l'introduzione dei nuovi attributi è vantaggiosa in termini di tempi di esecuzione, dobbiamo valutari i diversi costi per queste due operazioni.

5.1.8 Riprendiamo le specifiche di progetto

Al fine di prendere decisioni sulla struttura della base di dati, lo studente supponga che la base di dati contenga:

- 600 prodotti diversi
- 50 fornitori registrati

- 1500 clienti registrati
- 15 corrieri registrati
- 3 abbinamenti (in media) tra un prodotto x (di categoria X) e i prodotti della categoria Y.
- 5 ordini finora effettuati per ciascun prodotto.

5.1.9 Tavola dei volumi

- Prima è necessario valutare il numero di istanze di una data entità o relazione.
- Per fare questo dobbiamo basarci sulle indicazioni presenti all'interno delle specifiche.
- Nel caso in cui queste non siano sufficienti, dovremo fare delle ulteriori ipotesi che faranno parte della descrizione finale della base di dati.

- Cliente: 1500 istanze (da specifiche)
- Fornitore: 50 istanze (da specifiche)
- Corriere: 15 istanze (da specifiche)
- Utente: 1565 istanze (perché le persone che sono iscritte sia come clienti che come fornitori e/o corrieri hanno più di un account. Quindi gli utenti possono essere al più: 1565=1500+50+15)
- Prodotto: 600 istanze (da specifiche)
- Categoria: 30 istanze (supponiamo che la base di dati contenga 30 categorie diverse di prodotti)
- Spedizione: 6 istanze (supponiamo che la base di dati contenga 6 tipi di spedizione diversi)
- Ordine: 3000 istanze (perché da specifiche vengono effettuati in media 5 ordini per ciascun prodotto: 3000=600*5)
- Vende: 1800 istanze (supponiamo che ciascun fornitore venda in media 36 prodotti diversi: 1800=50*36)
- Accordo: 150 istanze (supponiamo che ciascun fornitore si sia accordato in media con 3 corrieri diversi: 150=50*3)
- Appartiene: 600 istanze (perchè ciascun prodotto appartiene ad una, ed una sola, categoria)
- Crea: 3000 istanze (perché ciascun ordine è eseguito da un solo cliente)
- Riceve: 3000 istanze (perché ciascun ordine è ricevuto da un solo fornitore)
- Consegna: 3000 istanze (perché ciascun ordine è recapitato da un solo corriere)
- Utilizza: 3000 istanze (perché ciascun ordine è recapitato utilizzando un solo tipo di consegna)
- Offre: 60 istanze (perché ciascun corriere offre in media 4 tipi di spedizione diversi: 60=15*4)
- Acquistati: 6000 istanze (perché ciascun ordine comprende in media 2 prodotti diversi: 6000=3000*2)
- Consiglia: 1800 istanze (perché da specifiche, ciascun prodotto ha in media 3 abbinamenti)

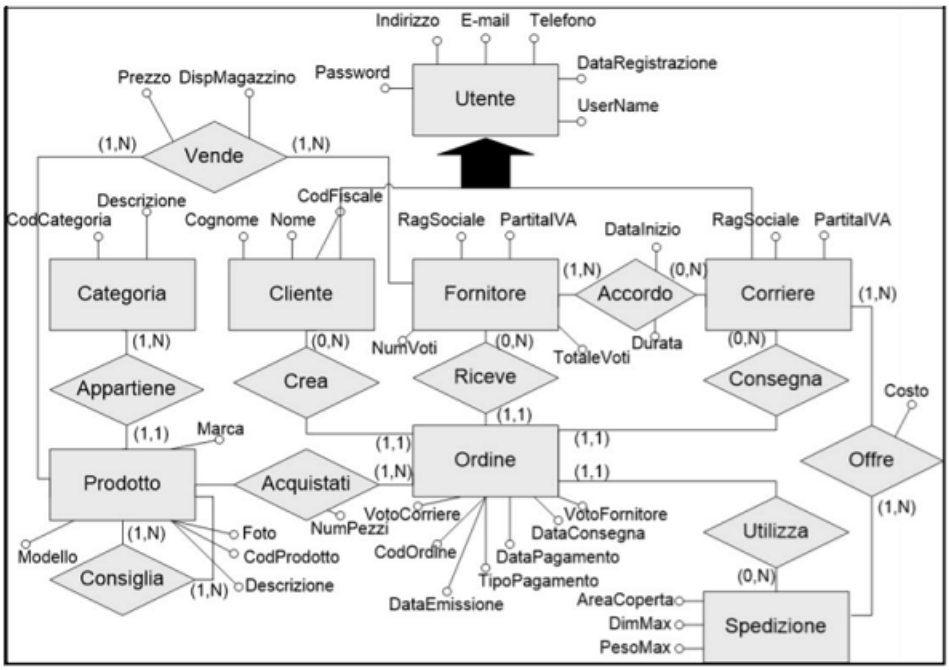
5.1.10 Operazioni elementari

- Date le operazioni coinvolte
  - Operazione 1: Elenco dei fornitori con voto medio più alto tra quelli che hanno un dato prodotto X presente in magazzino.
  - Operazione 2: Inserimento del grado di soddisfazione del cliente nei confronti dell'operato del fornitore e/o del corriere per un ordine di cui si conosce il codice.
- Dobbiamo individuare le operazioni elementari nei seguenti casi:
  - svolgimento dell'operazione 1 senza nuovi attributi
  - svolgimento dell'operazione 1 con nuovi attributi
  - svolgimento dell'operazione 2 senza nuovi attributi
  - svolgimento dell'operazione 2 con nuovi attributi
- Esecuzione Operazione 1 senza attributi aggiuntivi:
  - 1 lettura in Prodotto per recuperare il codice del prodotto
  - 3 letture in Vende per recuperare il codice dei fornitori che vendono quel prodotto (un prodotto è venduto in media da 3 fornitori: 3=1800/600).
  - 180 letture in Riceve per recuperare il codice degli ordini ricevuti dai fornitori che hanno quel prodotto in magazzino (ciascun fornitore riceve in media 60 ordini: 60=3000/50).
  - 180 letture in Ordine per poter leggere il voto assegnato nei vari ordini ai fornitori che hanno quel prodotto in magazzino.
- Esecuzione Operazione 1 con nuovi attributi:
  - 1 lettura in Prodotto per recuperare il codice del prodotto
  - 3 letture in Vende per recuperare il codice dei fornitori che vendono quel prodotto (un prodotto è venduto in media da 3 fornitori: 3=1800/600).
  - 3 letture in Fornitore per recuperare il TotaleVoti e il NumeroVoti di ciascun fornitore che ha quel prodotto in magazzino.
- Esecuzione Operazione 2 senza attributi aggiuntivi:
  - 1 scrittura in Ordine per inserire un voto non nullo
- Esecuzione Operazione 2 con attributi aggiuntivi:
  - 1 scrittura in Ordine
  - 1 lettura in Riceve per recuperare il fornitore che ha ricevuto l'ordine
  - 1 lettura in Fornitore per recuperare il TotaleVoti e NumeroVoti
  - 1 scrittura in Fornitore per scrivere il nuovo valore di TotaleVoti e NumeroVoti

5.1.10.1 Conclusioni

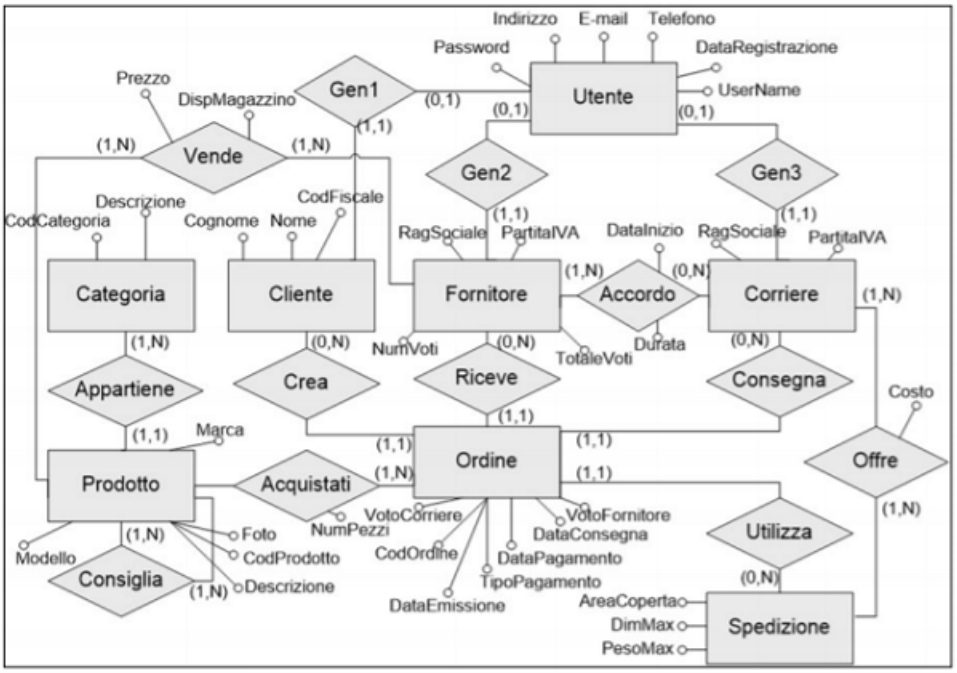
- Operazioni necessarie per svolgere l'operazione 1:
  363 letture → 363 operazioni elementari
- Operazioni necessarie per svolgere l'operazione 1 con attributi aggiuntivi:
  6 letture → 6 operazioni elementari
- Operazioni necessarie per svolgere l'operazione 2:
  1 scrittura → 2 operazioni elementari

- Operazioni necessarie per svolgere l'operazione 2 con attributi aggiuntivi
  - 2 scritture → 4 operazioni elementari
  - 2 letture → 2 operazioni elementari
- Supponendo che l'operazione 2 venga compiuta 10 volte più frequentemente rispetto all'operazione 1, avremo
  - con i nuovi attributi:
    - 6 Op. Elementari + 6*10 Op. Elementari
    - totale= 66 Op. Elementari
  - senza nuovi attributi:
    - 363 Op. Elementari + 2*10 Op. Elementari
    - totale = 383 Op. Elementari
- Al fine di ridurre le operazioni elementari introduciamo i nuovi attributi all'interno dello schema ER.

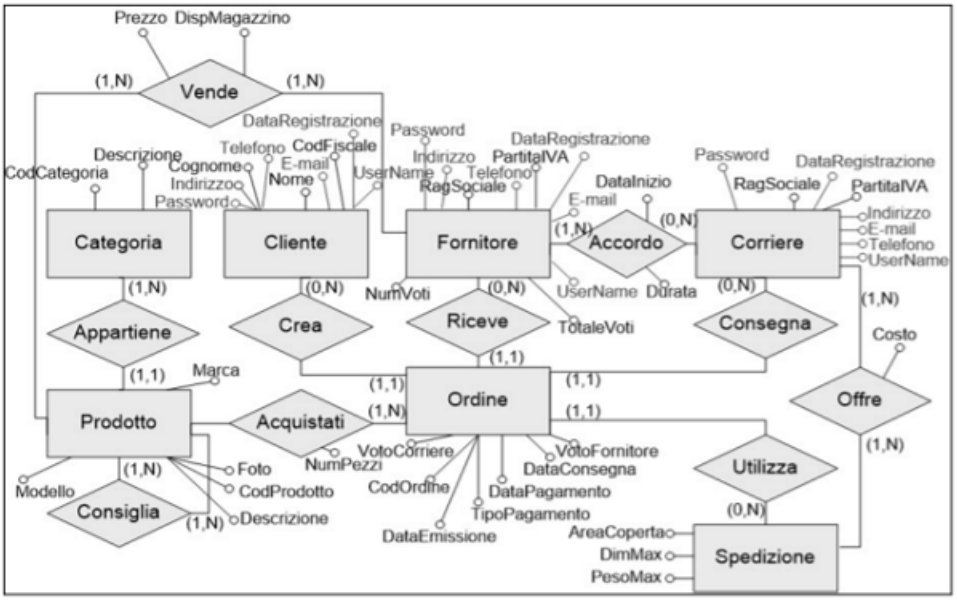


5.1.11 Traduzione generalizzazioni

- Vediamo adesso come possiamo tradurre la generalizzazione presente nello schema ER.
- Poiché si tratta di una generalizzazione totale, abbiamo tre possibilità:
  - Lasciare padri e figli separati e collegarli utilizzando alcune relazioni.
  - Accorpare l'entità padre sulle entità figlie
  - Accorpare le entità figlie sull'entità padre.
- Analizziamo gli schemi ER che otteniamo in ciascuno di questi tre casi in modo da decidere quale di questi utilizzare.

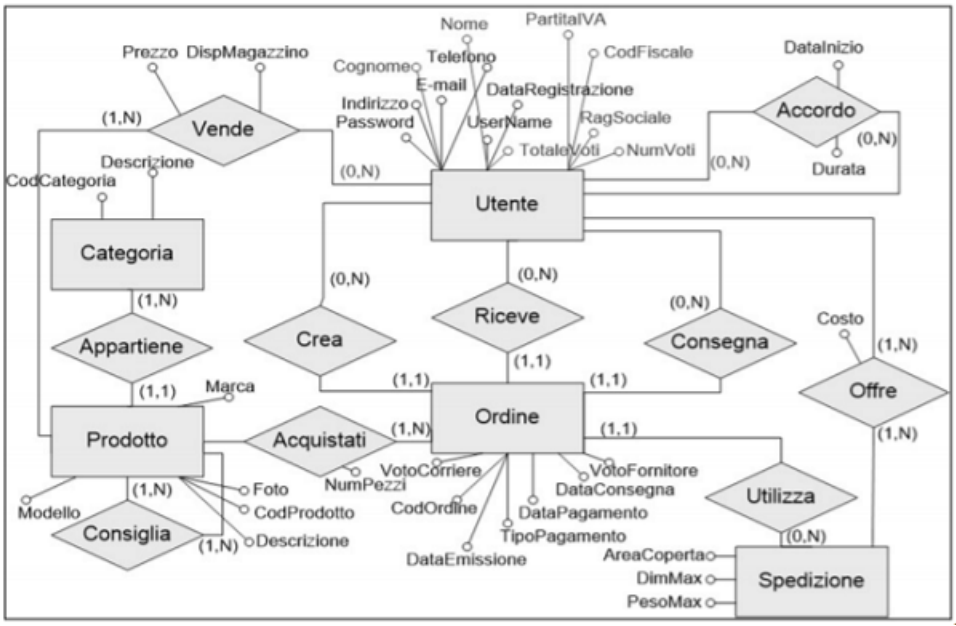


- In questo modo abbiamo introdotto nello schema ER tre nuove relazioni.
- Non abbiamo introdotto nessun valore NULL all'interno della base di dati.
- Abbiamo mantenuto un numero non troppo elevato di attributi per ciascuna entità
  - quanto più il numero di attributi di un'entità è elevato, tanto più la gestione della relativa tabella diventa pesante. È quindi buona norma tentare di limitare il numero di attributi di ciascuna entità.



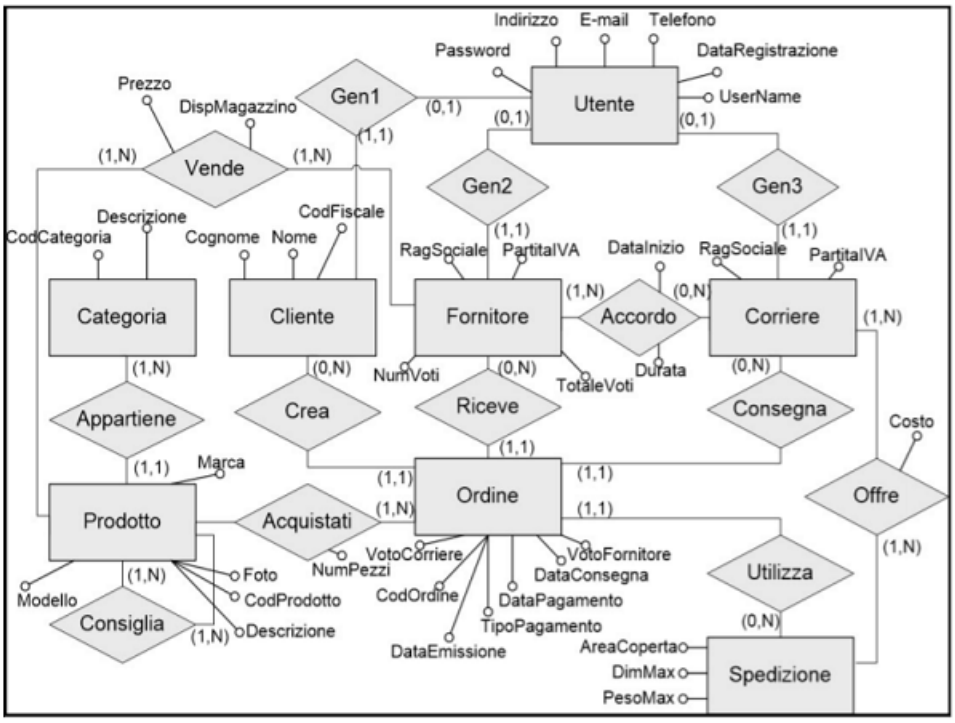
- In questo modo non abbiamo introdotto nello schema nessuna nuova relazione.
- Non abbiamo introdotto nessun valore NULL all'interno della base di dati.

- Abbiamo però aggiunto alle entità Cliente, Fornitore e Corriere molti nuovi attributi (evidenziati in rosso nello schema ER) ottenendo tabelle più pesanti da gestire.
- Inoltre, quando vogliamo controllare la correttezza di una coppia (username, password) dovremo controllare le coppie (username, password) presenti in tre diverse tabelle (a meno che non si sappia in anticipo se tale coppia appartiene ad un cliente, ad un fornitore o ad un corriere).



- In questo modo non abbiamo introdotto nello schema ER nessuna nuova relazione.
- Abbiamo però introdotto alcuni valori NULL all'interno dell'entità Utenti.
- Il numero di attributi dell'entità Utenti è cresciuto molto. La tabella Utenti è molto pesante da gestire.
- Notare che anche se un fornitore vende necessariamente almeno un prodotto (cardinalità (1,N) tra la relazione Vende e l'entità Fornitori), non è detto che un utente (che può essere anche un cliente) venda necessariamente un prodotto (quindi cardinalità (0,N) tra la relazione Vende e l'entità Utenti).
- Lo stesso vale per la relazione Accordo.

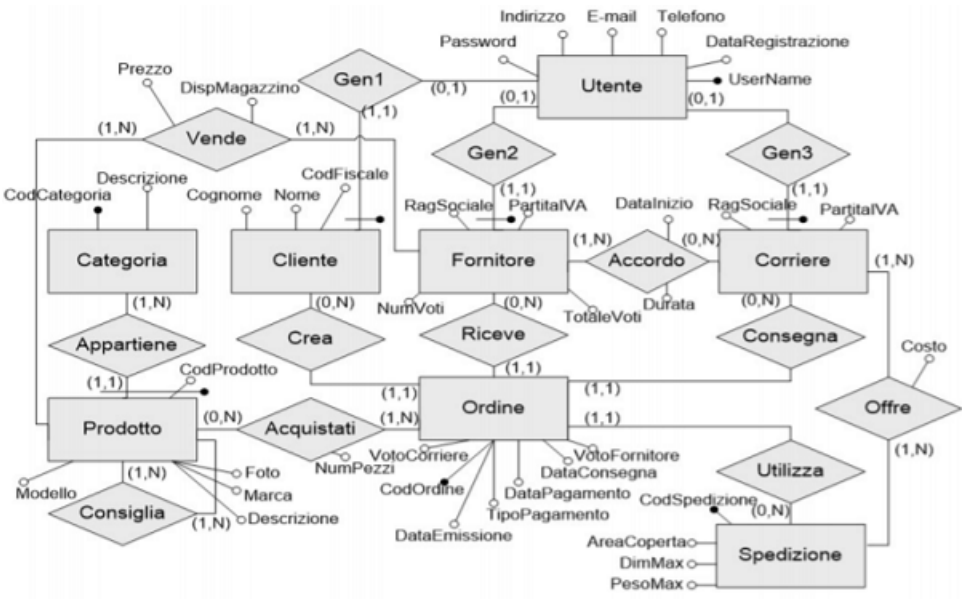
- Viste le tre possibili soluzioni, possiamo concludere che in questo caso conviene tradurre la generalizzazione lasciando l'entità padre e le entità figlie separate e collegate tramite relazioni.
- In questo modo non introduciamo nessun valore NULL e il numero di attributi di ciascuna entità resta sufficientemente limitato.



5.1.12 Specifica chiavi

Adesso dobbiamo specificare le chiavi delle varie entità:

- Categoria: utilizziamo CodCategoria come chiave.
- Prodotto: utilizziamo come chiave la coppia CodProdotto (attributo di prodotto) e CodCategoria (chiave dell'entità Categoria).
- Ordine: utilizziamo CodOrdine come chiave.
- Utente: utilizziamo UserName come chiave, in quanto ogni utente è caratterizzato dal proprio username.
- Fornitore: potremmo utilizzare PartitaIVA come chiave, in quanto ogni fornitore è caratterizzato dalla propria partita IVA. In questo caso, peró, utilizzeremo come chiave lo UserName dell'utente associato.
- Corriere: potremmo utilizzare PartitaIVA come chiave, in quanto ogni corriere è caratterizzato dalla propria partita IVA. In questo caso, peró, utilizzeremo come chiave lo UserName dell'utente associato.
- Cliente: potremmo utilizzare CodFiscale come chiave, in quanto ogni corriere è caratterizzato dal proprio codice fiscale. In questo caso, peró, utilizzeremo come chiave lo UserName dell'utente associato.
- Spedizione: per identificare la spedizione dovremmo utilizzare gli attributi AreaCoperta, DimMax e PesoMax. Poiché utilizzare tre attributi come chiave può risultare scomodo, possiamo aggiungere un attributo CodSpedizione ed utilizzarlo come chiave.



5.1.13 Traduzione in tabelle

- Traduciamo lo schema ER ottenuto in tabelle
- Ciascuna entità viene tradotta in una tabella a se stante
- Vediamo come tradurre le varie relazioni:
  - Gen1: la traduciamo accorpando la tabella sull'entitá Cliente (cardinalitá (1,1) tra la relazione Gen1 e l'entitá Clente)
  - Gen2: la traduciamo accorpando la tabella sull'entitá Fornitore (cardinalitá (1,1) tra la relazione Gen2 e l'entitá Fornitore)
  - Gen3: la traduciamo accorpando la tabella sull'entitá Corriere (cardinalitá (1,1) tra la relazione Gen3 e l'entitá Corriere)
  - Vende: la traduciamo in una tabella a sé stante (relazione da N a N)
  - Accordo: la traduciamo in una tabella a sé stante (cardinalitá (1,N) tra la relazione Accordo e l'entitá Fornitore e cardinalitá (0,N) tra la relazione Accordo e l'entitá Corriere)
  - Appartiene: la traduciamo accorpando la tabella sull'entitá Prodotto (cardinalitá (1,1) tra la relazione Appartiene e l'entitá Prodotto)
  - Crea: la traduciamo accorpando la tabella sull'entitá Ordine (cardinalitá (1,1) tra la relazione Crea e l'entitá Ordine)
  - Riceve: la traduciamo accorpando la tabella sull'entitá Ordine (cardinalitá (1,1) tra la relazione Riceve e l'entitá Ordine)
  - Consegna: la traduciamo accorpando la tabella sull'entitá Ordine (cardinalitá (1,1) tra la relazione Consegna e l'entitá Ordine)
  - Utilizza: la traduciamo accorpando la tabella sull'entitá Ordine (cardinalitá (1,1) tra la relazione Utilizza e l'entitá Ordine)
  - Acquistati: la traduciamo in una tabella a sé stante (cardinalitá (1,N) tra la relazione Acquistati e l'entitá Ordine e cardinalitá (0,N) tra la relazione Acquistati e l'entitá Prodotto)
  - Offre: la traduciamo in una tabella a sé stante (relazione da N a N)
  - Consiglia: la traduciamo in una tabella a sé stante (relazione da N a N)
- Quindi otteniamo le seguenti tabelle:

- **Utente** (UserName, DataRegistrazione, Telefono, E-mail, Indirizzo, Password)
- **Cliente** (UserName, Nome, Cognome, CodFiscale)
- **Fornitore** (UserName, RagSiociale, PartitaIVA, NumVoti, TotaleVoti)
- **Corriere** (UserName, RagSiociale, PartitaIVA)
- **Spedizione** (CodSpedizione, AreaCoperta, DimMax, PesoMax)
- **Ordine** (CodOrdine, DataEmissione, TipoPagamento, DataPagamento, DataConsegna, VotoFornitore, VotoCorriere, CodCliente, CodFornitore, CodCorriere, CodSpedizione)

- **Categoria** (CodCategoria, Descrizione)
- **Prodotto** (CodProdotto, CodCategoria, Foto, Marca, Descrizione, Modello
- **Vende** (CodFornitore, CodProdotto, CodCategoria, Prezzo, DispMagazzino)
- **Consiglia** (CodCategoria1, CodProdotto1, CodCategoria2, CodProdotto2)
- **Acquistati** (CodProdotto, CodCategoria, CodOrdine, NumPezzi)
- **Offre** (CodCorriere, CodSpedizione, Costo)

5.1.14 Vincoli integritá referenziale

Nel tradurre lo schema ER in tabelle, sono stati introdotti i seguenti vincoli di integritá referenziale:

- V.I.R.: tra l'attributo UserName della tabella Cliente e l'attributo UserName della tabella Utente.
- V.I.R.: tra l'attributo UserName della tabella Fornitore e l'attributo UserName della tabella Utente.
- V.I.R.: tra l'attributo UserName della tabella Corriere e l'attributo UserName della tabella Utente.
- V.I.R.: tra l'attributo CodCliente della tabella Ordine e l'attributo UserName della tabella Cliente.
- V.I.R.: tra l'attributo CodFornitore della tabella Ordine e l'attributo UserName della tabella Fornitore.
- V.I.R.: tra l'attributo CodCorrirere della tabella Ordine e l'attributo UserName della tabella Corriere
- V.I.R.: tra l'attributo CodSpedizione della tabella Ordine e l'attributo CodSpedizione della tabella Spedizione.
- V.I.R.: tra l'attributo CodCategoria della tabella Prodotto e l'attributo CodCategoria della tabella Categoria.
- V.I.R.: tra l'attributo CodCategoria della tabella Prodotto e l'attributo CodCategoria della tabella Categoria.
- V.I.R.: tra l'attributo CodFornitore della tabella Vende e l'attributo UserName della tabella Fornitore.
- V.I.R.: tra l'attributo CodProdotto della tabella Vende e l'attributo CodProdotto della tabella Prodotto.
- V.I.R.: tra l'attributo CodCategoria della tabella Vende e l'attributo CodCategoria della tabella Categoria.
- V.I.R.: tra l'attributo CodCategoria1 della tabella Consiglia e l'attributo CodCategoria della tabella Categoria.
- V.I.R.: tra l'attributo CodProdotto1 della tabella Consiglia e l'attributo CodProdotto della tabella Prodotto.
- V.I.R.: tra l'attributo CodCategoria2 della tabella Consiglia e l'attributo CodCategoria della tabella Categoria.
- V.I.R.: tra l'attributo CodProdotto2 della tabella Consiglia e l'attributo CodProdotto della tabella Prodotto.

- V.I.R.: tra l'attributo CodProdotto della tabella Acquistati e l'attributo CodProdotto della tabella Prodotto.
- V.I.R.: tra l'attributo CodCategoria della tabella Acquistati e l'attributo CodCategoria della tabella Categoria.
- V.I.R.: tra l'attributo CodOrdine della tabella Acquistati e l'attributo CodOrdine della tabella Ordine.
- V.I.R.: tra l'attributo CodCorriere della tabella Offre e l'attributo UserName della tabella Corriere.
- V.I.R.: tra l'attributo CodSpedizione della tabella Offre e l'attributo Cod Spedizione della tabella Spedizione.

5.2 Implementazione dei vincoli

L'algoritmo che ci permette l'uscita dallo schema E-R consiste in una serie di costrutti che permettono di creare le tabelle della nostra base di dati: le *CREATE TABLE*.

```
CREATE TABLE Impiegato (
Matricola CHAR(6) PRIMARY KEY,
Nome CHAR(20) NOT NULL,
Cognome CHAR(20) NOT NULL,
Dipart CHAR(15),
Stipendio NUMERIC(9) DEFAULT 0,
FOREIGN KEY(Dipart) REFERENCES Dipartimento(NomeDip),
UNIQUE(Cognome,Nome)
)
```

In questo caso abbiamo la relazione Impiegato, che presenta una serie di attributi. Ogni attributo è di un certo tipo e alcuni presentano delle clausole (*keyword* associate agli attributi) per definire vincoli:

- **Matricola** è la *PRIMARY KEY*, ossia la chiave scelta per gli accessi dall'esterno. Segue che il valore dell'attributo non dovrà mai essere nullo e soprattutto essere unico, per permettere l'identificazione dei vari record. Il gestore verificherà che il valore indicato non sia già stato usato.
- **Nome** e **Cognome** sono *NOT NULL*, cioè non possono essere mai vuoti.
- Con *FOREIGN KEY* poniamo un vincolo di integrità referenziale. Si dice che l'attributo **Dipart** fa riferimento alla tabella Dipartimento, precisamente all'attributo **NomeDip**. Il valore indicato deve esistere come chiave nella tabella Dipartimento, altrimenti l'impiegato non viene inserito.
- Con *UNIQUE* si stabilisce che non vi possono essere per forza aventi lo stesso **Nome** e **Cognome**.

```
CREATE TABLE Infrazioni(
Codice CHAR(6) PRIMARY KEY,
Data DATE NOT NULL,
Vigile INTEGER NOT NULL REFERENCES Vigili(Matricola),
Provincia CHAR(2),
Numero CHAR(6),
```

```
FOREIGN KEY(Provincia, Numero)
REFERENCES Auto(Provincia,Numero) ON DELETE SET NULL ON UPDATE CASCADE
)
```

Qua abbiamo una tabella contenente le infrazioni. Ricordiamo quanto detto nella prima lezione: possiamo indicare il metodo per mantenere la consistenza della base di dati. Nell'esempio ho impostato la modifica del valore a NULL in caso di eliminazione e la modifica a cascata in caso di modifica.

5.2.1 Vincoli di integrità generici: *CHECK*

Questi sono i vincoli basi garantiti al momento della tra duzione. Possiamo esprimere vincoli generici specificabili all'interno della tabella mediante la clausola *CHECK*. Restringo i domini e specifico predicati (condizioni come quelle della clausola *WHERE*) che devono essere soddisfatti ogni volta che un valore viene assegnato ad una variabile in quel dominio.

```
CREATE TABLE Impiegato (
Matricola CHARACTER(6),
Cognome CHARACTER(20,
Nome CHARACTER(20),
Dipart CHARACTER(6),
Ufficio CHARACTER(6),
Sesso CHARACTER NOT NULL CHECK (Sesso IN('M','F')),
Superiore CHARACTER(6),
Stipendio INTEGER CHECK (Stipendio <=
(SELECT J.Stipendio FROM Impiegato AS J WHERE J.Matricola = Superiore)
)
)
```

- Voglio verificare che l'attributo **Sesso** assuma solo uno dei valori permessi (M, F)
- Voglio verificare che lo **Stipendio** dell'impiegato sia inferiore a quello del capo. Coinvolgo più attributi effettuando addirittura un'interrogazione.

La clausola CHECK può essere utilizzata all'interno della CREATE DOMAIN per stabilire un dominio.

```
CREATE DOMAIN Voto
AS SMALLINT DEFAULT NULL
CHECK (Value >= 18 AND Value <= 30)
```

5.2.1.1 ASSERTION

Alcuni vincoli possono essere a livello non di un singola tabella ma di più tabelle. Attraverso un'asserzione, presente in una CREATE, stabilisco vincoli tra più tabelle.

```
CREATE ASSERTION AlmenoUnImpiegato
CHECK (1 <= (SELECT COUNT(*) FROM Impiegato))
```

Con le asserzioni posso stabilire un qualunque vincolo predefinito: quando un'asserzione è stabilita ogni variazione del database è consentita solo se le asserzioni non vengono violate.

5.2.1.2 Tipi di controllo

Si hanno due tipi di controllo

- **Immediato**, verifica dopo ogni modifica
- **Differito**, verifica dopo una transazione (sequenza di operazioni)

Possiamo stabilire il tipo di controllo mediante il seguente costrutto

```
SET CONSTRAINTS [NomeAss] (immediate | deferred)
```

Un vincolo immediato non soddisfatto causa l'annullamento dell'operazione che ha causato la violazione (*rollback parziale*), un vincolo differito non soddisfatto comporta l'annullamento di un'intera transazione (*rollback*).

Nome delle asserzioni Le asserzioni hanno un nome, quindi possono essere citate all'interno di istruzioni. Per esempio posso usare la seguente istruzione

```
DROP NomeAss
```

per eliminare l'asserzione con quel nome.

5.2.2 Trigger

Esiste un metodo diverso per inserire i vincoli, in un certo senso un vincolo dinamico. Esso reagisce in base a certi eventi avvenuti nel corso della computazione. Questi oggetti sono definiti *trigger*.
Si stabiliscono delle condizioni che dovranno essere vere: se vere le istruzioni associate al trigger saranno eseguite. Abbiamo quindi i seguenti elementi:

- **Evento** Solitamente si intendono modifiche dello stato del database: INSERT, DELETE, UPDATE.
  Se l'evento avviene il trigger è *attivato*
- **Condizione** La condizione che si deve verificare Se la condizione viene valutata il trigger è *considerato*
- **Azione** Ciò che viene eseguito con condizione vera Se l'azione è eseguita il trigger è *eseguito*

La sintassi dello standard attuale dell'SQL risale al 1999

```
CREATE TRIGGER NomeTrigger
{ BEFORE | AFTER } { INSERT | DELETE | UPDATE [of Column] } ON TabellaTarget
[REFERENCING
{[OLD TABLE [AS] VarTuplaOld] [NEW TABLE [AS] VarTuplaNew] } |
{[OLD [ROW] [AS] VarTabellaOld] [NEW [ROW] [AS] VarTabellaNew] }
]
[FOR EACH { ROW | STATEMENT }]
[WHEN Condizione]
```

5.2.2.1 Tipi di eventi

BEFORE Il trigger viene valutato prima della modifica dell'evento. Normalmente questa modalità è usata quando si vuole verificare una modifica prima che essa avvenga e "modificare la modifica"

AFTER Prima si fa la modifica e successivamente si verificano le modifiche. Questa è la modalità più utilizzata.

5.2.2.2 Granularità degli eventi

Modalità di default *statement-level* Introdotta attraverso l'opzione *for each statement*, il trigger viene considerato ed eseguito una volta sola per ogni statement che lo ha attivato, indipendentemente dal numero di tuple modificate.

5.2.2.3 Conflitto tra trigger

Se si hanno più trigger associati allo stesso evento possono emergere conflitti. Eseguo prima i BEFORE triggers e dopo la modifica gli AFTER triggers. Se i trigger appartengono alla stessa categoria l'ordine di esecuzione è stabilito in base alla data di creazione: i trigger più vecchi hanno priorità più alta.

5.3 Istruzioni fondamentali per le transazioni

- BEGIN TRANSACTION: si specifica l'inizio di una transazione
- COMMIT WORK: la transazione è terminata con successo e quindi le operazioni specificate dopo la BEGIN TRANSACTION vengono eseguite sulla base di dati
- ROLLBACK WORK: si rinuncia all'esecuzione dell'operazioni specificate dopo la BEGIN TRANSACTION

5.4 Integrazione con altri linguaggi di programmazione

Abbiamo già visto che è possibile innesatare il linguaggio SQL con linguaggi di programmazione completamente diversi, per esempio il C++. Un precompilatore, legato al DBMS, viene usato per analizzare il programma e tradurlo in un programma nel linguaggio ospite (sostituendo le istruzioni SQL con chiamate alle funzioni di una API del DBMS). Tutto ciò avverrà ricorrendo a un'apposita libreria e dipenderà dal sistema operativo presente sulla macchina.

Attenzione Il precompilatore è specifico per i linguaggi di programmazione coinvolti.

Conflitto di impedenza Differenza tra ciò che può utilizzare un linguaggio di alto livello e ciò che SQL mette a disposizione. Il conflitto è risolto attraverso una tecnica detta *cursore*: trasformiamo ciò che viene trasmesso dal DBMS, con il cursore scorriamo il buffer e leggiamo le tuple una alla volta. Successivamente il programma ricostituisce la struttura. Si ha una sorta di lettura da file dove il puntatore è appunto il cursore: si inizia un'operazione di lettura e alla fine si conclude con la chiusura del cursore.

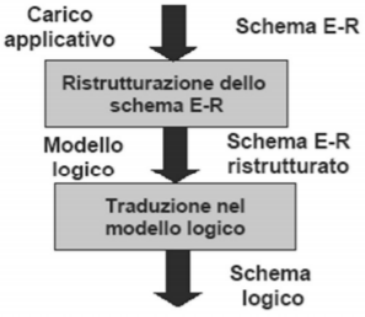
SQL Statico ed SQL Dinamico Le operazioni introdotte fino ad ora sono di tipo statico. Non sempre le istruzioni sono note quando scriviamo il programma! L'SQL dinamico permette la costruzione di istruzioni da parte del programma e non è banale gestire i risultati.
Un esempio si ha con la *Call Level Interface*: si presuppone la presenza di un'interfaccia che chiede l'esecuzione di una certa operazione. Questo permette di interrogare un database anche con strutture diverse.

SQL Immerso VS CLI La prima è più efficiente e permette l'esecuzione di tutte le funzioni di SQL. La CLI è indipendente, generica, permette di accedere a più basi, ma non fornisce tutte le funzionalità.

6 — Giovedì 03/04/2020

6.1 Dal modello concettuale al modello logico

Obiettivo Nel passaggio da modello concettuale a modello logico *traduciamo* in modo automatico lo schema concettuale in uno schema logico che rappresenti gli stessi dati in maniera corretta ed efficiente. Nel compiere questo passaggio indichiamo come dati in ingresso:



- lo schema concettuale
- il modello logico scelto
- le informazioni sul carico applicativo (dimensione dei dati)

Otteniamo, in uscita, lo schema logico.

No traduzione immediata Prima di tradurre dobbiamo ristrutturare lo schema E-R. Obiettivo è:

- semplificare (rimuovendo ciò che non può essere rappresentato nel modello logico)
- ottimizzare (tenendo in conto i requisiti di prestazione).

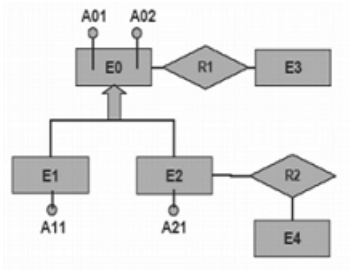
Operazioni eseguibili nella ristrutturazione

1. Eliminazione delle generalizzazioni (come già detto vengono perse nel modello logico)
2. Eliminazione degli attributi multivalore
3. Analisi ed eventuale eliminazione/aggiunta di ridondanze
4. Partizionamento/Accorpamento di entità e relationship

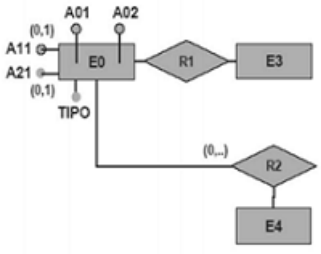
6.1.1 Eliminazione delle generalizzazioni

Le gerarchie devono essere sostituite con entità e relazioni. Scegliamo la soluzione adatta in base al numero e al tipo degli accessi fatti alle singole entità per eseguire le operazioni.

- **Accorpamento delle figlie della generalizzazione nel genitore**. Questo tipo di sostituzione conviene quando gli accessi al padre e alle figlie sono contestuali. Le proprietà delle entità figlie vengono accorpate alle proprietà dell'entità genitore. Aggiungo un ulteriore attributo che mi permette di capire se un'occorrenza del genitore era occorrenza di uno dei figli (o di nessuno con generalizzazioni parziali)

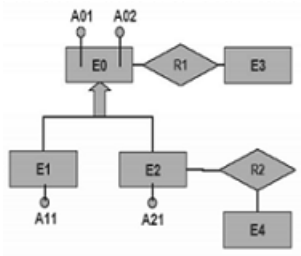


(a) Schema con generalizzazione

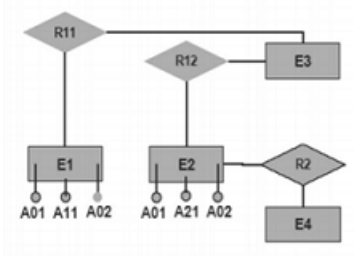


(b) Schema ristrutturato

- **Accorpamento del genitore della generalizzazione nelle figlie**. Questo tipo di sostituzione conviene con accessi solo alle figlie e questi sono distinti dall'una all'altra. Le entità figlie avranno anche gli attributi dell'entità genitore.

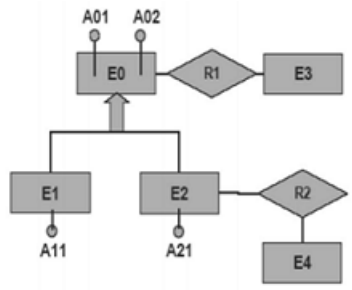


(a) Schema con generalizzazione

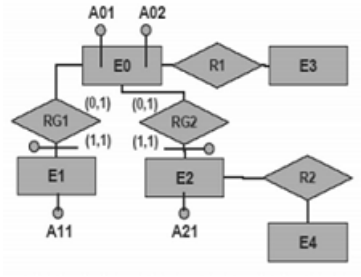


(b) Schema ristrutturato

- **Sostituzione della generalizzazione con relazioni**. Questo tipo di sostituzione conviene quando si effettuano accessi separati alle entità figlie e al padre. Non effettuo trasferimenti di attributi, le due nuove entità sono identificate esternamente. In alcuni casi, in base al tipo di generalizzazione, potrei avere bisogno di stabilire dei vincoli di integrità.

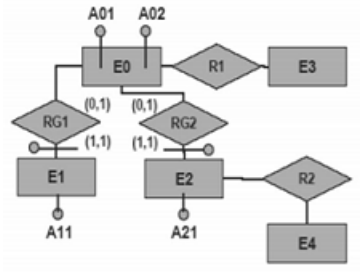


(a) Schema con generalizzazione

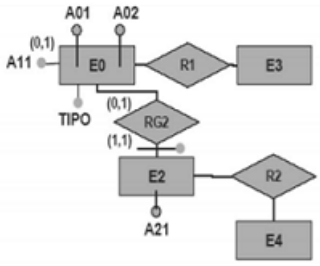


(b) Schema ristrutturato

Soluzioni ibride Individuiamo, soprattutto in gerarchie a più livelli, soluzioni ibride.



(a) Schema con generalizzazione



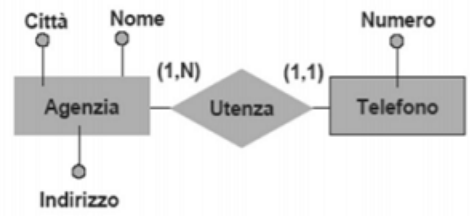
(b) Schema ristrutturato

6.1.2 Eliminazione degli attributi multivalore

Pensiamo a delle soluzioni Si potrebbe pensare a

- Ripetere le tuple con ogni valore diverso dell'attributo
- Una sola tupla dimensionata al numero massimo di numeri di telefono possibili

In entrambi i casi si ha spreco di memoria e con la prima soluzione si potrebbe incorrere in inconsistenze. Segue che la migliore sostituzione consiste nello schema seguente



6.1.3 Analisi ed eventuale eliminazione delle ridondanze

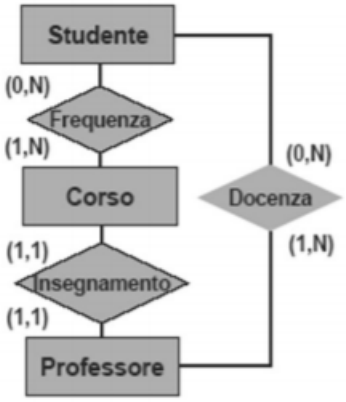
Cos'è una ridondanza Con ridondanza intendiamo un'informazione significativa, ma derivabile da altre.

Forme di ridondanza in uno schema E-R

- attributi derivabili da altri attributi della stessa entità (o associazione)

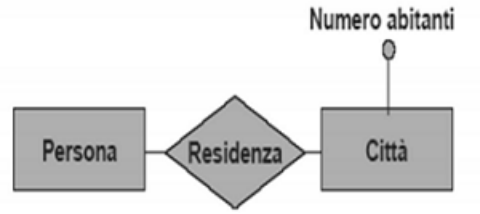
- attributi derivabili da attributi di altre entità (o associazioni)

- associazioni derivabili dalla composizione di altre associazioni (presenza di cicli)



6.1.3.1 Analisi di una ridondanza

Abbiamo il seguente schema con ridondanza evidente (l'attributo *Numero abitanti*).



Devo decidere se eliminare la ridondanza presente o mantenerla in base ad una valutazione del costo delle operazioni

Vantaggi Interrogazioni semplificate

Svantaggi Appesantimento degli aggiornamenti e maggiore occupazione di spazio

Come analizzo le prestazioni? Per ottimizzare abbiamo bisogno di analizzare le prestazioni, ma come possiamo fare questo su uno schema concettuale? Prendiamo i seguenti indicatori di prestazione:

- spazio: numero di occorrenze previste. Realizziamo una **tavola dei volumi** in cui elenchiamo i vari elementi e il loro volume (il numero di record).

- tempo: numero di occorrenze (di entità e relationship) visitate per portare a termine un'operazione. Partendo dai dati a disposizione costruisco una **tavola degli accessi** basata su uno schema di navigazione.

6.1.3.2 Esempio di risoluzione (Mantenere il numeroAbitanti come attributo?)

Riprendiamo lo schema a inizio sessione. Vogliamo eseguire le seguenti operazioni:

- **Operazione 1**: memorizza una nuova persona con la relativa residenza, supponendo che la città sia già presente (500 volte al giorno)
- **Operazione 2**: stampa tutti i dati di una città (incluso il numero di abitanti) (2 volte al giorno)

Devo decidere se mantenere l'attributo *numeroAbitanti* o rimuoverlo. La tabella dei volumi è la seguente:

Concetto	Tipo	Volume
Città	E	200
Persona	E	1000000
Residenza	R	1000000

Analizziamo le tavole degli accessi con e senza ridondanza per entrambe le operazioni

Operazione 1

Concetto	Costrutto	Accessi	Tipo
Persona	Entità	1	S
Residenza	Relazione	1	S
Città	Entità	1	L
Città	Entità	1	S

Operazione 2

Concetto	Costrutto	Accessi	Tipo
Città	Entità	1	L

(a) Accessi con ridondanza

Operazione 1

Concetto	Costrutto	Accessi	Tipo
Persona	Entità	1	S
Residenza	Relazione	1	S

Operazione 2

Concetto	Costrutto	Accessi	Tipo
Città	Entità	1	L
Residenza	Relazione	5000	L

(b) Accessi senza ridondanza

Conclusioni Se teniamo la ridondanza abbiamo:

- Ho 1500 accessi in scrittura e 500 accessi in lettura al giorno per l'operazione 1
- L'operazione 2 è trascurabile.

Contando doppi gli accessi in scrittura (poichè hanno peso maggiore) otteniamo 3500 accessi al giorno. Consideriamo le stesse operazioni senza ridondanza, abbiamo:

- Ho 1000 accessi in scrittura al giorno per l'operazione 1
- Ho 10.000 accessi in lettura al giorno per l'operazione 2

Contando doppi gli accessi (poichè hanno peso maggiore) in scrittura ho un totale di 12.000 accessi giornalieri.

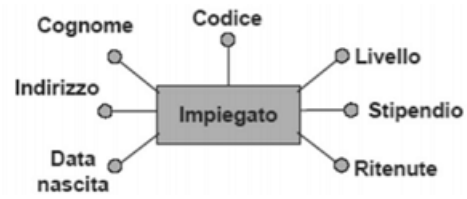
6.1.4 Partizionamento/accorpamento di entità e relationship

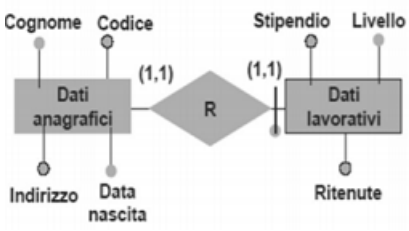
Le ristrutturazioni sono effettuate per rendere più efficienti le operazioni. Posso fare ciò riducendo gli accessi con queste strategie

- separare gli attributi di un concetto che vengono acceduti separatemente
- raggruppare attributi di concetti diversi acceduti insieme

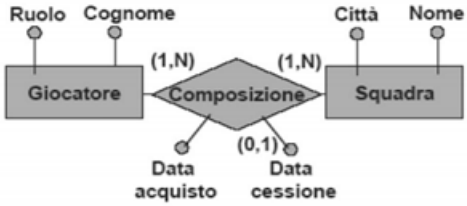
Abbiamo i seguenti casi

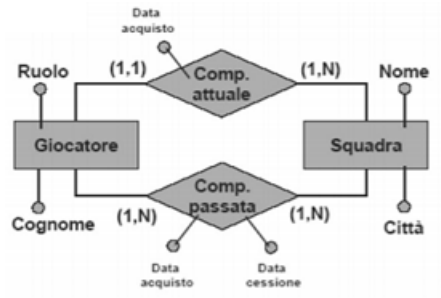
- Partizionamento di entità



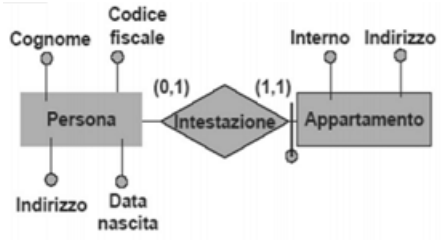


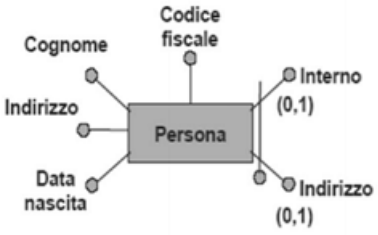
- Partizionamento di relationship





- Accorpamento di entità/relationship





Attenzione Valutare osservando le cardinalità!

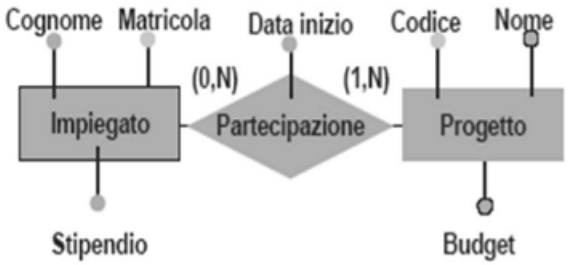
6.1.5 Traduzione verso il modello relazionale

In questa traduzione:

- Le entità diventano relazioni sugli stessi attributi

- Le associazioni diventano relazioni sugli identificatori delle entità coinvolte (più gli attributi propri)

Relationship molti a molti Il seguente schema



Mi porta ad avere le seguenti relazioni

`Impiegato(Matricola, Cognome, Stipendio)`
`Progetto(Codice, Nome, Budget)`
`Partecipazione(Matricola, Codice, DataInizio)`

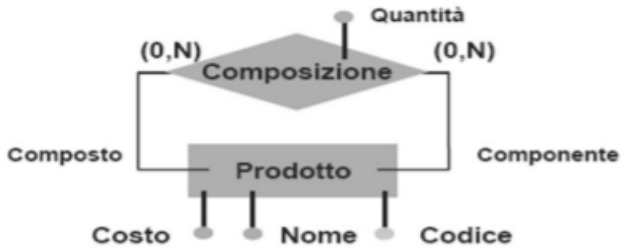
con i seguenti vincoli di integrità referenziale:

- Matricola in Partecipazione e (la chiave di) Impiegato
- Codice in Partecipazione e (la chiave di) Progetto

Nomi più espressivi per gli attributi della chiave della relazione che rappresenta la relationship I nomi degli attributi devono rendere chiara la relazione presente. Modifichiamo la relazione Partecipazione, che diventa così (ho alterato i nomi degli attributi chiave)

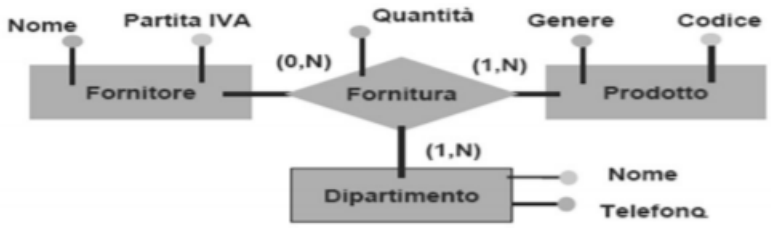
`Partecipazione(Impiegato, Progetto, DataInizio)`

Relationship ricorsive



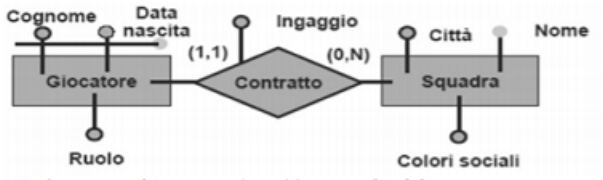
Prodotto(Codice, Nome, Costo)
Composizione (Composto, Componente, Quantità)

Relationship n-arie



Fornitore(PartitaIVA, Nome)
Prodotto(Codice, Genere)
Dipartimento(Nome, Telefono)
Fornitura (Fornitore, Prodotto, Dipartimento, Quantità)

Relationship uno-a-molti



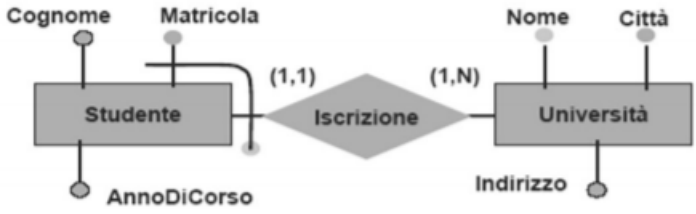
Giocatore (Cognome, DataNascita, Ruolo)
Contratto (CognGiocatore, DataNascG, Squadra, Ingaggio)
Squadra (Nome, Città, ColoriSociali)

Non potrei ottenere una soluzione compatta? Osservo che la cardinalità di Contratto rispetto a Giocatore è $(1, 1)$. Posso rinunciare alla relazione Contratto ottenendo la seguente situazione

`Giocatore(Cognome,DataNasc, Ruolo, Squadra, Ingaggio)`
`Squadra(Nome, Città, ColoriSociali)`

dove ho vincolo di integrità referenziale fra Squadra in giocatore e la chiave di squadra. Osservo che la cardinalità minima della relationship è 0 per Giocatore, segue che il valore dell'attributo Squadra, in Giocatore, può assumere valore nullo.

Entità con identificatore esterno



Studente (Matricola, Università, Cognome, AnnoDiCorso)
Università (Nome, Città, Indirizzo)
- con vincolo ...

Relationship uno-a-uno Seguono due sempi dove variano le cardinalità

Impiegato (Codice, Cognome, Stipendio)
Dipartimento (Nome, Sede, Telefono, Direttore, DataInizio)

- con vincolo di integrità referenziale, e senza valori nulli

Impiegato (Codice, Cognome, Stipendio)
Dipartimento (Nome, Sede, Telefono)
Direzione (Direttore, Dipartimento, Datainizio)

- con vincoli di integrità referenziale, senza valori nulli

7 — Mercoledì 22/04/2020

7.1 Qualità delle relazioni

Abbiamo progettato il nostro sistema attraverso un linguaggio concettuale, abbiamo utilizzato il modello E-R, abbiamo verificato le ridondanze e l'efficienza ottenendo alla fine il modello logico, cioè una serie di schemi di relazione. Adesso dobbiamo verificare la *qualità delle relazioni*, cioè se presentano caratteristiche utili per il mantenimento della base di dati.

Obiettivo Valutare la qualità della progettazione di schemi relazionali, capire se un raggruppamento di attributi in uno schema di relazione sia migliore di un altro.

Approccio adottato *top-down*.

Obiettivi impliciti

- Conservazione dell'informazione, cioè il mantenimento di tutti i concetti espressi mediante il modello concettuale, inclusi tipi di attributi, tipi di entità e tipi di associazioni.
- La minimizzazione delle ridondanze, cioè l'evitare la memorizzazione ripetuta della stessa informazione e quindi la necessità di effettuare molteplici aggiornamenti al fine di mantenere la consistenza tra le diverse copie della medesima informazione.

Linee guida

- **Semplice è bello**! Uno schema di relazione deve essere progettato in modo da rendere semplice la spiegazione del suo significato. Non devo raggruppare attributi provenienti da più tipi di entità e tipi di relazione in un'unica relazione. Se uno schema corrisponde a un solo tipo di entità o a un solo tipo di relazione risulta semplice spiegarne il significato, evitando così l'emergere di ambiguità semantiche.
- **Niente anomalie**! Gli schemi vanno progettati in modo da evitare anomalie nell'inserimento, nella cancellazione o nella modifica. Se possono presentarsi anomalie vanno rilevate e dobbiamo fare in modo che i programmi legati alla base di dati operino in modo corretto.
- **Evitare frequenti valori nulli**! Evitare di porre in una relazione attributi i cui valori possono essere frequentemente nulli. Se questi sono inevitabili ci si assicuri che si presentino solo in casi eccezionali rispetto al numero di tuple di una relazione.

Esempio 1 Prendiamo la seguente relazione

```
Fattura(CodF, CodProd, TotDaPagare, CostoNettoProd, IVA)
```

Osservo che:

- Il codice della fattura (CodF) determina il prodotto comprato, quindi il codice del prodotto (CodProd) e il costo totale (TotDaPagare)
- Il codice del prodotto (CodProd) determina il costo netto (CostoNettoProd) e l'iva da pagare (IVA)
- CostoNettoProd e IVA determinano quanto bisogna pagare (TotDaPagare)

Devo mantenere il tutto consistente. TotDaPagare è determinato da altri attributi, per evitare anomalie conviene rimuoverlo e calcolare il costo totale direttamente nella query.

Esempio 2 Prendiamo la seguente relazione

```
Anagrafe(CF, NomeP, Indirizzo, NomeC, NumAb)
```

Osservo che :

- Il codice fiscale (CF) determina il nome della persona (NomeP), l'indirizzo (Indirizzo) e il nome della città (NomeC)
- Il nome della città (NomeC) determina il numero di abitanti (NumAb)

Il numero di abitanti è ripetuto tante volte quanti sono i residenti, questo valore deve essere mantenuto consistente (cioè uguale) per ogni persona di una certa città. Risulta ovvio che calcolare per ogni persona il numero di abitanti sia alquanto pesante. **Come risolviamo?** Trasformiamo Anagrafe in due schemi separati (con vincolo di integrità referenziale su NomeC e un vincolo aggiuntivo su NumAb)

```
Persona(CF, NomeP, Indirizzo, NomeC)
Residenza(NomeC, NumAb)
```

7.1.1 Approccio formale

Abbiamo visto che ci sono attributi che dipendono da altri attributi, devo verificare se ci sono troppe dipendenze tra gruppi di attributi all'interno di una tabella. A tal proposito parliamo di **dipendenze funzionali**, che esprimono legami semantici tra due gruppi di attributi di uno schema di relazione.

- Una dipendenza funzionale è una proprietà dello schema di relazione, non di un particolare stato valido
- Una dipendenza funzionale non può essere dedotta a partire da uno stato valido, ma deve essere definita esplicitamente da qualcuno che conosce la semantica degli attributi (quindi non determino proprietà osservando i valori delle istanze, potrei confondere proprietà valide solo per quell'istanza con proprietà dello schema di relazione)

Definizione Consideriamo la relazione r su $R(X)$ e due sottoinsiemi non vuoti Y e Z di X. Si dice che esiste in r una dipendenza funzionale da Y a Z se, per ogni coppia di ennuple t_1 e t_2 di r con gli stessi valori su Y risulta che t_1 e t_2 hanno gli stessi valori anche su Z.

Notazione La notazione adottata è la seguente:

$$Y \longrightarrow Z$$

Attenzione Non è detto che se io ho $Y \longrightarrow Z$ abbia il contrario, cioè $Z \longrightarrow Y$

Forma normale Attraverso le dipendenze funzionali esprimo tutto quello che so riguardante la base di dati. In base a queste dipendenze vorrei costruirmi una caratteristica dello schema per cui valgono quelle dipendenze: ciò significa che se quello schema presenta quelle caratteristiche allora si ha consistenza.
Questo schema è detto **forma normale**, che garantisce l'assenza di particolari difetti dallo schema (determina un livello di qualità). In base alle proprietà che ho dato posso determinare se lo schema si trova in una certa forma normale o in nessuna.

Normalizzazione Se applico a un certo schema la definizione di forma normale e individuo che questo non è in tale forma, allora compio un processo di *normalizzazione*. La normalizzazione è utilizzata come tecnica di verifica dei risultati della progettazione, non costituisce metodologia di progetto.

Esempio 3 Vediamo la seguente tabella contenente le informazioni di una certa azienda

Impiegato	Stipendio	Progetto	Bilancio	Funzione
Rossi	20	Marte	2	tecnico
Verdi	35	Giove	15	progettista
Verdi	35	Venere	15	progettista
Neri	55	Venere	15	direttore
Neri	55	Giove	15	consulente
Neri	55	Marte	2	consulente
Mori	48	Marte	2	direttore
Mori	48	Venere	15	progettista
Bianchi	48	Venere	15	progettista
Bianchi	48	Giove	15	direttore

Abbiamo i cognomi, gli stipendi, i progetti con il loro bilancio e la funzione nel progetto di ciascun dipendente. Ogni impiegato può partecipare a più progetti, sempre con lo stesso stipendio, e con una sola funzione per progetto. Neri, per esempio, lavora su tre progetti differenti.
Potrei incorrere nelle seguenti anomalie effettuando modifiche in modo scorretto:

- **Anomalia di aggiornamento**: se lo stipendio di un impiegato varia è necessario andarne a modificare il valore più tuple
- **Anomalia di cancellazione**: se un impiegato si licenzia dobbiamo cancellarlo in diverse tuple (quindi rimuoverlo da tutti i progetti)
- **Anomalia di aggiornamento**: se un progetto varia il suo bilancio devo modificare l'attributo relativo in tutte le tuple dei dipendenti coinvolti nel progetto.

La causa sta, ovviamente, nella ripetizione dello stipendio di un impiegato e del bilancio di un progetto. L'errore è stato compiuto proprio nella fase di progettazione: abbiamo usato un'unica relazione per rappresentare gruppi di informazioni eterogenee. Trasformiamo le informazioni dette prima in dipendenze

- Ogni impiegato ha un solo stipendio: Impiegato $\longrightarrow$ Stipendio
- Ogni progetto ha un solo bilancio: Progetto $\longrightarrow$ Bilancio
- Ogni impiegato ha una sola funzione per progetto: Impiegato Progetto $\longrightarrow$ Funzione

Osservando la tabella individuiamo pieno rispetto di queste dipendenze. **Ma ne esistono anche altre?** Abbiamo le cosiddette **dipendenze banali**, che sono sempre soddisfatte (proprietà ovvie di una relazione). Si osserva

$$Impiegato\ Progetto \longrightarrow Progetto$$

Si dice che:

- $Y \to A$ è non banale se A non compare tra gli attributi di Y
- $Y \to Z$ è non banale se nessun attributo in Z appartiene a Y.

Legame tra dipendenze e anomalie Osserviamo che le prime due dipendenze comportano ripetizioni, mentre la terza no. Impiegato e Progetto non sono chiavi, Impiegato Progetto è chiave! La relazione contiene alcune informazioni legate alla chiave e altre ad attributi che non lo sono.

7.1.2 Teoria delle dipendenze

7.1.2.1 Implicazione

Sia F un insieme di dipendenze funzionali definite su $R(Z)$ e sia $X \longrightarrow Y$.

- Si dice che F implica la dipendenza $X \to Y$ ($F \Rightarrow X \to Y$) se per ogni istanza r di R che verifica tutte le dipendenze in F, risulta verificata anche $X \to Y$.
- Si dice anche che $X \to Y$ è implicata da F

7.1.2.2 Chiusura

Dato un insieme di dipendenze funzionali F definite su $R(Z)$ la chiusura di F è l'insieme di tutte le dipendenze funzionali implicate da F.

$$F^+ = \{X \to Y | F \Rightarrow X \to Y\}$$

Dato un insieme di dipendenze funzionali F definite su $R(Z)$, un'istanza r di R che soddisfa F soddisfa anche F^+.

7.1.2.3 Superchiave

Dato $R(Z)$ ed un insieme F di dipendenze funzionali, un insieme di attributi K appartenenti a Z si dice superchiave di R se la dipendenza funzionale $K \to Z$ è logicamente implicata da F ($K \to Z$ è in F^+)

Chiave Se nessun sottoinsieme proprio di K è superchiave di R allora K si dice chiave di R.

7.1.3 Calcolo di F^+ (regole di Armstrong)

Utilizziamo un approccio costruttivo per arrivare all'algoritmo. Esistono una serie di regole, definite da *Armstrong*, che mi permettono di arrivare a trovare l'insieme di tutte le dipendenze funzionali presenti nella chiusura.

Regole di inferenza di Armstrong

- **Riflessività**: se $Y \subseteq X$, allora $X \to Y$
- **Additività (o espansione)**: se $X \to Y$, allora $XZ \to YZ$ per qualunque Z
- **Transitività**: se $X \to Y$ e $Y \to Z$ allora $X \to Z$

Regole derivate di Armstrong

- **Regola di unione**: $\{X \to Y, X \to Z\} \Rightarrow X \to YZ$
- **Regole di pseudotransitività (o aggiunta sinistra)**: $\{X \to Y, WY \to Z\} \Rightarrow XW \to Z$
- **Regola di decomposizione**: se $Z \subseteq Y, X \to Y \Rightarrow X \to Z$

Proprietà delle regole di Armstrong

- **Teorema correttezza**: le regole di inferenze sono corrette, cioè, applicandole a un insieme F di dipendenze funzionali, si ottengono solo dipendenze logicamente implicate da F.
- **Teorema completezza**: le regole di inferenza sono complete, cioè, applicandole ad un insieme F di dipendenze funzionali, si ottengono tutte le dipendenze logicamente implicate da F.
- **Teorema minimalità**: le regole di inferenza sono minimali, cioè, ignorando anche una sola di esse, l'insieme di regole che rimangono non è più completo.

Esempio di dimostrazione Dimostrare che per ogni istanza di relazione

$$X \to Y \Rightarrow XZ \to YZ$$

Supponiamo per assurdo che esista un'istanza r di R in cui valga $X \to Y$ ma non $XZ \to YZ$. Devono esistere due tuple t_1 e t_2 di r tali che:

1. $t_1[X] = t_2[X]$
2. $t_1[Y] = t_2[Y]$
3. $t_1[XZ] = t_2[XZ]$
4. $t_1[YZ] \neq t_2[YZ]$

Cioè è assurdo. Dalla prima e dalla terza deduciamo che

$$t_1[Z] = t_2[Z]$$

con questa e la seconda concludiamo che

$$t_1[YZ] = t_2[YZ]$$

in contraddizione con la quarta!

Riprendiamo l'esempio 3 Avevamo le seguenti dipendenze funzionali:

- Impiegato $\longrightarrow$ Stipendio
- Progetto $\longrightarrow$ Bilancio
- Impiegato Progetto $\longrightarrow$ Funzione

Utilizzando la *regola di attività* sulle prime due otteniamo

- Impiegato Progetto $\longrightarrow$ Stipendio Progetto
- Impiegato Progetto $\longrightarrow$ Bilancio Impiegato
- Impiegato Progetto $\longrightarrow$ Funzione (Uguale a prima)

Infine, con la *regola di unione*, avremo

- Impiegato Progetto $\longrightarrow$ Stipendio Progetto Impiegato Bilancio Funzione

Quindi Impiegato Progetto è chiave. Posso trovare altre dipendenze funzionali in F^+?

7.1.4 Equivalenza

Dato un insieme F di dipendenze funzionali è molto utile poter determinare se un insieme G sia equivalente ad F. Affermo che F e G sono equivalenti se

$$F^+ = G^+$$

cioè per ogni $X \to Y \in F$ deve essere

$$X \to Y \in G^+$$

e, viceversa, per ogni $X \to Y \in G$ deve essere

$$X \to Y \in F^+$$

Esempio Supponiamo di avere questi due insiemi

$$F = \{\ A \to C, AC \to D, E \to AD, E \to H\}$$
$$G = \{\ A \to CD, E \to AH\}$$

Voglio verificare l'equivalenza di F e G. Devo dimostrare che le dipendenze funzionali in F sono derivabili dalle dipendenze funzionali in G, e viceversa.

- $A \to CD \Rightarrow \boxed{A \to C}, A \to D$
- $A \to CD, CCD \to CD \Rightarrow AC \to CD \Rightarrow AC \to C, \boxed{AC \to D}$
- $E \to AH \Rightarrow E \to A, \boxed{E \to H}$
- $E \to A, A \to D \Rightarrow E \to D$
- $E \to A, E \to D \Rightarrow \boxed{E \to AD}$

Ovviamente dovremo fare la stessa cosa al contrario. Da tutto questo capiamo quanto sia costoso il calcolo di F^+. Generalmente ci interessa sapere se F^+ contiene una certa dipendenza.

Chiusura transitiva Come alternativa posso utilizzare la *chiusura transitiva* di un insieme di attributi X rispetto a F. Data una relazione $R(U)$, un insieme di attributi $X \subseteq U$, la chiusura detta consiste nell'insieme degli attributi che dipendono da X (esplicitamente o implicitamente).

$$X^{+F} = \{A \mid A \in U\ \wedge\ F \Longrightarrow X \to A\}$$

Posso dimostrare che $X \to Y$ è in F^+ se $Y \subseteq X^+$

Riprendiamo l'esempio Invece di verificare se $X \to Y$ in F è anche in G^+ verifico se $Y \subseteq (X)^{+G}$ (chiusura di X rispetto a G). Le varie chiusure qua sotto sono state trovate analizzando l'insieme di dipendenze funzionali G.

- per $A \to C$ risulta $(A)^{+G} = ACD$. Ok ho $C \subseteq (A)^{+G}$
- per $AC \to D$ risulta $(AC)^{+G} = ACD$. Ok ho $D \subseteq (AC)^{+G}$
- per $E \to AD$ risulta $(E)^{+G} = EADCH$. Ok ho $AD \subseteq (E)^{+G}$
- per $E \to H$ risulta $(E)^{+G} = EHADC$. Ok ho $H \subseteq (E)^{+G}$

E viceversa per ogni dipendenza funzionale in G.

Esempio 2 Supponiamo di avere il seguente insieme

$$F = \{A \to B, BC \to D, B \to E, E \to C\}$$

Calcoliamo A^+, cioè l'insieme di attributi dipendenti da A. Trovo che

- $A^+ = A$
- $A^+ = AB$ poichè $A \to B$ e $A \subseteq A^+$

- $A^+ = ABE$ poichè $B \to E$ e $B \subseteq A^+$
- $A^+ = ABEC$ poichè $E \to C$ e $E \subseteq A^+$
- $A^+ = ABECD$ poichè $BC \to D$ e $BC \subseteq A^+$

Quindi da A dipendono tutti gli attributi dello schema, ovvero A è superchiave oltre che chiave.

7.1.5 Definizione aggiornata di equivalenza

Abbiamo modificato la definizione di equivalenza. Dati F e G, essi sono equivalenti se

- per ogni $X \to Y \in F, Y \in X^{+G}$, e
- per ogni $Z \to W \in G, W \in Z^{+F}$.

7.1.6 Importanza della chiusura in un insieme di attributi

Dato $R(Z)$ con le sue dipendenze F, la chiusura di un insieme $X \subseteq Z$ di attributi è fondamentale per diversi scopi:

- Possiamo verificare se una dipendenza funzionale è logicamente implicata da F

$$X \to Y \in F^+ \iff Y \subseteq X^{+F}$$

- Possiamo verificare se un insieme di attributi è chiave o superchiave:
  - X è superchiave di R se e solo se $X \to Z \in F^+$, cioè se e solo se $Z \subseteq X^{+F}$
  - X è chiave di R se e solo se $X \to Z \in F^+$ e non esiste alcun sottoinsieme $Y \subset X$ eliminando almeno un elemento, tale che $Z \subseteq Y^{+F}$

7.1.7 Ridondanze di un insieme di dipendenze funzionali

Alcuni attributi di una dipendenza funzionale possono essere ridondanti. L'obiettivo è individuare forme equivalenti più semplici! Posso semplificare rimuovendo attributi sul lato sinistro di una DF (poichè non essenziali per determinare la parte destra) o rimuovere intere DF poichè ridondanti. Ricorriamo alle regole di Armstrong:

- Applicazione della *Regola di unione*: $\{A \to B, B \to C, A \to CD\}$ può essere semplificata in $\{A \to B, B \to C, A \to C, A \to D\}$
- Attributi estranei a sx: $\{A \to B, B \to C, AC \to D\}$ può essere semplificata in $\{A \to B, B \to C, A \to D\}$

Un insieme F di dipendenze funzionali può contenere dipendenze ridondanti, ovvero ottenibili tramite altre dipendenze di F. Ricorriamo alla transitività: $A \to C$ è ridondante in $\{A \to B, B \to C, \boxed{A \to C}\}$. Ricordiamo che la regola fondamentale è semplificare al massimo la rappresentazione dell'informazione.

7.1.8 Dipendenze funzionali semplici

Possiamo portare un insieme di dipendenze funzionali F in forma standard, quella in cui sulla destra c'è un solo attributo.
Il seguente insieme

$$F = \{AB \to CD, AC \to DE\}$$

Possiamo riscriverlo così

$$F = \{AB \to C, AB \to D, AC \to D, AC \to E\}$$

Quanto fatto è solo un primo step.

7.1.9 Attributi estranei

In alcune dipendenze funzionali è possibile che ci siano attributi inutili (appunto estranei) sul lato sinistro. Come li identifico?

- Supponiamo di avere $F = \{AB \to C, A \to B\}$
- Ho $A^+ = A$ e $B^+ = B$
- $A^+ = AB$ poichè $A \to B$ e $A \subseteq A^+$
- $A^+ = ABC$ poichè $AB \to C$ e $AB \subseteq A^+$

C dipende solo da A, e in $AB \to C$ l'attributo B è estraneo poichè a sua volta dipende da A. Possiamo riscrivere l'insieme di dipendenze funzionali nel seguente modo

$$F^+ = \{A \to C, A \to B\}$$

Morale della favola In una dipendenza funzionale del tipo $AX \to B$ l'attributo A è estraneo se X^+ include B (ovvero X da solo determina B).

7.1.10 Algoritmo per la ridondanza di una dipendenza funzionale

Dopo aver eliminato gli attributi estranei si deve verificare se sono presenti intere dipendenze funzionali inutili, cioè se intere dipendenze funzionali sono implicate da altre. Come faccio a capire se una dipendenza del tipo $X \to A$ è ridondante?

- La elimino dall'insieme F
- Calcolo la chiusura di X (X^+)
- Verifico se include A, cioè se con le dipendenze funzionali che restano riusciamo ancora a dimostrare che $X \to A$

7.1.11 Copertura minimale

Un insieme di dipendenze funzionali F è minimale se:

- Nella parte destra di ogni dipendenza funzionale c'è un solo attributo
- Non si possono togliere attributi nella parte sinistra di qualche FD senza perdere l'equivalenza nell'insieme ottenuto con F
- Non si possono togliere intere dipendenze funzionali senza perdere l'equivalenza nell'insieme ottenuto con F

Segue che una copertura minimale di un insieme F è un insieme equivalente a F, ma di complessità minore.

NB La copertura minimale non è unica.

Esempio di individuazione di copertura minimale

- Abbiamo il seguente insieme: $F = \{AB \to C, B \to A, C \to A\}$
- A è estraneo in $AB \to C$, quindi: $F = \{B \to C, B \to A, C \to A\}$
- $B \to A$ è ridondante, quindi ottengo: $F = \{B \to C, C \to A\}$

Osserviamo che la dipendenze funzionale ridondante non può essere eliminata se prima non eliminiamo l'attributo estraneo. Segue l'importanza dello svolgere i passi spiegati nell'ordine mostrato!

8 — Venerdì 24/04/2020

8.1 Forme normali e normalizzazione

8.1.1 Forma normale di Boyce-Codd (BCNF)

Uno schema, se rispetta questa forma normale, è privo di informazione duplicata. Una relazione r è in forma normale di Boyce-Codd se per ogni dipendenza funzionale (non banale) $X \to Y$ definita su di essa, X è superchiave di r. Questa forma normale richiede che i concetti in una relazione siano omogenei (cioè che tutte le proprietà siano direttamente associate alla chiave)

Presenza di dipendenze non banali Se un insieme F di dipendenze per R non è in BCNF, allora in F c'è almeno una dipendenza $X \to Y$ non banale con X non superchiave di R.

Teorema Dato uno schema R e un insieme F di dipendenze funzionali, se F non contiene alcune $X \to Y$ non banale con X non superchiave di R, allora neanche F^+ la contiene.

Metodo di verifica Analizzo una ad una le dipendenze non banali in F verificando se ognuna ha una superchiave come membro sinistro.

- Occorre saper verificare se F è minimale
- Occorre saper verificare se un insieme di attributi è superchiave di una relazione.

Come normalizziamo? Se una relazione non è in BCNF la rimpiazziamo con altre relazioni BCNF: faccio questo decomponendo sulla base delle dipendenze funzionali, separando i concetti.

Esempio Prendiamo la seguente tabella, che presenta certe dipendenze funzionali.

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

$$\text{Impiegato} \rightarrow \text{Sede}$$
$$\text{Progetto} \rightarrow \text{Sede}$$

Procediamo alla decomposizione, ottenendo quanto segue

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Progetto	Sede
Marte	Roma
Giove	Milano
Saturno	Milano
Venere	Milano

Un buon indicatore della correttezza di quanto stiamo facendo è la ricomposizione della tabella iniziale dopo aver effettuato le divisioni: se non posso riottenere la relazione iniziale allora qualcosa non va. Si individua che alcune composizioni di Boyce-Codd potrebbero presentare questo problema

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Progetto	Sede
Marte	Roma
Giove	Milano
Saturno	Milano
Venere	Milano

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano
Verdi	Saturno	Milano
Neri	Giove	Milano

se io faccio JOIN con la sede ottengo tuple che precedentemente non esistevamo. Segue che dobbiamo trovare composizioni BCNF particolari!

Decomposizione senza perdita Una istanza r di una relazione R si decompone senza perdita su X_1 e X_2 se il join naturale delle proiezione di r su X_1 e X_2 è uguale ad r stessa.

Algoritmo per la decomposizione in BCNF Assumiamo che tutte le dipendenze funzionali in F abbiano un unico attributo come membro destro, e che U sia l'insieme di tutti gli attributi di R.

- $Decomponi(R, F)$
  - Se esiste $X \rightarrow A$ in F con X non superchiave di R:
    - Sostituisco R con una relazione R_1 con attributi $U - A$ ed una relazione R_2 con attributi $X \cup A$

- $Decomponi(R_1, F_{U-A})$
- $Decomponi(R_2, F_{X \cup A})$

Esempio Riprendiamo la solita tabella di prima. Abbiamo

$$F = \{Impiegato \to Sede, Progetto \to Sede\}$$

Applico l'algoritmo e ottengo

$$Decomponi(R, F) = R_1(Impiegato, Progetto), R_2(Impiegato, Sede)$$

Teorema, correttezza dell'algoritmo della decomposizione Qualunque sia l'input, l'esecuzione dell'algoritmo su tale input termina e produce una decomposizione della relazione originaria tale che:

- Ogni relazione ottenuta è in BCNF
- La decomposizione è senza perdita nel JOIN

Proiezione delle dipendenze funzionali di $R(U)$ su $X \subseteq U$ La proiezione di F su X, denotata da F_x è l'insieme delle dipendenze funzionali $Z \to Y$ in F^+ che coinvolgono solo attributi in X, cioè tali che $Z \subseteq X$ e $Y \subseteq X$.

Calcolare la proiezione delle DF su X

- CalcolaProiezione(F, X) :=
  - Pongo $result = \{\emptyset\}$
  - Per ogni sottoinsieme proprio $S \subset X$, per ogni attributo $A \in X$ tale che $A \notin S$ e tale che non esiste alcun sottoinsieme S' di S tale che

    $$S' \to A$$

    è in result, allora dico che se $A \subseteq S^+$ allora

    $$result = result \cup \{S \to A\}$$

Il metodo è funzionante ma in alcuni casi la proiezione di F su X può avere dimensione esponenziale rispetto alla dimensione di F ed X. Si può dimostrare che nessun insieme equivalente a quello delle proiezioni non puiò avere cardinalità minore.

Esempio da internet per capire una spiegazione arzigogolata Supponiamo di avere $R(A, B, C)$ e il seguente gruppo di dipendenze funzionali

$$F = \{A \to B, B \to C, C \to A\}$$

individuo le seguenti proiezioni

- $F_{AB} = \{A \to B, B \to A\}$.

- $F_{AC} = \{A \to C, C \to A\}$

La prima proiezione è interessante perchè ci fa capire che non possiamo limitarci a verificare solo che $X \subseteq AB$ ed $Y \subseteq AB$ con $X \to Y$. Per comodità dobbiamo trovare, dato un insieme F, la sua copertura minimale. A quel punto dobbiamo stare attenti agli attributi sinistri, cioè controllare se essi non dipendono a loro volta dagli elementi su cui stiamo facendo la proiezione. In questo caso:

- Pongo da $B \to C$ e $C \to A$

  $$BC \to A$$

  Ovviamente C dipende da B e per questo può essere rimosso.

- BC è l'insieme S che dicevamo nella spiegazione dell'algoritmo. In questo caso esiste un sottoinsieme S' formato solo da C... Infatti possiamo dire che

  $$C \to A$$

Proprietà dell'algoritmo di decomposizione A seconda dell'ordine con cui si considerano le dipendenze funzionali il risultato della decomposizione può cambiare. Riprendiamo l'esempio di prima e consideriamo le dipendenze funzionali in ordine diverso. Otteniamo uno schema diverso

```
Rx(Impiegato, Sede)
Ry(Impiegato, Progetto)
```

Supponiamo di voler inserire un nuovo impiegato, che sta a Milano, che lavorerà sul progetto Marte, con sede a Roma.

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Impiegato	Progetto
Rossi	Marte
Verdi	Giove
Verdi	Venere
Neri	Saturno
Neri	Venere
Neri	Marte

L'inserimento è legittimo ma emerge un'inconsistenza relativa alla sede della persona e la sede del progetto

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano
Neri	Marte	Milano

Non capiamo più se la sede indicata è quella dell'utente o del progetto! Non ho perdita di informazione attraverso il JOIN (le tuple che già esistevano prima della divisione della relazione esistono), ma la tupla nuova rappresenta qualcosa di inconsistente (la sede di Marte è a Milano, quando in realtà il progetto ha sede a Roma). Segue che quanto detto ci garantisce il JOIN ma non le dipendenze (nell'esempio sono state perse).

Verifica di decomposizione senza perdita di dipendenze La definizione di decomposizione senza perdita di dipendenze (considerando due relazioni $R1(X)$ ed $R2(Y)$ ottenute dalla decomposizione) è basata sul verificare che

$$(F_X \cup F_Y)^+ = F^+$$

cioè la chiusura dell'insieme F di dipendenze funzionali equivale alla chiusura dell'insieme ottenuto dall'unione della proiezione delle DF di F su X con la proiezione delle DF di F su Y. Per applicare questa definizione dobbiamo saper calcolare se un insieme di dipendenze funzionali è equivalente ad un altro. Inoltre dobbiamo saper calcolare la proiezione di un insieme di dipendenze funzionali su un insieme di attributi.

Qualità delle decomposizioni Una decomposizione dovrebbe sempre garantire:

- La BCNF
- L'assenza di perdite, in modo da poter ricostruire le informazioni originarie.
- La conservazione delle dipendenze, in modo da mantenere i vincoli di integrità originari.

Esempio di verifica della BCNF Supponiamo che ogni dirigente abbia una sede e che ogni progetto possa essere diretto da più persone, ma in sedi diverse. In questo caso la chiave sarà *Progetto Sede*.

Dirigente	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Marte	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

Il progetto da solo non può determinarmi il dirigente poichè questo può essere collocato in più sedi. Risulta valido dire che

$$Progetto\ Sede \rightarrow Dirigente$$

poichè ho un solo dirigente per ogni sede di progetto. Quanto segue, tuttavia, non è una BCNF

$$Dirigente \rightarrow Sede$$

poichè i dirigenti possono trovarsi in sedi diverse (Dirigente da solo non è chiave). Si osserva che la prima dipendenza coinvolge tutti gli attributi e che quindi decomporla comporta perderla!

Morale della favola Risulta difficile trovare un BCNF che conservi le dipendenze e il JOIN insieme. Seguono ulteriori forme normali!

8.1.2 Terza forma normale (3NF)

Una relazione r è in terza forma normale (3NF) se per ogni dipendenza funzionale non banale

$$X \to Y$$

definita su r, è verificata almeno una delle seguenti condizioni:

- X è superchiave di r
- Ogni attributo in Y è contenuto in almeno una chiave di r

Dirigente	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Marte	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

Progetto Sede → Dirigente
Dirigente → Sede
L'attributo Sede è contenuto nella chiave

Si individua una ridondanza nella ripetizione della sede del dirigente per i vari progetti che dirige.

8.1.2.1 Confronto tra BCNF e 3NF

- La terza forma normale è meno restrittiva rispetto alla BCNF (ammette relazioni con alcune anomalie e ridondanze).
- Il problema di verificare se una relazione è 3NF è NP-completo (quindi il miglior algoritmo deterministico conosciuto ha complessità esponenziale nel caso peggiore).
- Tuttavia si ha un vantaggio: ottengo sempre una decomposizione senza perdite e con conservazione delle dipendenze.

8.1.2.2 Algoritmo di decomposizione

Nelle diapositive sono presentati due algoritmi di decomposizione:

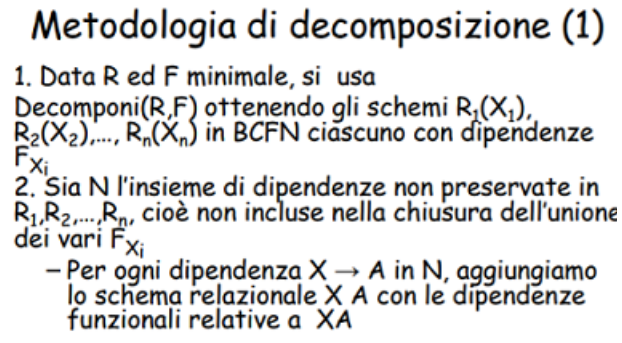
- Una che garantisce l'assenza di perdita sul JOIN e poi conserva le dipendenze
- Una che garantisce le dipendenze e poi risolve l'eventuale perdita sul JOIN

Generalmente si verifica che lo schema 3NF sia anche BCNF. Se la relazione ha una sola chiave allora le due forme normali coincidono.

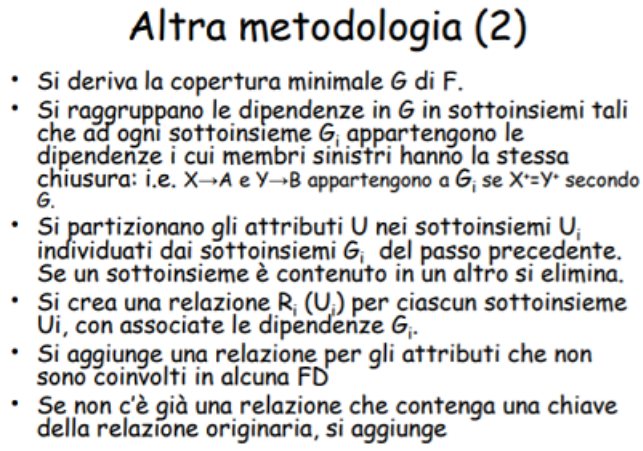
8.2 Conclusioni

Una decomposizione dovrebbe sempre garantire

- BCNF o 3NF
- L'assenza di perdite (in modo da poter ricostruire le informazioni originarie)



(a) Primo metodo, utilizzato in tutti gli esercizi della Prof



(b) Secondo metodo

- La conservazione delle dipendenze (in modo da mantenere i vincoli di integrità originari)

Il fatto che non si possa ottenere la BCNF è questione di cattiva progettazione. Tutta questa teoria serve per verificare la qualità dello schema logico: quanto spiegato può essere utilizzato anche nella progettazione concettuale per ottenere uno schema di buona qualità (verifica delle ridondanze, partizionamento di entità e relazioni....)

8.3 Normalizzazione e progetto

8.3.1 Alcune definizioni aggiuntive

- Se in uno schema di relazione c'è più di una chiave, ognuna di esse è detta chiave candidata. Una delle chiavi è detta chiave primaria
- Un attributo di R è detto attributo primo di R se è membro di una qualche chiave candidata di R. Un attributo è detto non-primo se non è membro di alcuna chiave candidata

8.3.2 Dipendenze funzionali complete e parziali

- **DF completa**: una DF $X \to Y$ si dice dipendenza funzionale completa (DFC) se la rimozione di qualsiasi attributo A da X comporta il venir meno della sussistenza della DF
- **DF parziale**: una DF $X \to Y$ si dice dipendenza funzionale parziale (DFP) se si possono rimuovere da X certi attributi A e la dipendenza continua a sussistere. Intendiamo che

$$X - \{A\} \to Y$$

8.3.3 Forme normali

Il processo di normalizzazione sottopone uno schema di relazione a una serie di test per verificare se sono soddisfatte le proprietà tipiche di una forma normale. Individuiamo le seguenti forme:

- Prima forma normale (1NF)

- Seconda forma normale (2NF)
- Terza forma normale (3NF)
- Forma normale di Boyce e Codd (BCNF)
- 4NF e 5NF (Non importante)

8.3.3.1 Prima forma normale (1NF)

La 1NF richiede che il dominio di un attributo comprenda solo valori atomici (semplici e indivisibili) e che il valore di qualsiasi attributo in una tupla sia un valore singolo del dominio. La forma normale 1NF è già parte integrante della definizione formale di relazione nel modello relazionale.

8.3.3.2 Seconda forma normale (2NF)

Uno schema di relazione R è in forma 2NF se ogni attributo non-primo A di R è funzionalmente dipendente in modo completo da ogni chiave di R. Si osserva che possono esistere dipendenze tra attributi non-primi.

8.3.3.3 Confronto tra forme normali

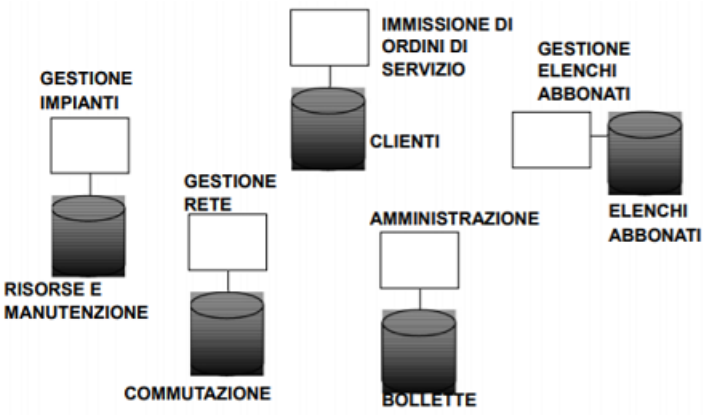
La BCNF implica la 3NF[1], che implica la 2NF. La 4NF e la 5NF riguardano dipendenza di tipo diverso, multivalore.

[1]Se tutte le volte X in $X \to Y$ è superchiave allora soddisfo tutte le volte una delle due condizioni possibili dalla 3NF

9 — Mercoledì 06/05/2020

9.1 Gestione delle transazioni

Noi abbiamo sistemi di tabelle che contengono informazioni diverse: gestione impianti, immissione di ordini di servizio, amministrazione, gestione elenchi abbonati... In alcune circostanze queste tabelle necessitano di essere visualizzate insieme.

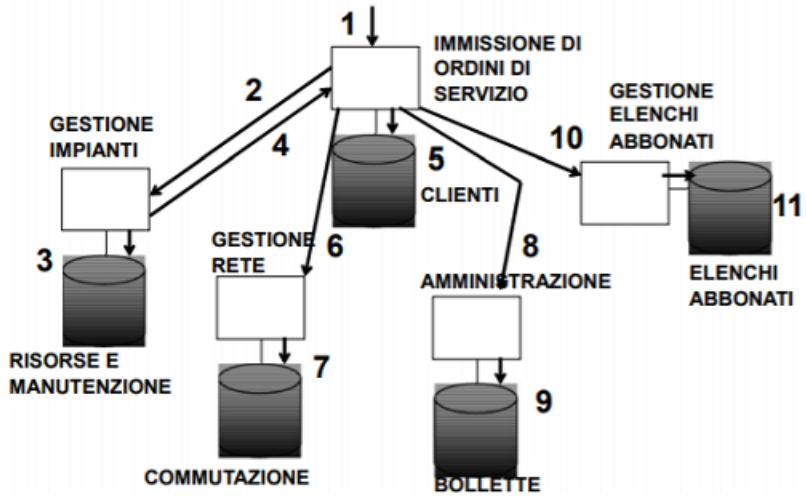


Dobbiamo tenere conto, come al solito, delle connessioni simultanee da parte di più utenti in uno stesso momento. Solitamente un DBMS è multiutente, cioè può essere usato contemporaneamente da più utenti.

Modalità interleaved o concorrente I sistemi utilizzati molti anni fa, con una sola CPU, erano già in grado di gestire un utilizzo concorrente del sistema. Un sistema solitamente è di tipo multitasking, cioè permette l'esecuzione in simultanea di più operazioni. Se si ha una sola CPU in realtà si può eseguire una sola operazione: semplicemente ci si limita ad eseguire un'operazione, a sospenderla temporaneamente per portarne avanti un'altra, e così via. Un'esecuzione dei processi può risultare, pertanto, concorrente o alternata (*interleaved*).

Definizione di transazione Le transazione non è un'operazione ma un insieme di comandi base: un'unità elementare di lavoro. A questa sequenza voglio associare caratteristiche quali la robustezza, la correttezza e l'isolamento (cioè non posso vedere l'interno della transazione e l'esito fino a conclusione). La cosa può essere immaginata anche nell'ambito MySQL con inserimenti, cancellazioni, modifiche o interrogazioni. I sistemi attualmente utilizzati sono transazionali: eseguono transazioni per conto di un certo numero di applicazioni concorrenti. Una transazione

coinvolge generalmente più tabelle con le sue operazioni:



Si inizia e si termina con i seguenti comandi, rispettivamente

```
begin transaction
end transaction
```

Uno dei due comandi deve essere eseguito una e una sola volta all'interno della transazione:

```
commit work
rollback work
```

Col primo eseguiamo la transazione con tutti i suoi comandi, col secondo blocchiamo tutto e non eseguiamo niente (è come se non avessi eseguito nessuna delle operazioni presenti nella transazione).

Proprietà delle transazioni (*ACID*)

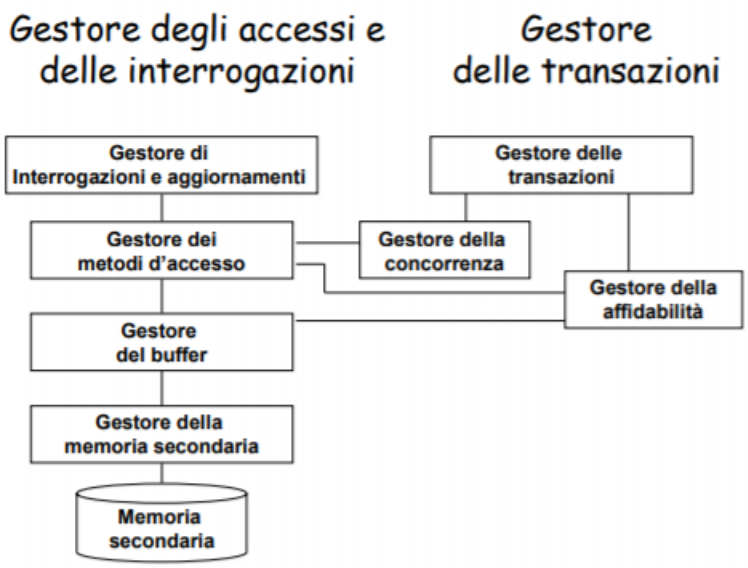
- **Atomicità**. Gli stati intermedi della transazione non si vedono: vedono lo stato iniziale corretto e lo stato finale, che dipendono dal comando eseguito (commit o rollback). Nel caso di errore dobbiamo:
  - annullare le operazioni svolte se ci troviamo prima del commit
  - ripetere le operazioni già fatte se ci troviamo dopo il commit

  Normalmente l'esito è il commit. L'abort può essere richiesto dall'applicazione (suicidio) o dal sistema (omicidio)
- **Consistenza**. Devono essere rispettati i vincoli di integrità: ciò avviene se sono corretti lo stato iniziale e quello finale. Nel corso della transazione posso avere incosistenze, l'importante è che non ci siano alla fine.
- **Isolamento**. Gli stati intermedi non vengono resi visibili agli altri processi. L'esecuzione concorrente di più transazioni deve produrre un risultato ottenibile mediante esecuzione sequenziale.
- **Durata**. Quanto fatto da una transazione andata in commit non viene perso. I guasti non hanno effetto sul sistema poichè si interviene per recuperare uno stato consistente.

Stati di una transazione

- **active**. Dopo il begin-transaction posso eseguire operazioni di lettura e scrittura.
- **partially committed**. Lo stato viene raggiunto dopo averr eseguito l'ultima operazione. Il gestore dell'affidabilità si assicura che le operazioni possano essere eseguite senza problemi
- **committed**. Si passa a questo stato se i controlli precedenti danno esito positivo.
- **failed**. Lo stato viene raggiunto se la verifica precedente fallisce
- **aborted**. Si ha un rollback: la base di dati mantiene consistenza e le operazioni non vengono eseguite.

Gestori L'atomicità e la durabilità sono garantiti dal *gestore dell'affidabilità*, l'isolamento dal *gestore della concorrenza*, la consistenza dal *gestore dell'integrità a tempo di esecuzione*.



Gerarchia di memoria All'interno della memoria distinguiamo due aree di memoria:

- la **memoria principale**, volatile. I dati vengono persi quando si spenge il dispositivo
- la **memoria secondaria**, permanente. I dati permangono rispettando una delle proprietà tipiche della base di dati (la persistenza). In questa area si troverà la base di dati vera e propria, che deve sopravvivere alle esecuzioni dei programmi.

Queste sono organizzate in modo diverso: la memoria secondaria è organizzata in blocchi di lunghezza generalmente fissa. Le uniche operazioni possibili sono lettura e scrittura dei dati di un blocco. La memoria principale, invece, è organizzata in pagine.

Buffer Il buffer (struttura fisica) permette l'interfacciamento del database, presente in memoria secondaria, con la memoria principale. Si trova anch'esso nella memoria principale ed è un elemento intermedio: ciò che deve essere letto dal computer viene spostato dalla memoria secondaria al buffer. Segue un numero di accessi estremamente limitato alla memoria secondaria (operazione più costosa).

- Il buffer è gestito dal DBMS
- Risulta organizzato in pagine (come già detto si trova in memoria principale)
- Considerando l'equivalenza pagina-blocco il caricamento di una pagina del buffer richiede una lettura in memoria secondaria, mentre il salvataggio richiede un'operazione di scrittura.

Il gestore del buffer:

- riceve richieste di lettura e scrittura di pagine dalle transazioni
- le esegue accedendo alla memoria secondaria solo quando indispensabile
- le pagine vengono mantenute finchè il buffer non è pieno e non possono essere inserite altre pagine

Località dei dati La probabilità di dover riutilizzare i dati attualmente in uso è molto alta. La *legge di Pareto* afferma che l'80% delle operazioni utilizza sempre lo stesso 20% dei dati!

Primitive di interfacciamento Il buffer funge da interfaccia ed esegue una serie di primitive.

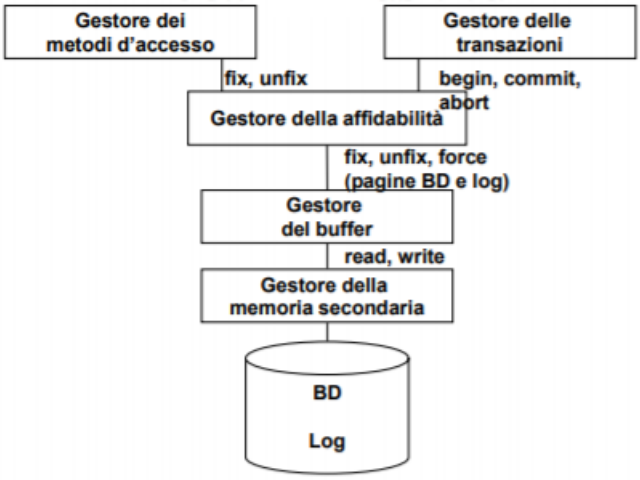
- **fix**. Richiesta di una pagina, lettura solo se la pagina non è nel buffer
- **setDirty**. Comunica che la pagina è stata modificata
- **unfix**. Comunica che la transazione ha concluso l'utilizzo della pagina (e quindi la stessa può essere richiesta da altre transazioni).
- **Force**. Trasferisce una pagina nella memoria secondaria su richiesta del *gestore dell'affidabilità*.

9.2 Gestione dell'affidabilità

Il gestore dell'affidabilità viene messo in funzione in presenza di malfunzionamenti, solitamente dovuti a

- **errori software** che portano a risultati scorretti
- **malfunzionamenti fisici**, come per esempio la mancanza di corrente in assenza di un generatore o un errore durante il trasferimento di dati (malfunzionamento del disco). I dati presenti nella memoria principale non sono più affidabili, quelli nella memoria secondaria sì.

Dobbiamo tener conto che i risultati intermedi sono salvati nel buffer, appartenente alla memoria principale. Se viene meno l'alimentazione i dati del buffer saranno persi (segue una certa procedura di recupero). In altre circostanze potrei avere anche errori legati alla memoria secondaria!



- Garantisce atomicità e durabilità.
- Prende in considerazione il begin/commit/abort
- Collabora col gestore del buffer in modo che le informazioni vengano poste in memoria secondaria.
- Si garantisce che l'esecuzione dei comandi transazionali e le operazioni di ripristino siano sempre eseguite garantendo la consistenza della base di dati
- Archivia le operazioni svolte in log (strumento necessario per fare quanto detto prima): esso consiste in un file sequenziale gestito dal controllore e scritto in memoria stabile. Le operazioni sono riportate in ordine e se ne hanno più copie in modo da non perderli in caso di disastri naturali.

Tipi di memoria Le memorie, ai fini del ripristino, vengono classificate in volatili, non volatili e stabili.

Memoria stabile La memoria stabile ospita i log. Non può danneggiarsi e non esiste in natura (astrazione). La sua "esistenza" è perseguita attraverso le ridondanze (un numero elevatissimo di copie tra vari dischi e oggetti).

Log Il log è un file sequenziale, gestito dal controllore dell'affidabilità e posto in memoria stabile. Può essere immaginato come un diario di bordo: in esso vengono salvate le operazioni delle transazioni e i record di sistema

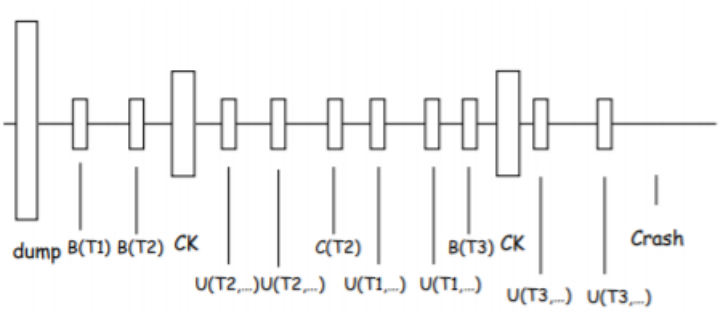
- **Operazioni delle transazioni** (attività svolte dalle transazioni):
  - begin $B(T)$

  - insert $I(T, O, AS)$
  - delete $D(T, O, BS)$
  - update $U(T, O, BS, AS)$
  - commit $C(T)$
  - abort $A(T)$

  dove T è l'identificativo della transazione, BS lo stato prima della modifica, AS lo stato dopo la modifica, O l'oggetto coinvolto nell'operazione.

- **Record di sistema** (scritti dal controllore dell'affidabilità):
  - dump
  - checkpoint

Struttura del log



- **dump**: copia di un database in un certo momento (supponiamo, per esempio, che venga fatta una copia ogni 12 ore).
- **checkpoint**: visione dello stato delle transazioni. Permette una ricostruzione più agevole delle operazioni eseguite in modo da evitare, in caso di errori, una ripartenza da zero.

Operazione Checkpoint Operazione che ha lo scopo di registrare quali transazioni sono attive in un certo istante, cioè le transazioni *a metà strada*. In contemporanea vogliamo essere certi che le altre operazioni non siano iniziate o siano finite. Cosa faccio ogni volta che eseguo questa operazione?

- Sospendo l'accettazione di operazioni di commit/abort
- Si forza la scrittura in memoria di massa delle pagine di buffer modificate da transazioni in commit
- Si forza la scrittura nel log di record
- Si riprende ad accettare tutte le operazioni

Tutto questo permette di garantire la persistenza delle transazioni che hanno eseguito il commit.

Dump Il dump consiste in una copia completa della base di dati. Solitamente viene prodotta mentre il sistema non è operativo e salvata in memoria stabile come backup.

Operazioni di undo e redo

- Undo di una azione su un oggetto O:
  - update, delete: copiare il valore del before state nell'oggetto O
  - insert: eliminare O
- Redo di una azione su un oggetto O:
  - insert,update: copiare il valore dell'after state nell'oggetto O
  - delete: eliminare O

Idempotenza di undo e redo Posso compiere le operazioni più volte, il risultato rimane sempre lo stesso:

- $undo(undo(A)) = undo(A)$
- $redo(redo(A)) = redo(A)$

Quando il controllore dell'affidabilità può consentire la modifica del log da parte delle transazioni Possiamo usare una delle seguenti regole:

- **Regola Write-Ahead-Log**: letteralmente *scrivi il log per primo*, impongo che il BS dei record del log venga scritto prima di effettuare la corrispondente operazione sul database.
- **Regola Commit-Precedenza**: si scrive la parte AS dei record di log prima di effettuare il commit.

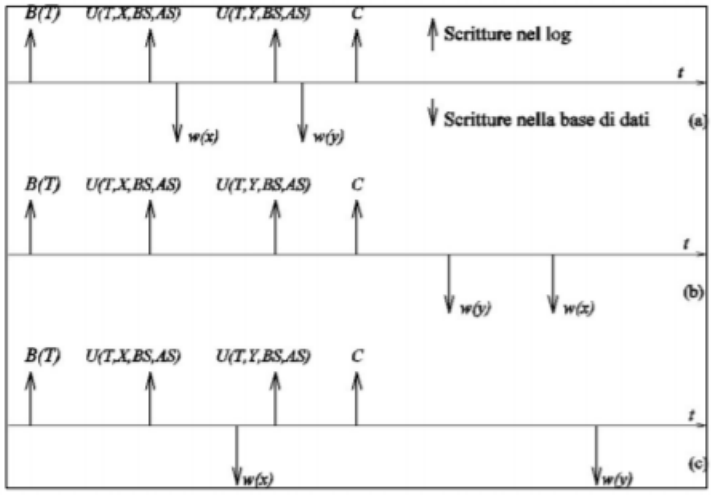
Nella pratica entrambe le componenti vengono scritte nel record in contemporanea. In modo semplificato possiamo dire:

- **Regola W-A-L**: scrivere i record di log prima delle corrispondenti operazioni nella base di dati (dopo il commit)
- **Regola C-P**: scrivere i record di log prima dell'operazione di commit

Esito di una transazione L'esito di una transazione è determinato in modo irrevocabile quando si scrive il record di commit del log. Segue un comportamento diverso in base alla posizione di un guasto:

- se il guasto avviene prima del commit eseguo una serie di *undo* per riportare la base di dati allo stato originale
- se il guasto avviene dopo il commit eseguo una serie di *redo* per ricostruire lo stato finale della base di dati. In questo caso non devo avere conseguenze.

Scrittura nel log e nella base di dati Nella base di dati posso scrivere indipendentemente dal log. Ogni volta che faccio la modifica scrivo il log e poi effettuo l'operazione, oppure effettuo l'operazione e scrivo nel log dopo il commit. Sono possibili vie di mezzo.



- **Modalità immediata**: il database contiene valori AS provenienti da transazioni uncommited. In caso di guasto devo fare undo, non ho bisogno di fare redo.
- **Modalità differita**: il database non contiene valori AS provenienti da transazioni uncommited. In caso di aborto non bisogna fare niente. L'undo è superfluo, poichè non si hanno operazioni di scrittura prima del commit. In caso di guasto rieseguo le operazioni.
- **Modalità mista**: la scrittura avviene in modalità sia immediata che differita, ricorrendo se necessario sia all'undo che al redo.

La modalità differita non è molto utilizzata pur permettendo una procedura di recovery più semplice ed efficiente: questo perchè il gestore del buffer non è libero di decidere quando scrivere in memoria secondaria. Risulta preferibile una gestione ordinaria più efficiente rispetto a una più semplice dei guasti, considerando la rarità di questi.

In conclusione

- La scrittura nella base di dati può avvenire in qualunque momento, anche prima del commit
- La scrittura sul log è effettuata prima della scrittura nella base di dati
- Il commit si considera effettuato quando il corrispondente record di log è scritto. Prima di questa scrittura il guasto causa l'undo; dopo il guasto causa il redo.

9.2.1 Rollback di una transazione

Quando una transazione deve essere cancellata tutte le operazioni tra il begin della transazione e l'abort devono essere disfatte. Sarà scritto un record di abort nel log. Distinguiamo i seguenti tipi di guasti

- **Guasti soft**: errori di programma, crash di sistema, caduta di tensione. Si perde la memoria centrale ma non quella secondaria (la base di dati e il log).
  In questo caso avremo un *warm restart*, cioè una **ripresa a caldo**
- **Guasti hard**: si perde anche la memoria secondaria, cioè parte della base dei dati. Ovviamente non si perde la memoria stabile contenente i log.
  In questo caso avremo un *cold restart*, cioè una **ripresa a freddo**

La perdita dei *log* o dei *dump* è considerato un evento catastrofico, uno dei più improbabili, e quindi non è definita alcuna strategia di recupero.

Cosa fa il gestore in caso di guasti? Il modello adottato è quello *fail-stop*

- Si forza l'arresto completo delle transazioni
- Il sistema operativo viene riavviato
- **Viene avviata una procedura di restart** (che sarà a caldo o a freddo)
- Al termine del restart il buffer è vuoto, ma le transazioni possono ripartire.

Considero la seguente classificazione di transazioni nel processo di restart:

- Transazioni completate (i dati sono in memoria e non dobbiamo fare niente)
- Transazioni attive ma con commit (da rifare, *redo*)
- Transazioni attive ma senza commit (da annullare, *undo*)

9.2.1.1 Processo di restart a caldo

Fasi

1. Trovare l'ultimo checkpoint (ripercorrendo il log a ritroso)
2. Costruire gli insiemi UNDO e REDO, cioè individuare le transazione da disfare e quelle da rifare.
   - Inizializzo UNDO con le transazioni attive al checkpoint. REDO inizialmente è vuoto.
   - Aggiungo all'insieme di UNDO tutte le transazioni che presentano un record di begin e sposto in REDO tutti gli identificativi delle transazioni in cui è presente record di commit.
   - Al termine i due insiemi conterranno, appunto, gli identificativi delle transazioni da disfare e di quelle da rifare
3. Ripercorrere il log all'indietro fino alla più vecchia azione delle transazioni in UNDO e REDO, disfacendo tutte le azioni delle transazioni in UNDO.
4. Ripercorrere il log in avanti, rifacendo tutte le azioni delle transazioni in REDO.

Esempio 1 Considerando il seguente log di un sistema di gestione di basi di dati, illustrare dettagliatamente i passi da compiere per effettuare la ripresa a caldo.

Dump, B(T1), B(T2), I(T1,O1,A1), D(T2,O2,B2), B(T3), B(T4), U(T3,O3,B3,A3), C(T2), CK(...), U(T1,O4,B4,A4), A(T3), B(T5), D(T4,O5,B5), C(T1), C(T4), I(T5,O6,A6), GUASTO

- Le transazioni attive (cioè quelle che scrivo nel checkpoint) sono T1,T3 e T4.
- T2 è già terminata, quindi ci interessa poco
- Realizzo i due insiemi di *UNDO* e *REDO*

RECORD	UNDO	REDO
CK(T1,T3,T4)	T1, T3, T4	
B(T5)	T1, T3, T4, T5	
C(T1)	T3, T4, T5	T1
C(T4)	T3, T5	T1, T4

- Le transazioni da disfare sono T3 e T5, mentre quelle da rifare sono T1 e T4. Riempo la tabella delle UNDO partendo dal fondo e considero le operazioni di T3 e T5.

RECORD	AZIONE
I(T5,O6,A6)	delete O6
U(T3,O3,B3,A3)	O3 := B3

- Si riempe la tabella delle REDO partendo dall'inizio e considerando le operazioni di T1 e T4

RECORD	AZIONE
I(T1,O1,A1)	insert O1 := A1
U(T1,O4,B4,A4)	O4 := A4
D(T4,O5,B5)	delete O5

Domanda Le transazione attive da inserire nel checkpoint si possono fissare:

- Rifiutando nuovi commit
- Rifiutando nuove transazioni e aspettando la conclusione di quelle già iniziate

Quale sarà la differenza nella gestione delle riprese a caldo?

- Nel primo caso si attua la strategia già vista
- Nel secondo caso il checkpoint non conterrà nessuna transazione poichè non si hanno transazioni attive. Nella ripresa a caldo mi limito a rieseguire tutte le operazioni che seguono il record di checkpoint.

- Una conseguenza è il degrado delle prestazioni poichè la base di dati viene fermata ogni volta che si deve eseguire un checkpoint.
- Inoltre questi errori sono rari.
- Segue che la prima soluzione è più conveniente: è improduttivo semplificare la gestione di questi errori (rari) se ciò comporta una gestione più complicata delle situazioni ordinarie.

Transazioni abortite Le operazioni derivanti dal rollback di una transazione possono essere inserite nell'insieme UNDO e fatte al momento del recovery, oppure essere eseguite al momento dell'abort venendo inserite nell'insieme di redo al momento del recovery da un guasto.

9.2.1.2 Ripresa a freddo

Fasi

- Ci si riporta al record di dump più recente nel log e si ripristina la parte di dati deteriorata
- Si eseguono le operazioni registrate sul log nella parte deteriorata fino all'istante del guasto
- Si esegue una ripresa a caldo

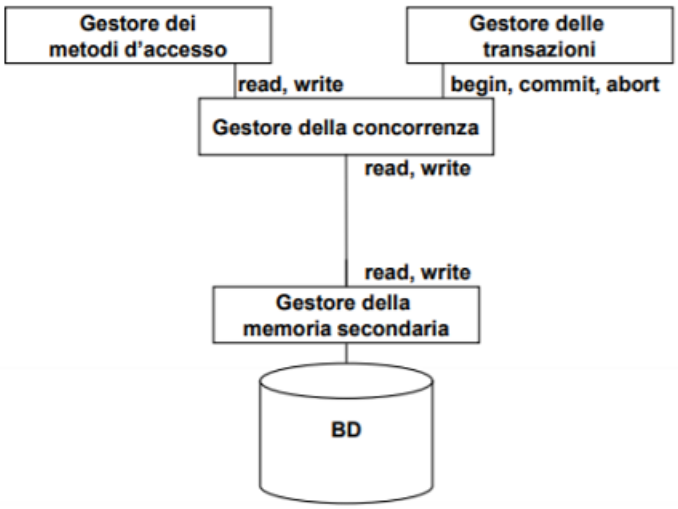
Esempio 2 Riprendio lo stesso log di prima. Considerando il seguente log di un sistema di gestione di basi di dati, illustrare dettagliatamente i passi da compiere per effettuare la ripresa a freddo dopo un guasto di dispositivo che interessa gli oggetti O1, O2, O3.

- Insert O1 = A1
- Delete(O2)
- O3=A3
- Commit(T2)
- Abort(T3)
- Commit(T1)
- Commit(T4)
- Ripresa a caldo

10 — Giovedì 07/05/2020

Le transazioni possono essere di una certa dimensione e richiedere un certo tempo per essere completate. Dobbiamo essere in grado di gestire le anomalie derivanti dall'esecuzione concorrente. Lavorare con una sola CPU significa adottare l'approccio dell'esecuzione di pezzi di transazioni alternati: questo può provocare problemi soprattutto se le due transazioni agiscono sugli stessi elementi. Segue la necessità di governare la situazione.

10.1 Gestore della concorrenza



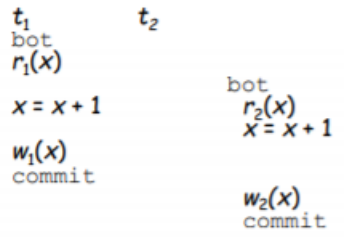
Il gestore della concorrenza si occupa di una parte estremamente complicata: sono necessari algoritmi estremamente sofisticati e complicati. Le transazioni devono andare avanti il più possibile.

10.1.1 Esempi di problemi

Perdita di aggiornamento (W-W) Utilizziamo le seguenti transazioni, identiche

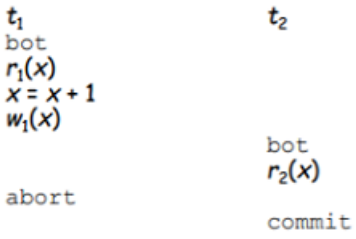
- $t1 : r(x), x = x + 1, w(x)$
- $t2 : r(x), x = x + 1, w(x)$

Inizialmente ho $x = 2$, dopo un'esecuzione seriale $x = 4$. In caso di esecuzioni concorrenti gestite in modo interfogliato viene perso un aggiornamento: otteniamo $x = 3$.

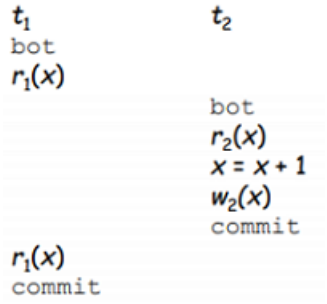


Nel modo indicato nell'immagine la lettura di x presente nella seconda transazione non è aggiornata. Pur avendo fatto commit nella prima transazione il risultato viene sovrascritto: il risultato della seconda transazione, posto dopo quello della prima, si basa su una lettura non aggiornata.

Lettura sporca (R-W o W-W con abort) La prima transazione viene abortita, tuttavia la seconda ha letto risultati intermedi della prima: con questo si intende il valore di x incrementato prima dell'abort (con l'abort l'operazione viene annullata).

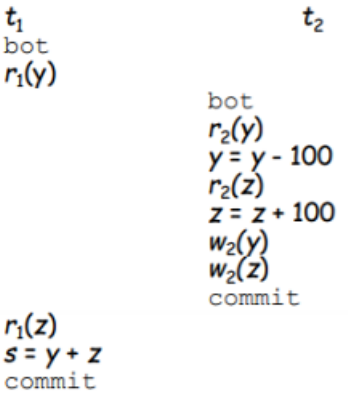


Lettura inconsistenti (R-W) Nella prima transazione si svolgono due operazioni di lettura di x. Nell'esempio qua sotto il primo x dipende dallo stato iniziale della base di dati, il secondo dipende dalla seconda transazione. Segue la lettura di due valori x diversi!



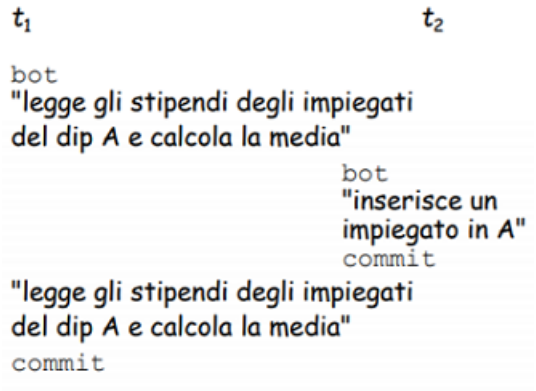
Si individuano letture diverse anche nei seguenti casi: entrambe le operazioni di lettura svolte prima del *bot* della seconda transazione; entrambe le operazioni di lettura svolte dopo il *commit* della seconda transazione.

Aggiornamento fantasma (R-W) Assumiamo ci sia un vincolo $y + z = 1000$. La prima transazione y e z. Il primo dipende dallo dato iniziale del db, il secondo dipende dalla seconda transazione.



Segue che y potrebbe dare problemi. Osserviamo che al termine della prima transazione, dopo aver sommato y e z,si ha come risultato 1100: ciò viola il vincolo stabilito.

Inserimento fantasma (R-W) Cosa analoga: prima leggo gli stipendi degli impiegati nella prima transazione e dopo, nella seconda transazione, inserisco un nuovo dipendente.



10.1.1.1 Ricapitoliamo

- **Perdita di aggiornamento (W-W)**: Abbiamo due transazioni che effettuano operazioni di scrittura su un oggetto. Le operazioni si basano su una precedente lettura. Il problema è che una delle letture non legge il dato aggiornato: segue che una delle due modifiche risulta persa!
- **Lettura sporca (R-W o W-W con abort)**: ho una transazione che legge e scrive e una che legge. La prima transazione abortisce, ma la seconda nel frattempo ha letto il valore di x modificato.
- **Letture inconsistenti (R-W)**: ho una transazione che svolge due operazioni di lettura e una che svolge un'operazione di lettura seguita da una di scrittura. In uno schedule serializzabile non è possibile che le due operazioni di lettura restituiscano valori diversi!
- **Aggiornamento fantasma (R-W)**: Abbiamo una transazione che legge dei valori aggiornati da un'altra transazione. L'interleaving potrebbe portare alla violazione di vincoli precedentemente imposti.
- **Inserimento fantasma (R-W)**: una transazione potrebbe effettuare una lettura di una serie di valori e successivamente usarli per calcolare un risultato. Tra la prima e la seconda cosa si ha l'inserimento, da parte di una seconda transazione, di un nuovo valore: questo sarà ignorato.

10.1.2 Come gestire?

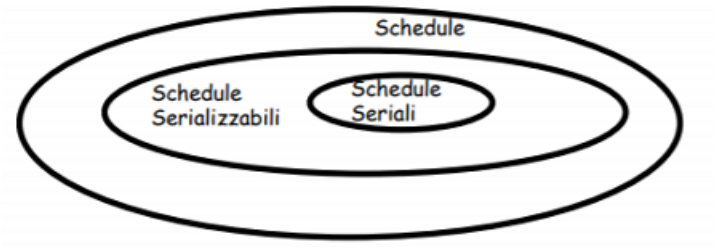
Schedule Dato che più transazioni vengono eseguite in modo concorrente le operazioni di scrittura/lettura vengono richieste da transazioni differenti in interleaving. Uno schedule consiste in una sequenza di operazioni di lettura/scrittura relativa a un insieme di transazioni concorrenti. Per fare ciò eseguiamo operazioni di *scheduling*, operazione fondamentale del controllore della concorrenza. Fondamentalmente:

- elenco delle operazioni di lettura e scrittura applicate ad oggetti;
- se la sequenza è permessa si ha come risultato finale un qualcosa ottenibile anche attraverso un'esecuzione in sequenza di istruzioni.

Controllo di concorrenza Il gestore, per evitare anomalie, determina un ordine delle richieste fatte. Continua ad accettare gli ordini finchè non si rende conto che accettando una certa operazione si avrebbe un qualcosa di assolutamente incompatibile con un comportamento seriale. Questa richiesta viene fermata e messa da parte finchè l'oggetto incriminato nell'operazione non sarà "liberato".

Tutte le sequenze di operazioni che risultano completate rappresentano degli schedule serializzabili, cioè equivalenti a qualche schedule seriale. Il comportamento seriale è corretto per definizione e non può portare problemi.

Idea di base L'obiettivo è individuare classi di schedule serializzabili la cui proprietà di serializzabilità sia verificabile a costo basso (tenendo conto che le operazioni vengono fatte online). Cioè, come posso capire se quell'esecuzione (con operazioni in un certo ordine) ci porterà a ottenere un qualcosa di serializzabile?



Distinguo schedule seriali da schedule serializzabili e infine schedule non serializzabili. Tutto ciò che è stato fatto deve essere buttato via se non è possibile ottenere un qualcosa di serializzabile. Attraverso algoritmi devono individuare classi di schedule grandi: più grandi saranno, migliore potrà essere considerato l'algoritmo.

11 — Mercoledì 13/05/2020

Abbiamo introdotto la scorsa volta il gestore della concorrenza: abbiamo visto come in certi casi questo gestore neghi temporaneamente l'accesso al buffer a una transazione, precisamente quando non si hanno atteggiamenti seriali. Tutto ciò viene gestito attraverso operazioni di *scheduling*: costruisco sequenze di operazioni R/W garantendo delle proprietà. Abbiamo, per esempio:

$$S1 : r_1(x)r_2(z)w_1(x)w_2(z)$$

Dove le operazioni con 1 sono eseguite dalla prima transazione e quelle con 2 dalla seconda transazione. Il tutto è eseguito in *interleaving*, cioè eseguiamo pezzi di transazione (nell'esempio le operazioni sono miste). Ribadiamo il ricorso ad algoritmi sofisticati!

Obiettivo Evitare le anomalie

Soluzione Scheduler. Come già detto consiste in un sistema che accetta o rifiuta, anche tramite riordino, le operazioni richieste dalle transazioni.

Schedule seriale Uno schedule si dice seriale se per ogni transazione t, tutte le azioni di t compaiono in sequenza, senza essere inframezzate da azioni di altre transazioni. Nel seguente schedule le transazioni vengono eseguite in sequenza

$$S2 : r_0(x)r_0(y)w_0(x)r_1(y)r_1(x)w_1(y)r_2(x)r_2(y)r_2(z)w_2(z)$$

Schedule serializzabile L'esecuzione di uno schedule S_i è corretta quando produce lo stesso risultato di un qualunque schedule seriale S_j definito dalle stesse transazioni di S_i. Lo schedule S_i è detto serializzabile.

11.1 Nozioni di equivalenza

11.1.1 Relazione *legge-da*

Abbiamo le transazioni i e j. Data un'operazione di lettura della prima $r_i(x)$ e un'operazione di scrittura della seconda $w_j(x)$ presenti in uno schedule S si dice che esiste una relazione (che noi chiameremo *legge-da*) se:

- l'operazione di scrittura $w_j(x)$ precede quella di lettura $r_i(x)$
- non si ha nessuna operazione di scrittura $w_k(x)$ tra di loro, con $k \neq j$.

Mi chiedo se l'operazione di lettura legge quanto ottenuto dall'operazione di scrittura o lo stato del database.

Scrittura finale Definisco $w_i(x)$ in S scrittura finale su x se è l'ultima scrittura dell'oggetto x nella schedule S.

11.1.2 Schedule view-equivalenti

Due schemi sono *view-equivalenti* se hanno:

- le stesse relazioni *legge-da*
- le stesse scritture finali su ogni oggetto

11.1.3 Schedule view-serializzabile

Uno schedule S è detto *view-serializzabile* se è *view-equivalente* ad un qualche schedule seriale. L'insieme degli schedule view-serializzabili si indica con VSR.

11.1.4 Esempi

Esempio con schedule view-serializzabili Consideriamo i seguenti schedule:

- $S3 : w_0(x)r_2(x)r_1(x)w_2(x)w_2(z)$
  - Non seriale
  - $r_2(x)$ e $r_1(x)$ leggono quanto scritto da $w_0(x)$. Dopo $w_2(x)$ non ho altre operazioni di lettura.
- $S4 : w_0(x)r_1(x)r_2(x)w_2(x)w_2(z)$
  - Seriale
  - Cambia l'ordine delle letture $r_1(x)$ e $r_2(x)$. Esse leggono quanto scritto da $w_0(x)$.
  - Lo stato finale di $S3$ è lo stesso di $S4$. Le operazioni finali di scrittura sono le stesse
- $S5 : w_0(x)r_2(x)w_2(x)r_1(x)w_2(z)$
  - Non seriale
  - Le operazioni $r_1(x)$ e $r_2(x)$ non leggono più la stessa cosa: il primo legge quanto scritto da $w_0(x)$, il secondo quanto scritto da $w_2(x)$. Individuo che $S5$ non è sicuramente view-equivalente ad $S4$. Non stiamo dicendo che non è view-serializzabile.
- $S6 : w_0(x)r_2(x)w_2(x)w_2(z)r_1(x)$
  - Seriale (non conta l'ordine delle transazioni ma se queste vengono interrotte da operazioni di altre transazioni per poi essere riprese)
  - Confrontiamo con lo schedule precedente: le operazioni $r_2(x)$ e $r_1(x)$ leggono rispettivamente da $w_0(x)$ e $w_2(x)$ (come prima).
  - Le operazioni di scrittura finale in $S5$ ed $S6$ sono le stesse: $w_2(x)$ e $w_2(z)$.

Cosa possiamo dire?

- La $S3$ è *view-equivalente* allo schedule seriale $S4$ (quindi è view-serializzabile)
- $S5$ non è *view-equivalente* allo schedule seriale $S4$, ma lo è rispetto allo schedule seriale $S6$ (quindi è view-serializzabile).
- Eseguo transazioni in modo diverso ma il risultato finale è lo stesso.

Esempi con schedule non view-serializzabili

- **Perdita di aggiornamento**: $S3 : r_1(x)r_2(x)w_1(x)w_2(x)$
  - Sia $r_1(x)$ che $r_2(x)$ leggono lo stato del database
  - Entrambe le transazioni leggono e scrivono lo stesso variabile.
  - Qualunque ordine io faccia entrambe le transazioni leggono lo stesso valore di x: questa cosa in una situazione normale non è possibile. Per forza una delle due deve leggere il risultato dell'altra.
  - Segue che questo schedule non è serializzabile.
- **Letture inconsistenti**: $S4 : r_1(x)r_2(x)w_2(x)r_1(x)$
  - Ho una transazione che compie due operazioni di lettura: $r_1(x)$ due volte. L'altra transazione compie un'operazione di lettura $r_2(x)$ e una di scrittura $w_2(x)$
  - L'unico ordinamento seriale con cui mi posso confrontare è quello in cui ottengo prima la transazione 1 e poi la 2. Quanto detto prima non può essere ricondotto a uno schedule seriale avente lo stesso risultato.
- **Aggiornamento fantasma**: $S5 : r_1(x)r_1(y)r_2(z)r_2(y)w_2(y)w_2(z)r_1(z)$
  - La transazione 1 legge x, legge y, legge z. La seconda transazione legge z, legge y, scrive y e scrive z.
  - Anche in questo caso non possiamo ricondurre lo schedule a un equivalente seriale con la stessa soluzione.

Non sono *view-serializzabili*, quindi non sono *view-equivalenti* a nessun schedule seriale.

11.1.5 Verifica della view-serializzabilità

Il gestore controlla se aggiungendo una certa operazione allo schedule si continua a rispettare i vincoli di ordinamento.

- La complessità della verifica della view-equivalenza è polinomiale poichè devo scorrere tutto lo schedule.
- Decidere la view-serializzabilità di uno schedule è un problema NP-Completo esponenziale!
- Segue che l'algoritmo di verifica è troppo complicato.

Come risolviamo? Definiamo una condizione di equivalenza più ristretta, che non copra tutti i casi di equivalenza tra schedule coperti della view-equivalenza. Questa dovrà essere utilizzabile nella pratica, quindi avere una complessità inferiore.

11.2 Conflitti tra operazioni

Già con la view-equivalenza abbiamo visto un conflitto, precisamente tra una scrittura e una lettura successiva. Adesso vediamo anche altri tipi di conflitti tra due operazioni:

- Una lettura seguita da una scrittura (conflitto *read-write*)
- Una scrittura seguita da una lettura (conflitto *read-write*)
- Una scrittura seguita da un'altra scrittura (conflitto *write-write*)

Definizione semplice di conflitto Un'operazione a_i è in conflitto con un'altra a_j con $i \neq j$ se operano sullo stesso oggetto e almeno una di esse è una scrittura (nei casi precedenti c'è sempre un'operazione di scrittura).

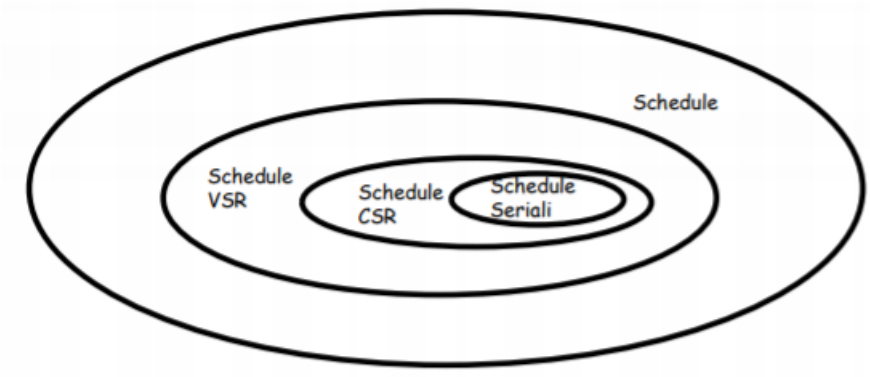
Conflict-equivalenza Due schedule sono *conflict-equivalenti* se

- includono le stesse operazioni
- ogni coppia di operazioni in conflitto compaiono nello stesso ordine in entrambi gli schedule

Conflict-serializzabilità Uno schedule è *conflict-serializzabile* se è conflict-equivalente ad un qualche schedule seriale.

Confronto con la classe di schedule vista prima Ricordiamo quanto detto prima parlando di *verifica della view-serializzabilità*. Risulta dimostrabile che questa classe di schedule è strettamente contenuta nella classe degli schedule precedenti. Cosa significa tutto ciò?

- Che ogni schedule conflict-serializzabile è anche view-serializzabile (CSR implica VSR)
- Che esistono schedule appartenenti a VSR che non appartengono a CSR.



11.2.1 Verifica di conflict-serializzabilità

La verifica avviene attraverso il *grafo dei conflitti*.

- è orientato

- ogni nodo rappresenta una transazione
- un arco orientato va da t_i a t_j se c'è almeno un conflitto fra un'operazione a_i e un'operazione a_j tale che a_i precede a_j.

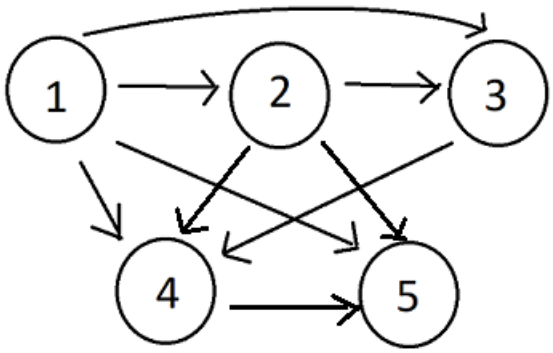
Teorema Uno schedule è in CSR se e solo se il grafico è aciclico!

Esempio di applicazione del teorema Abbiamo la seguente schedule

- S= r1(x)w2(x)r3(x)r1(y)w2(y)r1(v)w3(v)r4(v)w4(y)w5(y)

x	y	v
r1	r1	r1
w2	w2	w3
r3	w4	r4
	w5	

Con le informazioni a disposizione disegno il seguente grafo

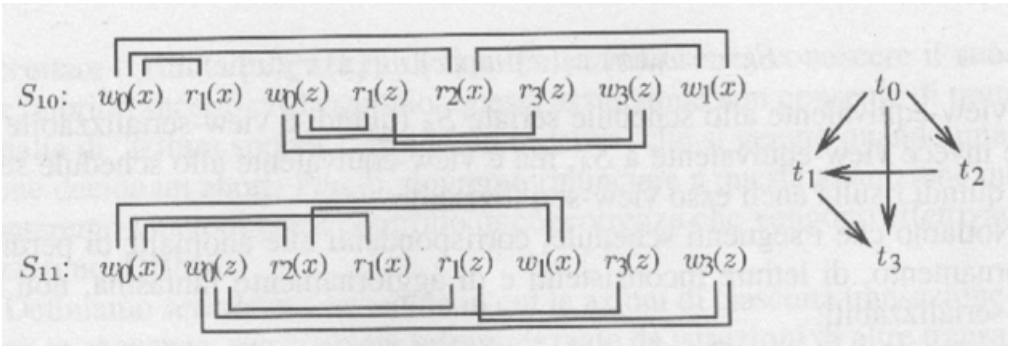


- Ho cinque transazioni, quindi disegno cinque nodi.
- Su ogni oggetto guardo l'ordine delle transazioni:
- in x:
  - Ho $r1$ e $w2$, quindi conflitto che va dal nodo 1 al nodo 2.
  - Non ho conflitto tra nodo 1 e nodo 3 (due operazioni di lettura)
  - Ho $w2$ ed $r3$, quindi conflitto che va dal nodo 2 al nodo 3
- in y:
  - Ho $r1$ e $w2$, conflitto già visto
  - Ho $r1$ e $w4$, quindi conflitto da nodo 1 a nodo 4
  - Ho $r1$ e $w5$, quindi conflitto da nodo 1 a nodo 5
  - Ho $w2$ e $w4$, quindi conflitto da nodo 2 a nodo 4
  - Ho $w2$ e $w5$, quindi conflitto da nodo 2 a nodo 5
  - Ho $w4$ e $w5$, quindi conflitto da nodo 4 a nodo 5

- in v:
  - Ho $r1$ e $w3$, quindi conflitto da nodo 1 a nodo 3
  - Ho $w3$ e $r4$, quindi conflitto da nodo 3 a nodo 4.

Osservo che il grafo è aciclico. Segue che la schedule è conflict-serializzabile!

Esempio 2 Ulteriore esempio, preso dall'Atzeni, di conflict-equivalenza e di grafo dei conflitti



Altri esempi Vedere le diapositive della professoressa.

Succo del discorso La conflict-serializzabilità è più rapidamente verificabile. Tuttavia necessità della costruzione, ogni volta, di un grafo dei conflitti (per ogni richiesta di scrittura).

Esempio di situazione problematica Supponiamo di avere 100 transazioni al secondo, ciascuna accede a 10 pagine per la durata di 5 secondi. Ogni istante devo gestire grafi con 500 nodi tenendo traccia di oltre 5000 accessi delle 500 transazioni atttive. Il grafo si modifica dinamicamente tutte le volte e rende difficoltoso per lo scheduler prendere decisioni. Non è praticabile verificare l'aciclicità ad ogni richiesta di operazione.

Cosa facciamo? Ricerchiamo un nuovo metodo di accettazione delle richieste (scheduling) che mi garantisca a priori la conflict-serializzabilità (senza costruire un grafo tutte le volte).

11.3 Lock, unlock e lock manager

Alla base di questo nuovo metodo abbiamo il *lock* e l'*unlock*.

- Ogni operazione di lettura è preceduta da *lock* e seguita da *unlock*.
- Ogni operazione di scrittura è preceduta da *lock* e seguita da *unlock*.

Il *lock manager* è parte dello scheduler: accetta le richieste di lock e unlock. Applicare il *lock* a una variabile significa renderla esclusivamente nostra, di una transazione in particolare. Dopo l'*unlock* la variabile potrà essere utilizzata da altre transazioni.

Tipi di lock Si distinguono due tipi di lock:

- **lock condiviso**, per operazioni di lettura (r_lock)
- **lock esclusivo**, per operazioni di scrittura (w_lock)

Questi tipi di lock diversi, usati in momenti diversi sulla stessa risorsa, mi permettono di aumentare la concorrenza. Nel caso in cui una transazione prima legga e poi scriva un oggetto può:

- richiedere subito un lock esclusivo, oppure
- chiedere prima un lock condiviso e poi uno esclusivo (*lock upgrade*)

11.3.1 Comportamento dello scheduler

Lo schedule agisce sfruttando la *tavola dei conflitti*: conosce lo stato delle risorse e in base allo stesso decide se concedere o non concedere il *lock* a una transazione.

- se un lock viene concesso si dice che la risorsa è *acquisita* dalla transazione
- nel momento dell'unlock la transazione *rilascia* la risorsa.

Richiesta	Stato della risorsa		
	free	*r_locked*	*w_locked*
r_lock	OK / r_locked	OK / r_locked	NO/ w_locked
w_lock	OK / w_locked	NO / r_locked	NO / w_locked
unlock	error	OK / depends (1)	OK / free

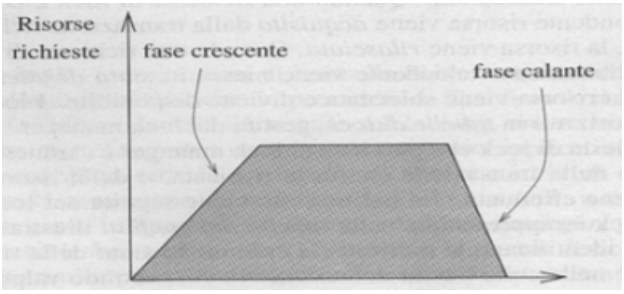
(1) Un contatore tiene conto del numero di "lettori"; la risorsa è rilasciata solo quando il contatore scende a zero

Cosa succede se il lock viene negato? La transazione viene messa da parte, in coda. Quando la risorsa è libera inizio a far scorrere la coda permettendo il lock!

11.3.2 Ordine di lock e unlock

In questo momento stiamo garantendo solo la mutua esclusione sulla risorsa ma non la serializzabilità. Possiamo determinare l'ordine delle richieste di acquisizione e rilascio in modo da ottenere degli schedule CSR?

Locking a due fasi Questo meccanismo mi permette di garantire la CSR a priori. La transazione, dopo aver rilasciato un lock, non può acquisirne altri finchè tutti quelli che ha acquisito non sono stati rilasciati! *Accumulo, e poi rilascio senza acquisire altro.*



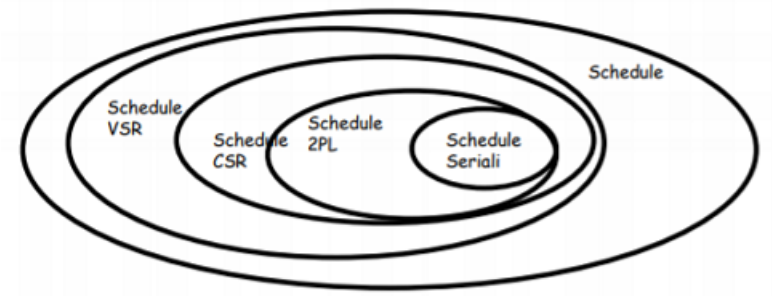
Confrontiamo due schedule con le stesse operazioni: quello a destra non rispetta la 2PL, l'altro sì.

• begin (T1)	• begin (T1)
• w1_lock(B);	• w1_lock(B);
• r1(B);	• r1(B);
• B:=B-50;	• B:=B-50;
• w1(B);	• w1(B);
• w1_lock(A);	• unlock(B);
• r1(A);	• w1_lock(A);
• unlock(B);	• r1(A);
• A:=A+50;	• A:=A+50;
• w1(A);	• w1(A);
• unlock(A);	• unlock(A);
• commit	• commit

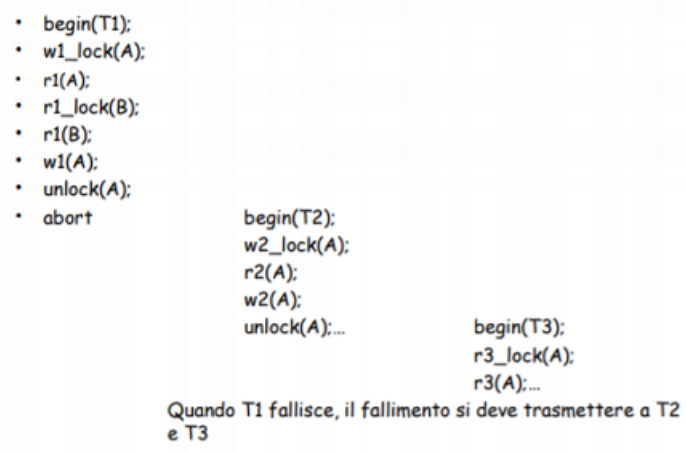
Nell'esempio in 2PL inizio a fare unlock quando ho già fatto lock su tutti gli oggetti che mi servivano: prima di lavorare su B faccio unlock su A, poichè ho già finito con quell'oggetto. Lavoro su B e faccio unlock su B. Finchè non ho liberato B non potrò richiedere altri lock!

Upgrade e downgrade L'upgrade può essere fatto solo nella fase di acquisizione, il downgrade nella fase di rilascio.

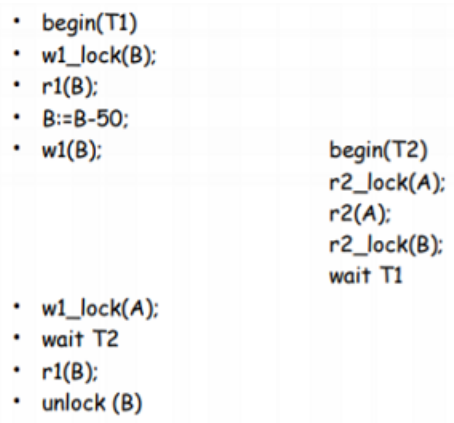
2PL e CSR Esistono schedule in CSR, ma non in 2PL.



Anomalie Essendo in 2PL si risolvono anomalie di perdita di aggiornamento, di aggiornamento fantasma, e letture inconsistenti. Rimangono però delle anomalie, in particolare quella della *lettura sporca*. Una transazione che a un certo punto fa abort va annullata: i valori intermedi potrebbero essere stati usati da altre transazioni. A questo punto dobbiamo dichiarare il fallimento di tutte le transazioni che hanno letto il valore scritto.



Altro problema sono le attese incrociate (o *deadlock*): due transazioni detengono ciascuna una risorsa e aspettano la risorsa detenuta dall'altra. In generale la probabilità di deadlock è bassa (inferiore a quella che si manifesti un conflitto) ma non nulla.



12 — Giovedì 14/05/2020

12.1 Locking a due fasi stretto (o rigoroso)

Ci siamo lasciati con due anomalie ancora da risolvere: le letture sporche e le *deadlock*. Per risolvere il primo problema si ricorre a un locking *a due fasi stretto*. Si impone un'ulteriore condizione:

- I lock possono essere rilasciati solo dopo aver fatto commit.

In questo modo gli stati intermedi non saranno visibili finchè non si avrà il commit! A quel punto siamo certi che le transazioni non saranno annullate.

Confronto tra schedule no 2PL (a destra) e schedule con 2PL stretto (a sinistra)

12.2 Alternativa: controllo di concorrenza basato su timestamp

Questa è una tecnica alternativa al 2PL. Ogni transazione è identificata in modo univoco attraverso un *timestamp*. Ci baseremo sull'ordine con cui sono state attivate le transazioni

Accettazione degli schedule Lo schedule deve essere equivalente allo schedule che riflette l'ordinamento delle transazioni (indotto dai timestamp).

Esempio Immaginiamo di avere le transazioni

$$T_1, T_2, T_3$$

attivate nell'ordine indicato (immaginiamo che gli indici identificativi siano i timestamp). Gli unici ordinamenti seriali accettabili sono quelli compatibili con l'ordine seriale delle transazioni. Segue che uno schedule equivalente a questo

$$T_2, T_1, T_3$$

non va bene.

Dettagli

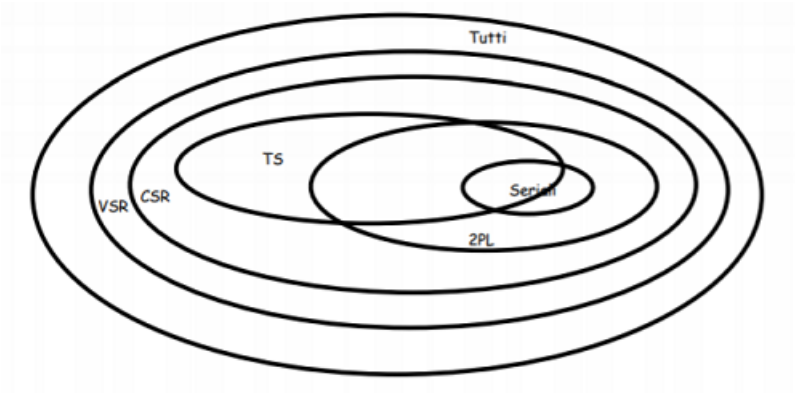
- Tutti gli oggetti coinvolti dallo scheduler presentano due contatori:
  - uno per le letture $RTM(x)$: consiste nel timestamp massimo tra quelli associati alle transazioni che hanno letto l'oggetto x
  - uno per le scritture $WTM(x)$: consiste nel timestamp dell'ultima transazione che ha scritto sull'oggetto x
- Lo scheduler riceve richieste di letture e scritture (assieme al timestamp della transazione):
  - lettura *read(x,ts)*:
    - se $\boxed{ts < WTM(x)}$ la richiesta è respinta e la transazione abortita
    - negli altri casi accolgo la richiesta ponendo $\boxed{RTM(x) = \max(RTM(x), ts)}$
  - scrittura *write(x,ts)*:
    - Se $\boxed{ts < RTM(x)}$ o $\boxed{ts < WTM(x)}$ la richiesta è respinta e la transazione abortita
    - negli altri casi la richiesta viene accolta ponendo $\boxed{WTM(x) = ts}$

Esempio

	Risposta	Nuovo Valore
read(x,1)	Ok	RTM(x)=1
write(x,1)	Ok	WTM(x)=1
read(z,2)	Ok	RTM(z)=2
read(y,1)	Ok	RTM(y)=1
write(y,1)	Ok	WTM(y)=1
read(x,2)	Ok	RTM(x)=2
write(z,2)	Ok	WTM(z)=2

12.2.1 2PL e TS

Gli schedule TS sono automaticamente CSR: corrispondono ad una esecuzione seriale (quella in cui le transazioni sono eseguite nell'ordine in cui sono iniziate). Tuttavia 2PL e TS sono incomparabili!



Attenzione L'ordine seriale delle transazioni è fissato prima che le operazioni vengano richieste. Altri ordinamenti non sono accettati.

Situazione Quando $T1$ comincia prima di $T2$ potrebbe essere abilitato uno schedule 2PL o CSR equivalente ad uno seriale $T2$ $T1$. Col TS ciò non è possibile, al limite uccido la $T1$ per poi farla ripartire dopo $T2$.

Quindi

- In $2PL$ la transazioni sono poste in attesa quando il lock non può essere acquisito. In TS vengono uccise e rilanciate.
- il $2PL$ conviene maggiormente poichè la ripartenze in TS sono più costose delle attese.
- Tuttavia il $2PL$ può causare *deadlock*. Mediamente si uccide una transazione ogni due conflitti, ma la probabilità di insorgenza di deadlock è molto minore della probabilità di un conflitto. Segue che il $2PL$ rimane la scelta migliore.

12.2.2 Risoluzione dello stallo

Vogliamo mantenere il $2PL$ e risolvere il problema del *deadlock*. La maggior parte dei sistemi in commercio adotta la metodologia del *timeout*:

- Le transazioni rimangono in attesa di una rirsorsa per un tempo prefissato.
- Se, trascorso tale tempo, la risorsa non è stata ancora concessa, alla richiesta di lock viene data risposta negativa.
- In tal modo una transazione in potenziale stato di deadlock viene tolta dallo stato di attesa e di norma abortita.

Un valore troppo elevato tende a risolvere tardi i blocchi critici. Un valore troppo basso rischia di interpretare come blocchi anche situazioni in cui una transazione sta attendendo la disponibilità di una risorsa destinata a liberarsi, uccidendo la transazione e sprecando il lavoro già svolto.

Come scelgo la transazione da uccidere

- Politiche interrompenti:
  - Un conflitto può essere risolto uccidendo la transazione che possiede la risorsa (in tal modo questa rilascia la risorsa).
  - Un criterio aggiuntivo potrebbe essere uccidere le transazioni che hanno svolto minore lavoro.
- Politiche non interrompenti: una transazione può essere uccisa solo nel momento in cui effettua una nuova richiesta.

Conseguenza del criterio aggiuntivo Il problema fondamentale è che una transazione, all'inizio, accede ad un oggetto richiesto da altre transazioni. Risulta presente un conflitto e segue che questa sarà ripetutamente uccisa. Non ho un deadlock, ma *starvation*.

Soluzione? Mantenere invariato il timestamp delle transazioni abortite, dando in questo modo priorità alle transazioni più anziane.

12.2.2.1 Esempi

Abbiamo la seguente transazione

$$S : r1(y)w3(z)r1(z)r2(z)w3(x)w1(x)w2(x)r3(y)$$

Vogliamo mostrare l'esecuzione delle operazioni in S quando

- è applicato il two phase locking stretto
- è applicato il protocollo basato su timestamp

2PL

r1(y)	r1_lock(y)
w3(z)	w3_lock(z)
r1(z)	T1 wait T3
r2(z)	T2 wait T3
w3(x)	w3_lock(x)
r3(y)	r3_lock(y)
commit T3	unlock (y, z, x) dequeue T1, T2
r1(z)	r1_lock(z)
r2(z)	r2_lock(z)
w1(x)	w1_lock(x)
commit T1	unlock (z, x)
w2(x)	w2_lock(x)
commit T2	unlock (z, x)

TS

Assumiamo che una transazione abortita venga fatta ripartire immediatamente con un nuovo time-stamp

r1(y)	RTM(y)=1
w3(z)	WTM(z)=3
r1(z)	abort, restart T1 come T4
r4(y)	RTM(y)=4
r4(z)	RTM(z)=4
r2(z)	abort, restart T2 come T5
r5(z)	RTM(z)=5
w3(x)	WTM(x)=3
r3(y)	RTM(y)=4
w4(x)	WTM(x)=4
w5(x)	WTM(x)=5

12.3 Dettagli riguardanti SQL

Le transazioni sono partizionate in *read-only* e transazioni *read-write* (queste di default). Le prime non possono modificare nè il contenuto nè lo schema della base di dati e si basano esclusivamente su *lock condivisi*.

read-only Di queste possiamo scegliere un certo livello di isolamento. Ciò significa accettare delle anomalie.

- **read uncommitted**: possibili letture sporche, letture inconsistenti, aggiornamenti fantasma e inserimenti fantasma (niente lock in lettura, assenza di rispetto per i lock altrui)
- **read committed**: evito letture sporche ma permetto letture inconsistenti, aggiornamenti fantasma e inserimenti fantasma (lock in lettura, senza $2PL$)
- **repeatable read**: evito tutte le anomalie tranne gli inserimenti fantasma ($2PL$ anche in lettura)
- **serializable**: evito tutte le anomalie ($2PL$)

La perdita di aggiornamento è evitata in ogni caso.

read-write Il $2PL$ stretto è sempre utilizzato nelle scritture. Quindi si evitano tutte le anomalie (in particolare la perdita di aggiornamento)

Ricapitoliamo

- ***Perdita di aggiornamento (W-W)**: Abbiamo due transazioni che effettuano operazioni di scrittura su un oggetto. Le operazioni si basano su una precedente lettura. Il problema è che una delle letture non legge il dato aggiornato: segue che una delle due modifiche risulta persa!*
- ***Lettura sporca (R-W o W-W con abort)**: ho una transazione che legge e scrive e una che legge. La prima transazione abortisce, ma la seconda nel frattempo ha letto il valore di x modificato.*
- ***Letture inconsistenti (R-W)**: ho una transazione che svolge due operazioni di lettura e una che svolge un'operazione di lettura seguita da una di scrittura. In uno schedule serializzabile non è possibile che le due operazioni di lettura restituiscano valori diversi!*
- ***Aggiornamento fantasma (R-W)**: Abbiamo una transazione che legge dei valori aggiornati da un'altra transazione. L'interleaving potrebbe portare alla violazione di vincoli precedentemente imposti.*
- ***Inserimento fantasma (R-W)**: una transazione potrebbe effettuare una lettura di una serie di valori e successivamente usarli per calcolare un risultato. Tra la prima e la seconda cosa si ha l'inserimento, da parte di una seconda transazione, di un nuovo valore: questo sarà ignorato.*

13 — Mercoledì 20/05/2020

13.1 Organizzazione fisica e gestione della memoria

13.1.1 Memoria principale, memoria secondaria e gestione dei buffer

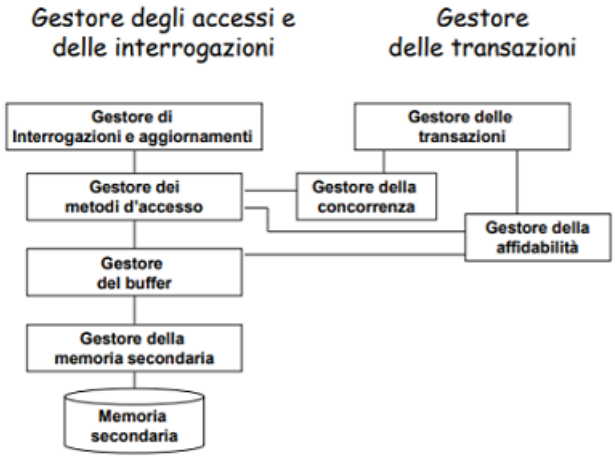
Il DBMS è una *scatola nera*. Non è strettamente necessario conoscere la sua struttura per poterne comprendere il funzionamento, ma in certe circostanze può essere utile.

Ricordiamo Un DBMS gestisce collezioni di dati grandi, persistenti, condivisi. Garantisce affidabilità e privatezza. Deve essere anche efficiente (ridurre i tempi e sfruttare al meglio le risorse) ed efficace (rendere produttiva l'attività di ogni singolo utente).

Grandezza e persistenza Il fatto che i dati siano grandi e persistenti richiede una gestione sofisticata della memoria secondaria. Gli utenti vedono il modello logico, ma le strutture che si celano dietro devono essere gestite efficientemente. Adotteremo delle strutture dati opportune.

Affidabilità La base di dati deve essere preservata, come già detto, anche in caso di malfunzionamento. L'affidabilità è impegnativa per via di aggiornamenti frequenti e per la gestione del buffer.

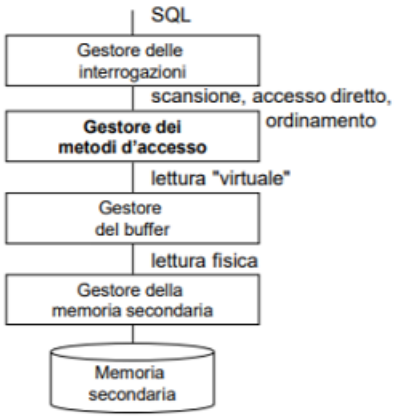
Condivisione Una base di dati è una risorsa condivisa fra più applicazione. Sono previsti meccanismi di autorizzazione e il cosiddetto controllo di concorrenza per gestire attività diverse e multi-utente su dati condivisi.



Tecnologie presenti

- Gestione della memoria secondaria e del buffer
- Organizzazione fisica dei dati
- Ottimizzazione delle interrogazioni (sfruttando il concetto di equivalenza e sfruttando dati a nostra disposizione per determinare la miglior forma di ottimizzazione)

Gestore degli accessi e delle interrogazioni

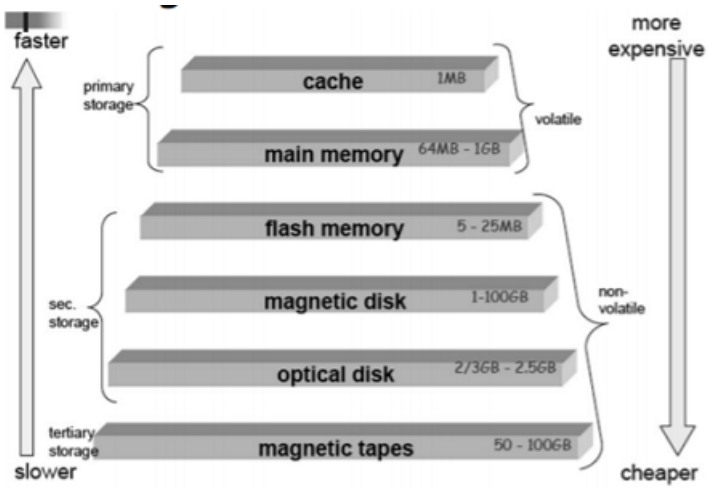


File system e DBMS

- Il file system è il componente del sistema operativo che gestisce la memoria secondaria.
- I DBMS ne utilizzano le funzionalità per creare ed eliminare file e per leggere/scrivere singoli blocchi o sequenze di blocchi contigui.
- Il DBMS gestisce i file allocati come se fossero un unico grande spazio di memoria secondaria e costruisce, in tale spazio, le strutture fisiche con cui implementa le relazioni.

Ricordare Il programma non può entrare in contatto con la memoria secondaria ma solo con quella principale. Abbiamo il buffer che permette un'interazione fra memoria principale e secondaria, limitando il più possibile gli accessi alla secondaria.

Gerarchia di memoria Abbiamo una serie di memoria, ciascuna con tecnologie e costi diversi. Si distinguono le memoria primarie, volatili, dalle memorie secondarie e terziarie, permanenti. Ovviamente il costo di mantenimento di una memoria di tipo permanente è più elevato.



Nell'accesso a memoria secondaria abbiano una testina che deve raggiungere il punto opportuno. Si considerano i seguenti tempi:

- tempo di posizionamento della testina (10-50ms)
- tempo di latenza (5-10ms)
- tempo di trasferimento (1-2ms)

In media non meno di 10ms. Si osserva che l'accesso a memoria secondaria è quattro o più ordini di grandezza maggiore di quello per operazioni in memoria principale. Segue che il costo delle operazioni dipende moltissimo dal numero di accessi in memoria secondaria.

Organizzazione delle memorie Ricordiamo che le memorie secondarie sono organizzate in blocchi di lunghezza fissa, mentre la memoria principale è organizzata in pagine. Nella memoria secondaria le uniche operazioni possibili sono quelle di lettura e scrittura dei dati di un blocco: in riferimento alla latenza (citata prima) si osserva che l'accesso a blocchi vicini è molto meno costoso (latenza a 0).

13.1.2 Buffer

Il buffer è un'area di memoria centrale gestita dal DBMS e condivisa fra le transazioni. Risulta organizzato in pagine di dimensioni pari o multiple di quelle dei blocchi di memoria secondaria (ricordiamoci che siamo in memoria principale). Il caricamento di una pagina del buffer equivale a un'operazione di lettura in memoria secondaria, mentre salvare una pagina equivale a un'operazione di scrittura.

Buffer manager Questo, oltre al buffer, si occupa di una directory che per ogni pagina mantiene il file fisico e il numero del blocco, e due variabili di stato:

- un contatore che indica quanti programmi utilizzano la pagina;
- un bit che indica se la pagina è *sporca*, cioè se è stata modificata.

Si considera:

- l'alta probabilità di dover riutilizzare i dati attualmente in uso
- la legge di Pareto: l'80% delle operazioni utilizza sempre lo stesso 20% dei dati.

Il buffer si occupa di:

- di ricevere richieste di lettura e scrittura dalle transazioni
- di eseguire queste operazioni accedendo alla memoria secondaria solo quando indispensabile

Ricordiamo, a proposito di queste richieste, le primitive:

- fix: richiesta di pagina. Richiedo una lettura solo se la pagina non è nel buffer. Incrementa il contatore associato alla pagina offerta dal buffer manager
- setDirty: comunica che la pagina è stata modificata
- unfix: indica che la transazione ha concluso l'utilizzo della pagina (decremento il contatore associato alla pagina)
- force: trasferisco una pagina in memoria secondaria (su richiesta del gestore dell'affidabilità, non del gestore degli accessi)

Esecuzione della fix

- cerco la pagina nel buffer
- se la trovo restituisco l'indirizzo
- se non la trovo cerco una pagina libera nel buffer (contatore 0):
  - se la trovo inserisco i dati letti dalla memoria secondaria e ne restituisco l'indirizzo
  - se non la trovo ho due opzioni:
    - seleziono una pagina *vittima* occupata dal buffer, scrivo i dati di questa in memoria secondaria, effettuo le letture in memoria secondaria e restituisco l'indirizzo
    - pongo in attesa l'operazione

Scopo della gestione del buffer Vogliamo ridurre il numero di accessi alla memoria secondaria:

- in caso di lettura, se la pagina è già presente nel buffer non c'è bisogno di accedere alla memoria secondaria
- in caso di scrittura, il gestore del buffer può decidere di differire la scrittura fisica in modo da accorparla ad altre scritture.

Scrittura in memoria secondaria Il buffer manager può far partire le scritture in due contesti diversi:

- in modo sincrono quando è richiesto esplicitamente con una force
- in modo asincrono, cioè quando lo ritiene opportono. Si possono anticipare o posticipare scritture per coordinarle e/o sfruttare la disponibilità dei dispositivi.

13.1.3 Gestione delle tuple nelle pagine

Tuple e blocchi

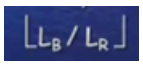
- Il file system ha i suoi file: questi sono logicamente organizzati in record.
- I record sono mappati nei blocchi di memoria secondaria.
- Le tuple di una relazione (record di file) stanno in blocchi contigui. A volte in un blocco ci sono tuple di relazioni diverse ma correlate (i JOIN sono favoriti)
- I blocchi (componenti fisici di un file) e le tuple o record (componenti logici di una relazione) hanno dimensioni in generale diverse:
  - la dimensione del blocco dipende dal file system
  - la dimensione del record dipende dalle esigenze dell'applicazione e può anche variare nell'ambito di un file.

Organizzazione delle tuple Ho varie alternative possibili. Solitamente inseriamo le tuple in modo sequenziale nei file. Inoltre:

- se la lunghezza delle tuple è fissa, la struttura può essere semplificata
- alcuni sistemi possono spezzare le tuple su più blocchi (tuple grandi, cosa difficile da gestire)

Fattore di blocco Attraverso il fattore di blocco stabilisco quanti record posso porre in un blocco.

- L_R: dimensione di un record (per semplicità costante nel file: record a lunghezza fissa)
- L_B: dimensione di un blocco
- se $L_B > L_R$ possiamo avere più record in un blocco



- lo spazio residuo può essere utilizzato per record *spanned* o non utilizzato.

Esercizio Calcolare il fattore di blocco e il numero di blocchi occupati da una relazione contenente $T = 500000$ tuple di lunghezza fissa pari a $L = 100$ byte in un sistema con blocchi di dimensione pari a $B = 1$ kilobyte. Risolviamo così:

- $N_B = D_T/B$
- $D_T = T * L$
- $F_B = B/L$
- $F_B = 1024/100$

Operazioni sulla pagina

- Inserimento/Modifica di una tupla (la cosa può richiedere allocazione di ulteriore spazio)
- Cancellazione
- Accesso ad una tupla o ad un campo di una tupla

13.1.4 Strutture per l'organizzazione di file

Definizioni

- Le tuple organizzate all'interno dei blocchi dei file costituiscono le **strutture primarie**, cioè quelle che contengono propriamente i dati. Le principali sono
  - strutture sequenziali (seriali / ordinate / ad array)
  - strutture ad accesso calcolato (hash)
  - strutture non sequenziali ad albero
- Esistono anche blocchi contenenti **strutture secondarie**, che sono quelle che favoriscono l'accesso ai dati senza contenerli (in un certo senso sono dei puntatori ai dati).

13.1.4.1 Strutture primarie sequenziali

Una struttura primaria sequenziale può essere

- **seriale** (detto anche *heap*, miscuglio): ordinamento fisico ma non logico. Gli inserimenti vengono effettuati:
  - in coda (gli elementi sono di fatto posti nell'ordine di inserimento)
  - al posto di record cancellati (che ho precedentemente sostituito con una *marca* che mi segnala la cancellazione)

  L'eliminazione, ma anche la sostituzione di una tupla con un'altra di dimensione minore, comportano uno spreco di spazio e un conseguente aumento del numero di blocchi utilizzati: segue la necessità di svolgere riorganizzazioni periodiche. Una ricerca è di tipo sequenziale: segue una complessità lineare e la necessità di scorrere, nel caso peggiore, tutto il file.
- **ordinata**: ordinamento fisico delle tuple coerente con quello di un campo detto chiave (per esempio, data una lista di studenti universitari, la matricola)
  - Sono possibili ricerche binarie

  Le riorganizzazioni periodiche sono necessarie anche in questo tipo di organizzazione. Si ha un'ulteriore problema: dobbiamo mantenere l'ordinamento a seguito di un qualunque cambiamento. Ciò rende le riorganizzazioni decisamente più importanti e complesse.
- **array**: le tuple hanno stessa dimensione, all'interno di uno o più blocchi contigui abbiamo una serie di posizioni identificate da indici.
  - Il caricamento iniziale delle informazioni avviene mediante incremento di un contatore e inserimento nelle varie posizioni

- La cancellazione lascia spazi vuoti
- Gli inserimenti possono essere effettuati solo in posizioni vuote o in fondo al file.

Generalmente questa struttura non è quasi mai utilizzata nei DBMS.

13.1.4.2 Strutture primarie ad accesso calcolato

Questa struttura, detta anche *hash*, sfrutta le proprietà tipiche dell'organizzazione sequenziale ad array anche in circostanze in cui l'array non risulta immediatamente applicabile. L'array, normalmente, è ottimo, per organizzare insiemi di record che hanno come *campo chiave* valori consecutivi (esempio numero di matricole da 1 a 10000): questa cosa, in moltissime situazioni, non è possibile.

- i file hash permettono un accesso diretto molto efficiente.
- Ricordiamo che la funzione hash:
  - associa ad ogni valore della chiave un "indirizzo", in uno spazio di dimensione leggermente superiore rispetto a quello strettamente necessario
  - poiché il numero di possibili chiavi è molto maggiore del numero di possibili indirizzi, la funzione non può essere iniettiva e quindi esiste la possibilità di collisioni (chiavi diverse che corrispondono allo stesso indirizzo)
  - le buone funzioni hash distribuiscono in modo causale e uniforme, riducendo le probabilità di collisione (che si riduce aumentando lo spazio ridondante)

Come risolvo le collisioni? Esistono varie tecniche:

- Occupare le posizioni successive
- Ricorrere a tabelle/catene di overflow (gestita in forma collegata)
- Adottare funzioni hash alternative

Osserviamo che:

- le collisioni sono quasi sempre presenti (tenendo conto che non possiamo avere funzioni iniettive nella maggior parte dei casi)
- le collisioni multiple hanno probabilità che decresce al crescere della molteplicità
- la molteplicità media delle collisioni è molto bassa

Hash su file Ricordiamo l'organizzazione di un file: esso è diviso in blocchi, e questi a loro volta presentano all'interno delle tuple. La funzione hash restituisce un indice rappresentante il blocco all'interno del quale è presente la tupla incriminata. Questo ci permette di ammortizzare le probabilità di collisione

- In caso di conflitto il record è memorizzato nello stesso blocco dove sono presenti altre tuple

- Quando lo spazio di un blocco è esaurito viene allocato un ulteriore blocco, collegamento al precedente, e il record viene posto al suo interno. Questa tecnica è detta *catena di overflow*.
- Segue che per una parte di tuple, associate a un certo indice, sarà necessario un solo accesso a blocco. Per questa tupla, appena aggiunta, invece, serviranno due accessi!

Esempio di esercizio

- Prendiamo la seguente tavola hash (con collisioni) e facciamo delle analisi relative all'*hash su file*

- 40 record
- tavola hash con 50 posizioni:
  - 1 collisione a 4
  - 2 collisioni a 3
  - 5 collisioni a 2

M	M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49

- Consideriamo:
  - T il numero di tuple previsto per il file
  - F il fattore di blocco
  - f il fattore di riempimento (la frazione dello spazio fisico disponibile mediamente utilizzata)
- Svolgiamo il seguente calcolo per trovare il numero di blocchi utilizzato

$$B = \frac{T}{f \times F} = \frac{40}{\frac{4}{5} \times 10} = 5$$

- Abbiamo 5 blocchi con 10 posizioni ciascuno!
- Prendiamo la seguente figura, che contiene gli stessi dati della precedente tavola hash

60600	66301	205802	200268	200604
66005	205751	200902	205478	201159
116455	115541	116202	210533	200464
200205	200296	205912	200138	205619
205610	205796	205762	102338	205724
201260		205617	205693	206049
102360		205667	200498	
205460		210522		
200430		205977		
102690		205887		
205845		206092		

Complessivamente abbiamo 30 chiavi: per raggiungere due chiavi è necessario l'accesso a due blocchi, uno solo per le rimanenti 28.

13.1.4.3 Strutture ad albero (non sequenziali, dette *indici*)

Le strutture ad albero, come quelle *hash*, favoriscono l'accesso ai dati in base al valore di uno o più campi. Sono consentiti accessi

- puntuali
- corrispondenti a intervalli di valori

L'organizzazione ad albero, come vedremo, può essere utilizzata per realizzare sia strutture primarie che secondarie.

Cosa intendiamo con indice? Supponiamo di avere un file f e un campo chiave (che può avere al suo interno più attributi): un **indice secondario** consiste in un altro file dove ciascun record è logicamente composto da due campi:

- uno contenente il valore del campo chiave
- uno contenente l'indirizzo o gli indirizzi fisici dei record del file f che hanno quel valore di chiave

Un esempio per capire in cosa effettivamente consiste un indice secondario è l'indice analitico presenti in fondo a certi libri: ho un ordine alfabetico dei termini presenti e le pagine dove sono presenti tali termini. Un indice si dice **indice primario** se l'ordinamento è lo stesso della struttura dati o se contiene al suo interno i dati: in questo caso non garantisco solo un accesso in base a un campo chiave, ma contengo anche i record fisici necessari per memorizzare i dati. Un esempio per comprendere l'indice primario è l'indice generico presente in ogni libro: ho le sezioni e il numero delle pagine dove queste sono posizionate.

File Un file può avere un solo indice primario (questo non è presente solo quando l'organizzazione primaria è hash oppure sequenziale) e più indici secondari.

Puntatori in un indice I puntatori di un indice reindirizzano

- al blocco dove è presente la tupla, o
- alla tupla stessa.

il tempo per effettuare la ricerca di una tupla all'interno di un blocco è trascurabile. Questo ci permette di limitare, in certi casi, il numero di puntatori per blocco: potrei puntare alla tupla contenente valore della chiave minore o maggiore.

Indici densi e sparsi Consideriamo gli indici con puntatori a tuple. Essi saranno detti:

- **densi**, se tutte le tuple dei vari blocchi sono puntate da elementi dell'indice (solitamente quando adottiamo un'ordinamento diverso da quello fisico e quindi non è più possibile garantire la presenza di una tupla all'interno di un blocco).
- **sparsi**, se non tutte le tuple sono puntate (solitamente quando seguiamo l'ordinamento fisico, quindi puntando a una certa tupla di un blocco - rappresentante un valore di riferimento - sono certo che ciò che sto cercando sarà all'interno di quella tupla)

Gli indici secondari sono per forza densi, poichè basati su criteri di ordinamento diversi da quello fisico. L'indice primario può essere sparso.

Ricerche Il fatto di avere un ordinamento permette di svolgere ricerche binarie con complessità logaritmica. Sono possibili anche ricerche basate su intervalli e scansioni sequenziali ordinate (cose inefficienti sulle strutture hash).

Caratteristiche degli indici

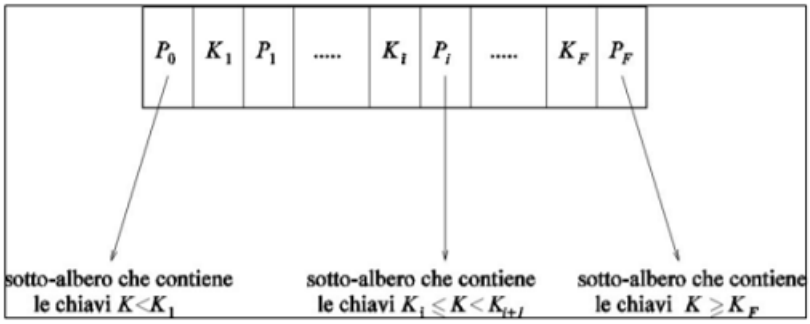
- Accesso diretto ed efficiente sulla chiave
- Modifiche della chiave, inserimenti, eliminazioni inefficienti (come nei file ordinati). Possiamo adottare le seguenti tecniche per migliorare la situazione:
  - file o blocchi di overflow
  - marcatura per le eliminazioni
  - riempimento parziale
  - blocchi collegati (non contigui)
  - riorganizzazioni periodiche

Segue una scarsa flessibilità in presenza di elevata dinamicità.

Indici multilivello

- Gli indici sono file essi stessi e quindi ha senso costruire indici sugli indici, per evitare di fare ricerche fra blocchi diversi
- Possono esistere più livelli fino ad avere il livello più alto con un solo blocco; i livelli sono di solito abbastanza pochi, perché
  - ogni nodo (al di fuori delle foglie) ha un numero di discendenti abbastanza elevato, dipendente dall'ampiezza della pagina.
  - l'indice è ordinato, quindi l'indice sull'indice è sparso
  - i record dell'indice sono piccoli

Sfruttando questi concetti andiamo a costruire degli alberi dove ciascun nodo consiste in un file, quindi in un indice!



Indici e alberi binari

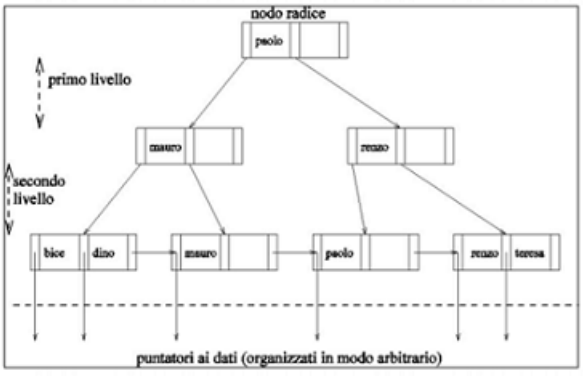
- Gli indici utilizzati dai DBMS sono in generale
  - indici dinamici multilivello
  - Vengono memorizzati e gestiti come B-tree (intuitivamente: alberi di ricerca bilanciati): Alberi binari di ricerca, Alberi n-ari di ricerca, Alberi n-ari di ricerca bilanciati

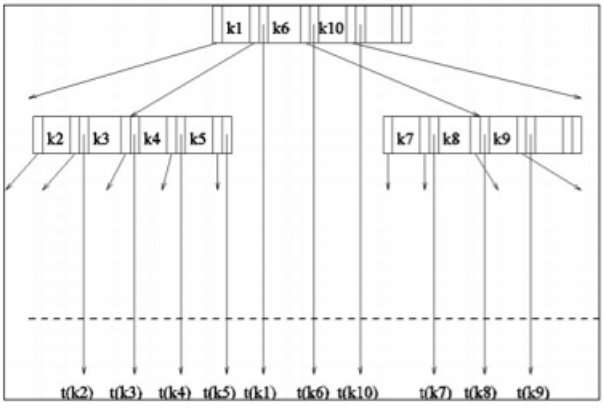
Si ricorre ad alberi generici di ordine P (dove ciascun nodo presenta fino a P figli e $P-1$ etichette). Questi alberi saranno tenuti bilanciati nel tempo grazie a:

- Riempimento parziale (mediamente 70%)
- Riorganizzazioni (locali) in caso di sbilanciamento

Si distingue una versione B-tree da una B+-tree:

- nel B+-tree a le foglie costituiscono una lista e sono collegate fra di loro
- nel B-tree possiamo avere, in nodi intermedi, puntatori diretti ai dati.





Entrambi gli alberi sono molto efficienti: la forma preferita è la B+-tree in quanto permette di svolgere ricerche basate su un intervallo di valori. Con le foglie collegate posso effettuare selezioni basate su un intervallo di valori: individuo il primo valore dell'intervallo e poi effettuo una scansione sequenziale per trovare i valori maggiori rispetto a questo.

Definizione degli indici SQL Non è standard ma presente in forma simile nei vari DBMS

- `create [unique] index IndexName on TableName(AttributeList)`
- `drop index IndexName`

Diapositive Vedere attorno alla diapositiva 66 le strutture fisiche presenti in alcuni DBMS

13.2 Esecuzione e ottimizzazione delle interrogazioni

Riprendiamo un discorso iniziato con l'algebra relazionale. Abbiamo visto il *query processor*, un modulo del DBMS nel quale avviene un processo di ottimizzazione basato su equivalenze (e sul catalogo, che contiene una serie di informazioni utili per l'esecuzione[1]). Le interrogazioni sono espresse ad alto livello (ricordare il concetto di indipendenza dei dati): insiemi di tuple, poca proceduralità.

Fase finale di ottimizzazione Abbiamo lasciato in sospeso l'ultima delle tre fasi del *query processor*. In questa abbiamo un'**ottimizzazione in base ai costi**: consideriamo il numero di trasferimenti in memoria da fare e stimiamo le dimensioni dei risultati intermedi. Nell'ottimizzazione dobbiamo tener conto anche

- delle operazioni da eseguire (vedere paragrafo dopo)
- i dettagli del metodo (per esempio quale tipo di JOIN eseguire)
- ordine delle operazioni (ho un join di tre relazioni, da dove mi conviene partire?)

Implementazione degli operatori dell'AR Le interrogazioni sono svolte mediante un linguaggio ad alto livello, cioè lontano dalla macchina. I DBMS implementano gli operatori dell'algebra relazionale per mezzo di operazioni di livello abbastanza basso. Abbiamo

- **Operatori fondamentali**: accesso diretto e scansione
- **A livello più alto**: ordinamento
- **A livello ancora più alto**: JOIN, operazione più costosa (che coinvolge, ricordiamoci, il prodotto cartesiano).

[1] Esempi di profili:

- cardinalità di ciascuna relazione
- dimensioni delle tuple e dei valori degli attributi
- numero di valori distinti degli attributi
- valore minimo e massimo di ciascun attributo

13.2.1 Accesso diretto

Fare *accesso diretto* significa ottenere, dato il valore di un campo, l'indirizzo del blocco in cui il record si trova. Ciò può essere fatto solo con strutture hash o indici. I tipi di accessi sono:

- puntuali, del tipo $A_i = V$
- su intervallo, del tipo $V_1 \leq A_i \leq V_2$

Il primo tipo di accesso è efficiente in entrambe le strutture (in particolare in quelle hash), il secondo è efficiente solo negli accessi basati su indice.

Interrogazioni Nelle interrogazioni consideriamo predicati congiuntivi e disgiuntivi. Se si ha un solo predicato predicato valutabile conviene usare indice o hash. In caso contrario:

- **predicati congiuntivi**[2]: prima scelgo il più selettivo (quello che mi leva più tuple), dopo verifico gli altri scorrendo quanto salvato in memoria principale (cioè le tuple che rispettano il primo predicato). Nel caso hash pongo come argomento della funzione il cognome e verifico l'altra condizione.
- **predicati disgiuntivi**[3]: se tutti i predicati sono valutabili utilizzo gli indici per ogni elemento. Sono convenienti, rispetto a una scansione, solo se molto selettivi. Bisogna fare attenzione anche ai duplicati. Il caso nel footnote è impossibile in hash.

13.2.2 Scansione

La scansione consiste in un'operazione di ricerca: è implementabile tramite un algoritmo di ricerca la cui complessità media, date n tuple, è lineare. **I blocchi vengono trasferiti dalla memoria secondaria al buffer**.

Metodo alternativo Un metodo alternativo è il ricorso agli indici. Inoltre, se

- le tuple sono ordinate
- la selezione è fatta sull'attributo su cui la relazione è ordinata
- i blocchi sono memorizzati in modo contiguo

allora possiamo ottenere un numero medio di trasferimenti logaritmico $\log_2 n$.

13.2.3 Ordinamento

La procedura di ordinamento è necessaria sia per ottenere risultati ordinati che per una corretta realizzazione delle proiezioni, con eliminazione dei duplicati. Risulta utile anche per implementare in modo efficiente operazioni come il JOIN, o il raggruppamento. Possiamo ottenere l'ordinamento con:

- **quickSort**, se la relazione può essere posta tutta nel buffer;
- **mergeSort**, se la relazione non può essere posta tutta nel buffer e quindi dobbiamo lavorare su più frammenti.

[2] Cognome = 'Rossi' AND Nome = 'Maria'
[3] Cognome = 'Rossi' OR Nome = 'Maria'

13.2.4 JOIN

Il JOIN è l'operazione a più alto livello. Si hanno vari metodi:

- Nested-loop (*quello che si fa ad occhio* - cit)
- Merge scan
- Hash-based

13.2.4.1 Nested-loop

Algoritmo Abbiamo una tabella, una *interna* e una *esterna*. Per ogni tupla della tabella esterna scandisco la tabella interna individuando le tuple che fanno JOIN.

Costo

- Prendiamo le seguenti tabelle:
  - Tabella R con 10.000 record che occupano 400 blocchi
  - Tabella S con 5.000 record che occupano 100 blocchi.
- Consideriamo R esterna
- Prendiamo il caso peggiore: nel buffer si può inserire solo un record di S alla volta, quindi devo effettuare innumerevoli accessi. Ottengo il seguente numero di trasferimenti

  $$N = 10000 * 100 + 400$$

  Per ogni tupla della tabella R scandisco tutti i blocchi della tabella S, portandoli ripetutamente in memoria. La tabella R rimane nel buffer, quindi la porto in memoria principale soltanto una volta.
- Nel caso migliore, cioè quando tutta la tabella S entra nel buffer, ottengo

  $$N = 400 + 100$$

13.2.4.2 Merge scan

Questa tecnica si basa sull'ordinamento delle tabelle in base agli attributi di JOIN. L'algoritmo risulta molto efficiente quando le tabelle sono già ordinate o se sono presenti indici adeguati.

- Fondamentalmente abbiamo una scansione in parallelo delle due tabelle, come nella funzione *merge* dell'algoritmo *mergeSort*.
- Quando i valori dei due attributi coincidono ottengo una tupla appartenente al risultato. Il meccanismo fa sì che il risultato sia ordinato.

Il costo consiste nei trasferimenti in memoria ($N = br + bs$, ogni blocco è letto una volta soltanto assumendo che tutte le tuple per un dato valore di JOIN stiano insieme nel buffer) e nell'ordinamento (questo se le tabelle non sono già ordinate). L'algoritmo è utilizzato per il JOIN naturale o l'Equi-JOIN.

13.2.4.3 Hash-JOIN

- Utilizzo una funzione h di hash sugli stessi attributi per memorizzare una copia di ciascuna delle due tabelle (difetto del metodo è un utilizzo maggiore di memoria) in memoria centrale
- La funzione h comporta una partizione delle tabelle R ed S coinvolte: dati i valori del dominio di tali attributi ottengo come risultato lo stesso indice.
- L'indice identificherà una partizione di R ed una di S: a quel punto mi basta effettuare dei semplici JOIN tra queste partizioni.

Questa tecnica è utile solo per JOIN-naturale ed Equi-JOIN, quest ultimo svolto in modo più efficiente se confrontato con il nested-loop.

Costo del metodo

- Le relazioni R ed S, per essere partizionate, devono essere portate in memoria centrale. Intanto individuiamo i seguenti trasferimenti

  $$2(br + bs)$$

  Due volte poichè dobbiamo svolgere un'operazione di lettura e una di scrittura
- Per ogni tupla t_r, in ogni partizione di R, si considera la tupla t_s nella partizione corrispondente secondo la funzione hash, quindi si legge ciascuna partizione una volta. Seguono ulteriori trasferimenti

  $$(br + bs)$$
- Il costo complessivo dell'hash-JOIN sarà

  $$3(br + bs)$$

13.2.5 Misura del costo di una query

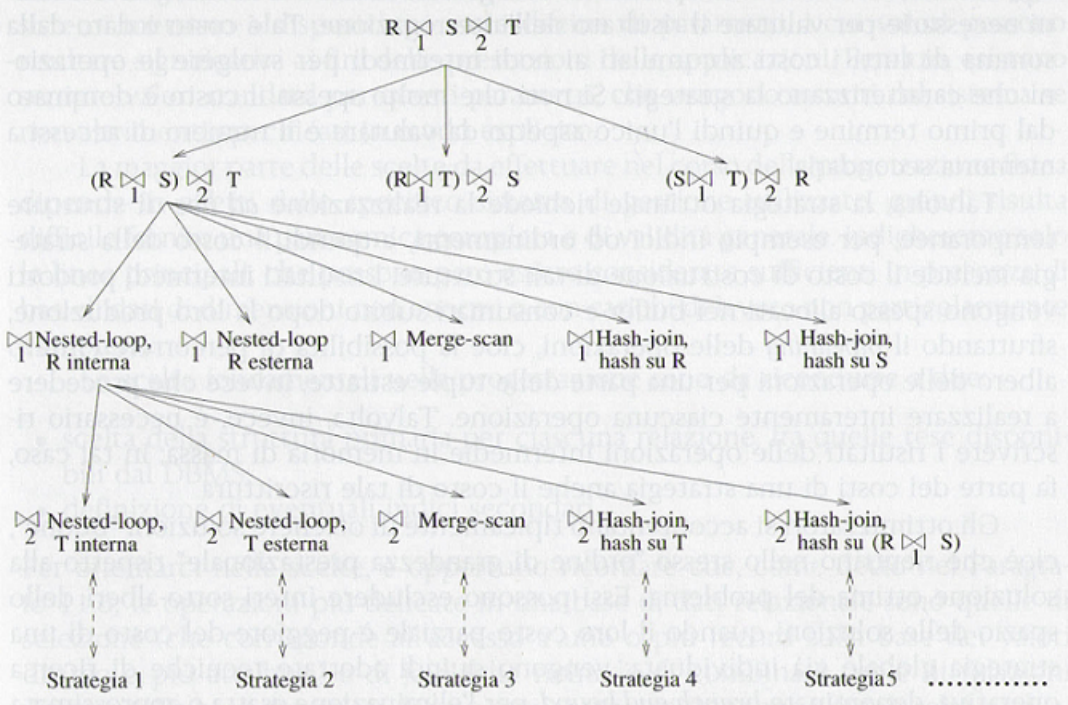
Molti fattori contribuiscono al costo di una query, cioè il tempo necessario per avere la risposta:

- L'accesso al disco, il tempo predominante calcolabile considerando:
  - il numero di scansioni
  - il numero di lettura
  - il numero di scritture
- il tempo di CPU
- il tempo di rete

Poniamo per semplicità il fattore t unico come tempo per accedere a un blocco.

- Dobbiamo considerare, oltre al numero di trasferimenti, la memoria utilizzata per memorizzare i risultati intermedi.
- Abbiamo ribadito come l'hash-JOIN necessiti di tabelle intermedie.
- Operazioni con più operandi devono essere eseguite uno step alla volta.
  - **Congiunzione di condizioni**:
    - Una congiunzione di operazioni di selezione può essere scomposta in una sequenza di selezioni semplici
      $$\sigma_{\theta 1 \ni \theta 2}(R) = \sigma_{\theta 1}(\sigma_{\theta 2}(R))$$
    - Le operazioni di selezione sono commutative
      $$\sigma_{\theta 1}(\sigma_{\theta 2}(R)) = \sigma_{\theta 2}(\sigma_{\theta 1}(R))$$
    - L'ordine viene scelto in base alla dimensione del risultato intermedio.
  - **Ordine dei JOIN**
    - L'ordine in cui si fanno i JOIN dipende dalla dimensione dei risultati intermedi
    - Date le relazioni $R1, R2, R3$, sappiamo che vale la proprietà associativa
      $$(R1 \bowtie R2) \bowtie R3 = R1 \bowtie (R2 \bowtie R3)$$
    - Se la dimensione di $R2 \bowtie R3$ è molto grande e quella di $R1 \bowtie R2$ è piccola, conviene eseguire i JOIN nel seguente ordine
      $$(R1 \bowtie R2) \bowtie R3$$
      in modo da dover memorizzare una relazione intermedia più piccola

Processo di ottimizzazione Si costruisce un albero di decisione con le varie alternative, si valuta il costo di ciascun piano e si sceglie il piano di costo minore. L'ottimizzatore sceglie di solito una soluzione *buona* non necessariamente una soluzione *ottima*.



Esercizio Calcoliamo il piano di esecuzione migliore per la seguente interrogazione dal punto di viste della dimensione dei risultati intermedi

$$\pi_T(\sigma_{(C=D \vee C=B) \wedge A=X \wedge N=Y}(F \bowtie Ac \bowtie I)$$

Abbiamo le seguenti tabelle

```
F(Fid, Titolo, Anno, Categoria, ...)
Ac(Aid, Nazionalità, ...)
I(Fid, Aid,...)
```

E i seguenti dati:

- Numero di tuple in F: $N(F) = 30000$
- Numero di tuple in A: $N(A) = 2000$
- Numero di tuple in I: $N(I) = 600000$
- Numero di film distinti nelle interpretazioni: $N(Fid, I) = 30000$
- Numero di attori distinti nelle interpretazioni: $N(Aid, I) = 2000$
- Numero di nazionalità distinte tra gli attori: $N(Nazionalita, A) = 4$
- Numero di categorie distinte tra i film: $N(Categoria, F) = 5$

- Numero di anni distinti tra i film: $N(Anno, F) = 20$

Procediamo

1. Per prima cosa eseguiamo le ottimizzazioni sempre valide. Applicando *push selections down* e *push projections down* otteniamo

$$\pi_T(\pi_{T,FId}(\sigma_{(C=D \vee C=B) \wedge A=X}(F)) \bowtie \pi_{AId}(\sigma_{N=Y}(Ac)) \bowtie \pi_{FId,AId}(I))$$

2. Prendiamo $\pi_{T,FId}(\sigma_{(C=D \vee C=B) \wedge A=X}(F))$

   - $\sigma_{(C=D \vee C=B)}(F)$
     - Vogliamo stabilire che gli oggetti appartengono alla categoria C o alla categoria D
     - Calcoliamo il numero di elementi appartenenti a una categoria

$$\frac{N(F)}{N(Categoria, F)} = \frac{30000}{5} = 6000$$

     - Poichè mi interessano due categorie moltiplico per 2: $6000 * 2 = 12000$
   - $\sigma_{A=X}(F)$
     - Vogliamo anche che siano selezionati solo i film realizzati nell'anno X
     - Per questo calcoliamo il numero di film per anno

$$\frac{N(F)}{N(Anno, F)} = \frac{30000}{20} = 1500$$

   - Attraverso i calcoli precedenti individuiamo che l'ordine più conveniente è il seguente

$$\sigma_{(C=D \vee C=B)}(\sigma_{A=X}(F))$$

   - Il numero finale di tuple si ottiene risvolgendo i calcoli: non pongo 30000 tuple ma 1500

$$\frac{1500}{5} * 2 = 300 * 2 = \boxed{600}$$

   - La proiezione $\pi_{T,FId}(\dots)$ non altera il numero di tuple visto che contiene la chiave. Il numero di tuple trovato prima (300) è confermato!

3. $\pi_{AId}(\sigma_{N=Y}(Ac))$

   - Consideriamo la selezione: vogliamo soltanto gli attori di nazionalità Y
   - Calcoliamo il numero di attori per nazionalità

$$\frac{N(Ac)}{N(Nazionalita, Ac)} = \frac{2000}{4} = \boxed{500}$$

   - La proiezione non altera il numero di tuple poichè si proietta la chiave di Ac

4. $\pi_{FId,AId}(I))$

- Il numero di tuple relative all'interpretazione è $N(I) = \boxed{600000}$
- La proiezione contiene la chiave, quindi il numero di tuple non viene alterato.

5. Abbiamo determinato l'ordine delle selezioni ed eventali riduzioni di tuple a causa delle proiezioni

6. Adesso dobbiamo determinare l'ordine dei JOIN.

   - $\pi_{T,FId}(\sigma_{(C=D \vee C=B) \wedge A=X}(F)) \bowtie \pi_{AId}(\sigma_{N=Y}(Ac))$
     - Non avendo attributi comuni il JOIN naturale risulta in un prodotto cartesiano. Segue il seguente numero di tuple

       $$500 * 600 = 300000$$

   - $\pi_{AId}(\sigma_{N=Y}(Ac)) \bowtie \pi_{FId,AId}(I)$
     - $\min(500 * 600000/2000, 600000 * 1) = 150000$
   - $\pi_{T,FId}(\sigma_{(C=D \vee C=B) \wedge A=X}(F)) \bowtie \pi_{FId,AId}(I)$
     - $\min(600 * 600000/30000, 600000 * 1) = 6000$

7. Attraverso questi risultati finali sappiamo a quale JOIN dare priorità, cioè quello che mi restituisce il minor numero di tuple! Il JOIN in questione è

   $$\pi_{T,FId}(\sigma_{(C=D \vee C=B) \wedge A=X}(F)) \bowtie \pi_{FId,AId}(I)$$

13.3 Progettazione fisica: fase finale

La progettazione fisica consiste nella fase finale del processo di progettazione di una base di dati.

- **Input**: Schema logico e informazioni sul carico applicativo
- **Output**: Schema fisico, costituito dalle definizioni delle relazioni con le relative strutture fisiche (e molti parametri, spesso legati allo specifico DBMS)

Dettagli

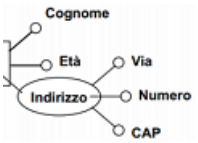
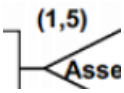
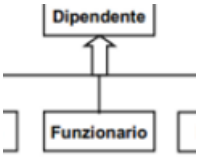
- La caratteristica comune dei DBMS relazionali è la disponibilità degli indici: la progettazione logica, spesso, coincide con la scelta degli indici.
- Le chiavi primarie delle relazioni sono di solito coinvolte in selezioni e JOIN: molti sistemi prevedono di definire indici sulle chiavi primarie.
- Altri indici vengono definiti con riferimento ad altre selezioni o JOIN importanti
- Se le prestazioni sono insoddisfacenti si pongono nuovi indici o si eliminano indici già esistenti. Attraverso il comando

  ```
  show plan
  ```

  possiamo caricare la lista degli indici creati.

Parte III

Roba aggiuntiva

Immagine	Descrizione
Entità	
	Classe di oggetti esistenti, presenti nella realtà: persone, cose, città, conti correnti... Ad ogni elemento sono associate proprietà comuni. **Occorrenze**: elemento appartenente alla classe **Nomi**: singolari e significativi
Associazione	
	Legame fra due o più entità. Non ho frecce: non si ha un verso. **Occorrenze**: insieme di tuple, di occorrenze di entità. **Nomi**: singolari, significativi, possibilmente senza verbi (no verso).
Attributo	
	Proprietà elementari tipiche di un'entità o di un'associazione. Ogni attributo presenta un certo dominio ed è per forza associato ad occorrenze di entità e di associazione. **Raggruppamenti**: gli attributi possono essere raffigurati in gruppi (per esempio le informazioni riguardanti un indirizzo).
Cardinalità	
	Coppia di valori che specifica il numero minimo e massimo di occorrenze della relazione a cui può partecipare una occorrenza di entità. Banalmente, indico quante coppie posso stabilire in un'associazione. Presente sia in associazioni che in attributi! **Simbologia**: 0 (associazione opzionale) e 1 (associazione obbligatoria) per la cardinalità minima; 1 e N (numero formalmente illimitato di associazioni) per cardinalità massima.
Generalizzazione	
	Costrutto gerarchico che viene perso col passaggio al modello logico. Mi permette di stabilire delle sottoclassi di entità. Presentano degli attributi specifici che li differenziano dalle altre sottoclassi! **Tipi**: la generalizzazione è totale (freccia piena, genitore occorrenza di almeno uno dei figli), o parziale (freccia vuota, genitore non è per forza occorrenza di uno dei figli). La generalizzazione si dice anche esclusiva (genitore è occorrenza di al più una delle entità figlie) o sovrapposta (genitore occorrenza di più di una delle entità figlio). **Alberi**: posso stabilire degli alberi di gerarchia parallela in cui distinguere generalizzazione totale da quella parziale.

A — Esempi di espressioni in algebra relazionale

Impiegati e supervisione

```
IMPIEGATI(Matricola, Nome, Eta, Stipendio)
SUPERVISIONE(Impiegato, Capo)
```

Trovare le matricole dei capi degli impiegati che guadagnano più di 40.000 euro

$$\Pi_{Capo}((Supervisione) \bowtie_{Impiegato=Matricola} (\Pi_{Matricola}(\sigma_{Stipendio>40.000}(Impiegati))))$$

- Osservo che le informazioni necessarie si trovano in entrambe le tabelle: procederemo con un JOIN associando ai supervisori i record degli impiegati che dirigono.
- Ricordiamo la *pushing selections down* e la *pushing projections down*: le proiezioni e le selezioni si eseguono prima di applicare l'operatore JOIN.
- Non applico proiezioni a Supervisione, tutti gli attributi mi servono: *Capo* per essere proiettato e *Impiegato* per il JOIN
- Mi interessano soltanto gli impiegati che hanno stipendio superiore a 40.000 euro: prima di applicare la proiezione della matricola (che mi serve per il JOIN) escludo coloro che non soddisfano la condizione.

Trovare nome e stipendio dei capi degli impiegati che guadagnano più di 40.000 euro

$$\Pi_{Nome,Stipendio}((Impiegati) \bowtie_{Matricola=Capo} (\Pi_{Capo}(Supervisione \bowtie_{Impiegato=Matricola} \Pi_{Matricola}(\sigma_{Stipendio>40.000}(Impiegati)))))$$

- Contrariamente a prima devo indicare *Nome* e *Stipendio*. Dovremo fare un ulteriore JOIN per individuare questi dati e proiettarli.
- L'espressione del primo esempio fa JOIN con Impiegati per trovare le informazioni del supervisore corrispondente

Trovare le matricole dei capi i cui impiegati guadagnano tutti più di 40.000 euro

$$\Pi_{Capo}(Supervisione) - \Pi_{Capo}(\\ (Supervisione) \bowtie_{Impiegato=Matricola} \Pi_{Matricola}(\sigma_{Stipendio \leq 40.000}(Impiegati)) \\ )$$

- A differenza di prima troviamo il termine *tutti*, quindi non mi limito a verificare un solo impiegato ma tutti gli impiegati. Faccio questo attraverso la sottrazione: la differenza avviene tra l'insieme di tutti i capi e quello dei capi che hanno impiegati con stipendio minore o uguale a 40.000. Il risultato è una relazione con i capi aventi solo impiegati che guadagnano più di 40.000 euro al mese.
- Il secondo termine della differenza si ottiene attraverso un JOIN tra i supervisori e i record degli impiegati che dirigono. Prima di fare JOIN seleziono gli impiegati con stipendio minore o uguale a 40.000 e proietto la matricola.

Trovare gli impiegati che guadagnano più del proprio capo, mostrando matricola, nome e stipendio dell'impiegato e del capo

Prima versione

$$\Pi_{Matr,Nome,Stip,MatrC,NomeC,StipC}(\\ \sigma_{Stipendio > StipC}(\\ \rho_{MatrC,NomeC,StipC,EtaC \leftarrow Matr,Nome,Stip,Eta}(Impiegati) \\ \bowtie_{MatrC=Capo} \\ (Supervisione \bowtie_{Impiegato=Matricola} Impiegati) \\ ) \\ )$$

- Dobbiamo mettere a confronto lo stipendio di due impiegati: quello del capo e quello del suo impiegato. Saranno necessari più JOIN: la relazione *Supervisione* mi permette di stabilire chi è capo di chi. Prima faccio JOIN tra Supervisione e Impiegati per trovare le informazioni degli impiegati ($Impiegato = Matricola$), successivamente riapplico il JOIN tra la relazione ottenuta e Impiegati (again) per trovare le informazioni dei capi ($MatrC = Capo$).
- Utilizzo la ridenominazione poichè coinvolgiamo in un operatore su relazione due relazioni aventi attributi con lo stesso nome (il pensiero di porre due volte Impiegati in un'espressione algebrica, quando utilizziamo i JOIN, dovrebbe accenderci la lampadina)
- Con i JOIN ottengo una relazione i cui record contengono le informazioni di un impiegato assieme a quelle del suo capo. Gli impiegati capi, a meno che non siano supervisionati da altri capi, non saranno considerati.
- Alla relazione così ottenuta applico la selezione, per scegliere i record dove un impiegato ha lo stipendio superiore a quello del capo, e la proiezione, per proiettare gli attributi richiesti.
- VERIFICARE Le proiezioni interne prima del JOIN non sono strettamente necessarie: sono coinvolti tutti gli elementi ad eccezione dell'età (buona parte sono attributi da proiettare alla fine, altri sono JOIN)

Seconda versione

$$\Pi_{Matr,Nome,Stip,MatrC,NomeC,StipC}($$
$$\rho_{MatrC,NomeC,StipC,EtaC \leftarrow Matr,Nome,Stip,Eta}(Impiegati)$$
$$\bowtie_{MatrC=Capo\ AND\ Stipendio>StipC}$$
$$(Supervisione \bowtie_{Impiegato=Matricola} Impiegati)$$
$$)$$

La versione alternativa rinuncia alla selezione includendo una nuova condizione nel predicato del JOIN. Pongo

$$\bowtie_{MatrC=Capo\ AND\ Stipendio>StipC}$$

Il JOIN avviene tra le informazioni del dipendente e quelle del capo solo se è soddisfatta la condizione richiesta dall'esercizio (stipendio dell'impiegato superiore a quello del capo).

Esami A.A 18-19

Relazione

```
ATTORE (CodAttore, NomeAttore, CognomeAttore, AnnoNascita, NazioneNascitaA);
INTERPETAZIONE (CodAttore, CodFilm)
FILM (CodFilm, Titolo, CasaProduzione, AnnoProduzione, LuogoProduzione, NomeRegista,
CognomeRegista, Genere, NomeProduttore, CognomeProduttore, CostoFinale, IncassoTotale)
REGISTA(NomeRegista, CognomeRegista, NazioneNascitaR)
NAZIONE (Nazione, Continente, Città)
PRODUZIONE( NomeCasaProduzione, NomeAgente, CognomeAgente, Sede, Capitale)
```

Scrivere un'espressione in algebra relazionale che elenchi i nomi e cognomi degli agenti che nel 2016 hanno prodotto solo film il cui costo è stato superiore all'incasso

$$\Pi_{NA,CA}(Produzione) - \Pi_{NA,CA}($$
$$\Pi_{NCP,NA,CA}(Produzione)$$
$$\bowtie_{NCP=CP}$$
$$\Pi_{CP}(\sigma_{A=2016\ AND\ CF \leq IT}(Film))$$
$$)$$

- *solo film*, all'interno della consegna, mi fa pensare a una sottrazione: sottraggo all'insieme di tutti gli agenti quelli che hanno prodotto film con un costo finale minore o uguale all'incasso nel 2016. La differenza mi permette di ottenere gli agenti che nel 2016 hanno prodotto solo film con costo superiore all'incasso.
- Gli attributi richiesti sono *NomeAgente* e *CognomeAgente*: se eseguo la differenza le due relazioni dovranno proiettare solo queio due attributi.
- Eseguo un JOIN tra produzione e film: mi interessano esclusivamente i film prodotti nel 2016 con costo finale minore o uguale all'incasso totale, quindi applico la selezione (ricordando la *pushing selections down*.
- Applico le proiezioni alle singole relazioni coinvolte nel JOIN, ricordando la *pushing projections down*: gli attributi proiettati sono quelli necessari per il JOIN (se non li includo è come se non ci fossero) e quelli che devo proiettare alla fine (Nome e cognome dell'agente).

Scrivere un'espressione in algebra relazionale che elenchi i nomi e i cognomi dei registi che hanno diretto almeno due film di genere spionaggio nel 2000 senza Sean Connery tra gli interpreti

- Utilizziamo quanto detto con le relazione derivate assegnando a un'espressione un certo nome. In questo modo indichiamo con un solo nome un risultato intermedio evitando confusione nell'espressione finale.
- Osservo che gli unici film che mi interessano sono stati prodotti nel 2000 e sono di genere Spionaggio. Segue

$$Film2000S = \sigma_{AnnoProduzione=2000\ AND\ Genere=`Spionaggio'}(Film)$$

- Devo trovare registi che hanno fatto almeno due film con le caratteristiche dette prima e senza Sean Connery. Sicuramente sarà presente la differenza: sottraggo a tutti i film di genere spionaggio del 2000 quelli con le stesse proprietà dove è coinvolto Sean Connery come attore.

$$PrimoFilm = \pi_{N,C,CF}(Film2000S) - \pi_{N,C,CF}(\pi_{N,C,CF}(Film2000S) \bowtie Interpretazione \bowtie \pi_{CodAttore}(\sigma_{NA=`Sean'\ AND\ CognA=`Connery'}(Attore)))$$

- Devo trovare almeno due film fatti dallo stesso regista (uguaglianza di nome e cognome): farò un JOIN.
- Considero che entrambi i film presentano le stesse caratteristiche: quindi farò fare JOIN tra due blocchi uguali (quello scritto prima), applicando la ridenominazione a uno dei due (avviene un self join, devo distinguere gli attributi del primo blocco da quelli del secondo blocco per poi stabilirne l'uguaglianza)

$$SecondoFilm = \rho_{X' \leftarrow x}(\pi_{N,C,CF}(Film2000S) - \pi_{N,C,CF}(\pi_{N,C,CF}(Film2000S) \bowtie Interpretazione \bowtie \pi_{CodAttore}(\sigma_{NA=`Sean'\ AND\ CognA=`Connery'}(Attore))))$$

- Segue

$$PrimoFilm \bowtie_{CF \neq CF'\ AND\ NR=NR'\ AND\ CR=CR'} SecondoFilm$$

Scrivere un'espressione in algebra relazionale che elenchi i nomi e cognomi degli attori che hanno interpretato almeno due film di genere romantico nel 2018 e nessuno di genere avventura

- Utilizziamo quanto detto con le relazione derivate assegnando a un'espressione un certo nome. In questo modo indichiamo con un solo nome un risultato intermedio evitando confusione nell'espressione finale.
- Osservo che gli unici film che mi interessano sono quelli di genere romantico del 2018. Segue

$$Film2018R = \sigma_{AnnoProduzione=2018\ AND\ Genere=`Romantico'}(Film)$$

Considero anche i film di avventura del 2018

$$Film2018A = \sigma_{AnnoProduzione=2018\ AND\ Genere=`Avventura'}(Film)$$

- Avremo una differenza tra coloro che hanno interpretato almeno due film di genere romantico nel 2018 e coloro che hanno interpretato film di genere avventura nel 2018. Questa ci permetterà di ottenere coloro che hanno interpretato almeno due film di genere romantico nel 2018 e che non hanno mai interpretato film di genere avventura

$$DueGenereRomantico2018 - GenereAvventura2018$$

- L'espressione corrispondente di *GenereAvventura2018* non è difficile da trovare

Scrivere un'espressione in algebra relazionale che elenchi i nomi e cognomi dei registi che hanno girato nel 2019 film sempre con lo stesso produttore, anche per case di produzione diverse

- Utilizziamo quanto detto con le relazione derivate assegnando a un'espressione un certo nome. In questo modo indichiamo con un solo nome un risultato intermedio evitando confusione nell'espressione finale.
- Osservo che gli unici film che mi interessano sono quelli di genere romantico del 2019. Segue

$$Film2019 = \sigma_{AnnoProduzione=2019}(Film)$$

- Svolgo una differenza tra i registi che hanno girato film nel 2019 e i registi che hanno girato film nel 2019 con produttori diversi. Ottengo così i registi che hanno girato film nel 2019 sempre con lo stesso produttore.

$$Registi2019 - Registi2019pd$$

- Il primo elemento è semplice

$$Registi2019 = \pi_{NR,CR}(Film2019)$$

- Per il secondo dobbiamo fare un self join tra film: avremo una tupla se è possibile associare a un film di un certo regista un altro diretto dallo stesso con un produttore diverso

$$Registi2019pd = \pi_{CF,NR,CR,NP,CP}(Film2019) \bowtie_{CF \neq CF' \ AND \ NR=NR' \ AND \ CR=CR' \ AND \ NP \neq NP' \ AND \ CP \neq CP'} \rho_{X' \leftarrow X}(\pi_{CF,NR,CR,NP,CP}(Film2019))$$

  - I film sono diversi (CodFilm diverso)
  - I registi sono gli stessi (Nome e cognomi registi uguali)
  - I produttori sono diversi (quindi non avranno stesso nome e cognome)

Scrivere un'espressione in algebra relazionale che elenchi i cognomi dei produttori che hanno lavorato negli ultimi 5 anni in almeno due case produttrici diverse, ma con lo stesso regista

- Utilizziamo quanto detto con le relazione derivate assegnando a un'espressione un certo nome. In questo modo indichiamo con un solo nome un risultato intermedio evitando confusione nell'espressione finale.
- Osservo che gli unici film che mi interessano sono quelli degli ultimi 5 anni. Segue

$$Film5 = \pi_{CF,NR,CR,NCP,CP}(\sigma_{AnnoProduzione>2015}(Film))$$

- Faccio un self Join (segue adozione della ridenominazione per una delle due relazioni)

  $\pi_{CP}(Film5 \bowtie_{CF \neq CF' \ AND \ NR=NR' \ AND \ NCP \neq NCP' \ AND \ CP=CP' \ AND \ NP=NP'} \rho_{X' \leftarrow X}(Film5))$

  - I due film hanno CodFilm diverso
  - Presentano lo stesso regista, quindi Nomi e Cognomi del regista sono gli stessi
  - Le case di produzione dei due film sono diverse
  - Il produttore è lo stesso (quindi cognome e nome del produttore è lo stesso in entrambi i film)

  Di tutto questo proietto i cognomi dei produttori e basta (CP)

Scrivere un'espressione in algebra relazionale che elenchi i titoli di film che, negli anni 30, sono stati prodotti negli Stati Uniti ma diretti da registi di origine tedesca

- Utilizziamo quanto detto con le relazione derivate assegnando a un'espressione un certo nome. In questo modo indichiamo con un solo nome un risultato intermedio evitando confusione nell'espressione finale.
- Osservo che gli unici film che mi interessano sono quelli prodotti negli anni 30. Segue

  $$Film30 = \sigma_{AnnoProduzione>1929 \ AND \ AnnoProduzione<1940}(Film)$$

- I film che voglio trovare sono prodotti negli Stati Uniti e diretti da registi di origine tedesca:
  - Per il primo elemento faccio Join tra Film e Nazione in modo da individuare a quale stato appartiene una certa città
  - Per il secondo elemento faccio Join tra Film e Regista, tabella che contiene la nazione di nascita di ogni singolo regista.
- Segue che mi interessano solo le città degli Stati Uniti

  $$CittaUSA = \pi_{Citta}(\sigma_{Nazione='Usa'}(Nazione))$$

  e i registi di origine tedesca

  $$RegistiTedeschi = \pi_{NR,CR}(\sigma_{NNR='Germania'}(Regista)$$

- Segue (con le proiezioni legate a Film)

  $$\pi_T(\pi_{NR,CR,T}(\pi_{LP,T,NR,CR}(Film30) \bowtie_{LP=Citta} CittaUSA) \bowtie RegistiTedeschi)$$

  Tenere sempre conto della proiezione di attributi necessari solo per fare Join

B — Esempi di espressioni in calcolo relazionale

Esami A.A 18-19

Relazione

```
ATTORE (CodAttore, NomeAttore, CognomeAttore, AnnoNascita, NazioneNascitaA);
INTERPETAZIONE (CodAttore, CodFilm)
FILM (CodFilm, Titolo, CasaProduzione, AnnoProduzione, LuogoProduzione, NomeRegista,
CognomeRegista, Genere, NomeProduttore, CognomeProduttore, CostoFinale, IncassoTotale)
REGISTA(NomeRegista, CognomeRegista, NazioneNascitaR)
NAZIONE (Nazione, Continente, Città)
PRODUZIONE( NomeCasaProduzione, NomeAgente, CognomeAgente, Sede, Capitale)
```

Scrivere un'espressione in calcolo relazionale che elenchi i cognomi dei produttori che hanno lavorato negli ultimi 5 anni in almeno due case produttrici diverse, ma con lo stesso regista

{CP: cp | Film(CF: cf, AP: ap, NCP: ncp, NR: nr, CR: cr, CP: cp,...) ∧ Film(CF: cf', AP: ap', NCP: ncp', NR: nr, CR: cr, CP: cp,...) ∧ cf≠cf'∧ ncp≠ncp' ∧ ap>'2015' ∧ ap'>'2015'}

- Nella *target list* pongo soltanto il cognome del produttore (CP)
- Pensiamo alla corrispondente espressione in algebra relazionale: per ottenere il risultato dobbiamo per forza fare un self join fra tuple di *Film*.
- Poniamo in f:
  - Due schemi di *Film*: presentano la stessa struttura ma le variabili assegnati ai vari attributi presentano delle differenze
    - I codici film (CF) sono assegnati a variabili diverse (cf e cf'). Successivamente porremo come condizione $cf \neq cf'$
    - Stessa cosa per il nome della casa di produzione (NCP). Successivamente pongo che $ncp \neq ncp'$
    - Nome/Cognome dei registi (NR e CR) sono assegnati alle stesse variabili (cr e nr). Questo, senza porre ulteriori condizioni, mi permette di dire che i due film sono stati condotti dallo stesso regista.
    - Stessa cosa per il cognome del produttore (CP), in entrambi i casi si ha la variabile cp.
  - Pongo sia l'anno di produzione del primo film che quello del secondo film maggiore di 2015 ($ap > 2015, ap' > 2015$). Ci interessano soltanto i film degli ultimi cinque anni!

Scrivere un'espressione in calcolo relazionale che elenchi i nomi e cognomi dei registi che hanno diretto almeno due film di genere "spionaggio" nel 2000 senza Sean Connery tra gli interpreti

{NR:nr, CR:cr | Film (CF:cf', NR:nr, CR:cr, G:g, A:a,..) $\wedge$ Film (CF:cf, NR:nr, CR:cr, G:g, A:a,...) $\wedge$ g='Spionaggio' $\wedge$ a='2000' $\wedge$ cf $\neq$ cf' $\wedge$ $\neg\exists$ na, ca . (Interpretazione(CF:cf', CodA:coda) $\wedge$ Interpretazione(CF:cf, CodA:coda) $\wedge$ Attore (CodA:coda, NA:na, CA:ca,...) $\wedge$ na='Sean' $\wedge$ ca='Connery')}

- Nella *target list* pongo il nome e il cognome del regista
- Osserviamo la corrispondente espressione algebrica: non ho soltanto dei Join ma anche delle differenze. Quest'ultimo operatore comporta l'introduzione di quantificatori esistenziali e/o universali nell'espressione in calcolo relazionale.
- Poniamo in f:
  - Due schemi di *Film*. Osservo che
    - i codici dei film (CF) sono assegnati a variabili diverse (cf e cf'). Successivamente diremo che $cf \neq cf'$ (associamo due film diversi)
    - il nome e il cognome dei registi (NR e CR) dei due film sono gli stessi (nr e cr)
    - il genere (G) è lo stesso in entrambi i film (g). Diremo che $g =$ '$Spionaggio'$
    - L'anno di produzione (A) dei due film è lo stesso (a). Diremo che $a =$ '$2000'$.
  - Utilizziamo un quantificatore esistenziale due volte (sopra trovate la versione semplificata posta dalla prof, immaginate che ci siano due $\exists$ con più parentesi tonde rispetto a quelle presenti).
    - Quanto presente all'interno delle parentesi è valido solo con certi valori di na e ca. Precisamente abbiamo...
    - ... due schemi della relazione *Interpretazione*: uno è associato al primo film (cf'), l'altro al secondo film (cf).
    - ... uno schema della relazione *Attore*: osservo che l'attore associato ai due schemi di Interpretazione è lo stesso ($coda$)
    - concludo affermando che il nome dell'attore (na) è *Sean* e il cognome (ca) *Connery*.

    Tutto questo è posto dopo il simbolo di negazione: se le proprietà di questo predicato sono verificate allora l'intero predicato f non è verificato.
  - Segue che il predicato f è valido solo se i film individuati non presentano tra gli interpreti *Sean Connery*!

Scrivere un'espressione in calcolo relazionale che elenchi i titoli di film che, negli anni '30, sono stati prodotti negli Stati Uniti ma diretti da registi di origine tedesca

{T: t | Film (T: t, AP: ap, NR: nr, CR: cr, CP: c,...) $\wedge$ Nazione (N: n, Cn:cn, C: c) $\wedge$ Regista(NR: nr, CR: cr, NNR: nnr) $\wedge$ n='USA' $\wedge$ nnr='Germania' $\wedge$ ap>'1929' $\wedge$ ap< '1940'}

- Nella *target list* pongo soltanto il titolo del film.
- Nella seconda parte, dove ho la formula f, devo porre una serie di condizioni:
  - Gli schemi delle relazioni coinvolte. In questo caso abbiamo le relazioni *Film*, *Nazione*, *Regista*. La prima contiene i film, la seconda le nazioni in cui si trovano le città, la terza la nazione di origine dei registi.
  - Gli schemi sono validi solo ponendo certe variabili: le imposto in modo tale che il nome/-cognome regista sia lo stesso sia in film che in regista (nr, cr), stessa cosa per la città di produzione in Film e Nazione (c).

– Successivamente pongo che la nazione della città (n) sia *USA* e che l'anno di produzione (ap) sia compreso tra 1929 e 1940 (estremi esclusi)

C — Esempi di esercizi sulle dipendenze funzionali

C.1 Qualche nozione

- Se parto da una relazione e una spiegazione (come nei seguenti esercizi) ricostruisco i legami semantici fra i vari attributi e scrivo le dipendenze trovate. A quel punto dovrei ottenere una copertura minimale.
- Se si manifestano relazioni aventi la stessa chiave le unisco per formarne una sola
- Se ho DF del tipo $X \to Y$ e $Y \to X$ fondo gli schemi (tra i due ne elimino uno)
- Al termine verifico se esiste almeno una relazione che contiene nella chiave la stessa della relazione originaria. In caso contrario creo un'ulteriore relazione contenente solo la chiave della relazione originaria.

C.2 Esercizio 1

Si consideri la relazione che segue contenente informazioni relative alle varie sedi del museo diffuso di una città.

Guida (CodSede, NomeSede, Indirizzo, ZonaCittà, GiornoChiusura, OrarioChiusura, TipoMuseoSede, Autore, BiografiaAutore, NomeOpera, NumeroSala, CustodeSala).

Un autore può avere opere esposte in varie sedi. Le sale sono numerate all'interno di ogni sede del museo. Il nome delle opere è unico nel museo. Un custode può essere responsabile di più di una sala.

- **Individuare la chiave e tutte le dipendenze funzionali non banali**:
  - Osservazioni:
    * La frase *Un autore può avere opere esposte in varie sedi* esclude l'esistenza di una dipendenza funzionale dove la sede dipende dall'autore
    * La frase *Le sale sono numerate all'interno di ogni sede del museo* stabilisce che il numero sala non può essere da solo, sia nel lato sinistro che nel lato destro. Due sedi possono avere sale con le stesse numerazioni.
    * La frase *Il nome delle opere è unico nel museo* ci da la certezza che stiamo parlando di una certa opera realizzata da un certo autore.
    * La frase *Un custode può essere responsabile di più di una sala* indica che un custode si può occupare di più di una sala. Escludo quindi una dipendenza funzionale in cui il custode determina la sala.
  - Quindi individuiamo:

* Qua poniamo le informazioni riguardanti una sede, raggiungibili in modo univoco attraverso *CodSede*

  $CodSede \rightarrow NomeSede, Indirizzo, GiornoChiusura, OrarioChiusura, TipoMuseoSede$

* La zona della città, pur essendo un dato relativo alla sede, non risulta direttamente dipendente dal codice della sede. La collocazione della sede in una certa zona della città si intuisce dall'indirizzo della sede. Segue

  $$Indirizzo \rightarrow ZonaCitta$$

* Gli attributi riguardanti le informazioni sull'autore sono *Autore* e *BiografiaAutore*. L'autore determina la biografia, segue

  $$Autore \rightarrow BiografiaAutore$$

* L'opera è realizzata da un certo autore ed è collocata in una certa sala. Abbiamo più sedi dove le numerazioni possono coincidere, segue

  $$NomeOpera \rightarrow CodSede, NumeroSala, Autore$$

* Ad ogni sala è associato in modo univoco un certo custode. Sapendo che le numerazioni possono coincidere in più sedi segue

  $$CodSede, NumeroSala \rightarrow CustodeSala$$

* La chiave, non indicata in anticipo, è *NomeOpera*: questo è l'unico attributo che non compare nel lato destro delle varie dipendenze funzionali. La chiave è verificabile ricorrendo al calcolo della chiusura transitiva $NomeOpera^+$ rispetto all'insieme delle DF trovate prima.

- **Verificare se *Guida* è in 3NF e, se non lo è, portarla in 3NF**:
  - Risulta facilmente intuibile che non siamo in 3NF: data una qualunque dipendenza funzionale

    $$X \rightarrow Y$$

    X non è mai superchiave e in Y si individuano sempre attributi che non appartengono alla chiave candidata.
  - Costruisco una relazione a partire da ogni singola DF introdotta prima. Otteniamo quanto segue

    Sede (CodSede, NomeSede, Indirizzo, GiornoChiusura, OrarioChiusura, TipoMuseoSede)
    Zona(Indirizzo , ZonaCittà)
    Autore(Autore , BiografiaAutore)
    Opera(NomeOpera , CodSede, NumeroSala, Autore)
    Custode(CodSede, NumeroSala CustodeSala)

  - Adesso si ha la 3NF, poichè X è sempre superchiave.
  - La chiave candidata, costituita da una solo attributo, non viene divisa dalla creazione delle varie relazioni. Segue che non è necessario creare un'ulteriore relazione contenente la chiave!

C.3 Esercizio 2

Si consideri la relazione che segue contenente informazioni relative alle prenotazioni di un albergo.

Mostre (TitoloMostra, Città, Indirizzo, ZonaCittà, AnnoMostra, DataInizio, DataFine, GiornoChiusura, OrarioApertura, ArgomentoMostra, CommentoMostra, Autore, BiografiaAutore, NomeOpera)

Un autore può avere opere esposte in varie mostre, ma di argomento differente. Una mostra può contenere opere di autori diversi e più di una dello stesso autore. Il nome delle opere è unico. La stessa mostra può avere luogo in più di una città ma in anni diversi.

- **Individuare la chiave e tutte le dipendenze funzionali non banali**:
  - Osservazioni:
    - La frase *Un autore può avere opere esposte in varie mostre, ma di argomento differente* stabilisce che non possiamo stabilire una dipendenza del tipo $Autore \to TitoloMostra$.
    - La frase *Una mostra può contenere opere di autori diversi e più di una dello stesso autore* stabilisce che non possiamo avere DF del tipo $TitoloMostra \to Autore$.
    - La frase *Il nome delle opere è unico* stabilisce che globalmente ogni opera ha un nome diverso. Posso avere in mostre diverse la stessa opera ma non possono esistere due opere diverse aventi lo stesso nome.
    - La frase *La stessa mostra può avere luogo in più di una città ma in anni diversi* stabilisce che una mostra può tenersi in più città, ma che queste città non possono ospitare la mostra nello stesso anno.
  - Quindi individuiamo:
    - Qua poniamo le informazioni riguardanti una mostra svolta in una certa città, raggiungibili in modo univoco attraverso *TitoloMostra* e *AnnoMostra*. Ricordiamo che la mostra può avere luogo in più città, ma in anni diversi.

      $$TitoloMostra, AnnoMostra \to Citta, Indirizzo, DataInizio, DataFine, GiornoChiusura, OrarioApertura$$

    - La zona della città, pur essendo un dato relativo alla mostra non risulta direttamente dipendente da *TitoloMostra* e *AnnoMostra*. La collocazione della mostra in una certa zona della città si intuisce dall'indirizzo della mostra e dalla città. Segue

      $$Citta, Indirizzo \to ZonaCitta$$

    - Gli attributi riguardanti le informazioni sull'autore sono *Autore* e *BiografiaAutore*. L'autore determina la biografia, segue

      $$Autore \to BiografiaAutore$$

    - Gli attributi riguardanti le informazioni sul contenuto della mostra sono *TitoloMostra*, *ArgomentoMostra* e *CommentoMostra*. Gli ultimi due dipendono dal primo, quindi

      $$TitoloMostra \to ArgomentoMostra, CommentoMostra$$

    - Il nome delle opere è unico globalmente, quindi sono certo che

      $$NomeOpera \to Autore$$

    - La chiave, non indicata in anticipo, è *TitoloMostra, AnnoMostra, NomeOpera*: questi attributi non compaiono nei lati destri delle varie dipendenze funzionali. La chiave è verificabile ricorrendo al calcolo della chiusura transitiva $TitoloMostra, AnnoMostra, NomeOpera^+$ rispetto all'insieme delle DF trovate prima.

- **Verificare se *Mostre* è in 3NF e, se non lo è, portarla in 3NF**:
  - Risulta facilmente intuibile che non siamo in 3NF: data una qualunque dipendenza funzionale

    $$X \to Y$$

    X non è mai superchiave e in Y si individuano sempre attributi che non appartengono alla chiave candidata.
  - Costruisco una relazione a partire da ogni singola DF introdotta prima. Otteniamo quanto segue

    Titolo(TitoloMostra, AnnoMostra, Città, Indirizzo, DataInizio, DataFine, GiornoChiusura, OrarioApertura)
    Zona(Città , Indirizzo, ZonaCittà)
    Autore(Autore, BiografiaAutore)
    Documentazione(TitoloMostra , ArgomentoMostra, CommentoMostra)
    Opera(NomeOpera, Autore)

  - Non abbiamo ancora la 3NF poichè la chiave candidata non è presente nella sua interezza in nessuna delle chiavi delle varie relazioni. Segue la necessità di creare un'ulteriore relazione, la seguente

    $$Chiave(TitoloMostra, AnnoMostra, NomeOpera)$$

    Adesso siamo in 3NF!