

Alessandro Gerotto

Università degli studi di Udine

Docente *Angelo Montanari*

Anno accademico 2023-24

Corso di Basi di Dati

Introduzione ai sistemi di basi di dati.....	9
Caratteristiche distintive di un database.....	9
Persistenza dei dati.....	9
Mole dei dati.....	9
Globalità dei dati.....	9
Caratteristiche generali di un database.....	9
Facilità d'uso (indipendenza dei dati).....	9
Efficienza delle operazioni di accesso ai dati (strutture di indicizzazione e ottimizzazione delle interrogazioni);..	10
Efficacia (convenienza).....	10
Perché non usare il file system?.....	10
Sistemi di basi di dati o programmi.....	10
Sistemi di basi di dati o sistemi operativi.....	11
Cos'è un database e cos'è un DBMS.....	11
Componenti di un database.....	11
Il gestore.....	11
L'amministratore.....	12
Utenti.....	12
Astrazioni e livelli.....	12
Modelli dei dati.....	13
Esempio: filiale di una banca.....	13
Modello entità-relazioni ER.....	14
I costrutti fondamentali.....	14
Le entità.....	14
Entità deboli.....	14
Gli attributi.....	15
Chiavi o identificatori interni.....	15
Il dominio di un attributo e il valore NULL.....	15
Attributi semplici e composti.....	16
Attributi a singolo valore o multivалore.....	16
Attributi obbligatori o opzionali.....	16
Attributi primitivi o derivati.....	17
Modelli basati sui valori.....	17
Le relazioni.....	17
Le molteplicità.....	18
Vincoli di partecipazione e rapporto di cardinalità.....	18
Relazioni ricorsive.....	19
Attributi di relazione.....	20
Relazioni ternarie o di grado superiore.....	20
Soluzione 1.....	20
Soluzione 2: la reifica.....	21
Soluzione 3: evitarle.....	21
Costrutti aggiuntivi.....	22
La generalizzazione.....	22
Tipi di generalizzazione.....	22
Le generalizzazioni attribute-defined.....	22
Il problema dell'ereditarietà multipla.....	22
Tipi unione (o categorie).....	22
Esercizi sul ER.....	24
I cicli e i vincoli d'integrità.....	25
Altro esempio di ciclo non problematico.....	25
Esercizio 22 luglio 2020.....	26

Esercizio 4 settembre 2019 (lez. 16).....	27
Considerazioni.....	27
Modello relazionale.....	28
Problematiche.....	28
Discrimine tra presenza o assenza di una ridondanza.....	28
La soluzione.....	29
Costrutti fondamentali.....	29
Dominio, attributo, relazione, tupla.....	29
Attributi.....	29
Relazioni.....	30
Schema e istanza di una base di dati.....	30
Intra-relazionali.....	30
Inter-relazionali.....	30
Un esempio di schema relazionale.....	31
Chiavi esterne.....	31
Logical design.....	32
Le fasi.....	32
1. Analisi delle ridondanze.....	32
Esempio di analisi delle ridondanze.....	33
2. Rimozione delle generalizzazioni.....	34
3.1. Partizionamento delle entità.....	34
3.2. Partizionamento delle relazioni.....	35
3.3. Unione delle entità.....	35
3.4. Rimozione attributi multivaleore.....	35
4. Scelta della chiave primaria.....	36
Esempio.....	36
Modello ER.....	36
Tabella dei volumi e delle operazioni.....	36
Traduzione di schemi ER in schema relazionale.....	38
Entità.....	38
Relazioni.....	38
Casi 1 a 1.....	38
Casi 1 a N.....	39
Casi N a N.....	40
Forme normali e normalizzazioni.....	41
Esempio di relazione con anomalie.....	41
Perché non va bene?.....	41
Le dipendenze funzionali.....	41
Le dipendenze funzionali banali e non.....	42
Assiomi di Armstrong.....	42
Boyce–Codd Normal Form (BCNF).....	42
Note.....	42
Esempio.....	42
Scomposizioni in BCNF problematiche: Lossless decomposition.....	43
Condizione sufficiente per avere una lossless decomposition.....	43
Preservazione delle dipendenze.....	44
La Terza Forma Normale.....	44
Decomposizione in 3NF.....	45
Esempio.....	45
Algebra relazionale.....	47
Limiti dell'algebra relazionale.....	47
Le operazioni di base.....	48

1. Operazioni di selezione (σ).....	48
proiezione (π).....	48
2. Operazioni di rinomina (ρ).....	48
3. Operazione di unione insiemistica (\cup).....	49
Esempio.....	49
4. Operazione di differenza insiemistica (\setminus o -).....	49
Esempio.....	49
5. Operazione di prodotto cartesiano (\times).....	50
Esempio.....	50
Il conflitto di nomi negli attributi.....	51
Operazioni derivate.....	53
1. Operazioni di theta join.....	53
Caso particolare: equi join.....	53
Esempio.....	53
Esempio 2.....	54
2. Operazioni di natural join.....	54
Caso particolare senza attributi in comune.....	54
Caso particolare con tutti gli attributi in comune.....	55
Esempio di natural join.....	55
3. Operazione di divisione.....	55
Esempio.....	55
4. Operazione di semi join sinistro (e destro).....	56
Esempio.....	56
Operazioni addizionali.....	57
1. Operazione di proiezione generalizzata.....	57
2. Funzioni aggregate.....	57
Funzioni aggregate max e min.....	57
3. Operazione di outer join.....	57
Esempio.....	58
Esercizi sull'algebra relazionale.....	59
Esercizio 1.....	59
Esercizio 2.....	59
Esercizio 3.....	60
Esercizio 4. (7-2-19).	60
Esercizio 5. (8-9-17).	63
Calcolo relazionale.....	66
Le interrogazioni in linguaggio SQL.....	67
Il blocco fondamentale SELECT-FROM-WHERE.....	67
SELECT.....	67
FROM.....	67
WHERE.....	67
Esempi.....	67
Sottoblocco SELECT-FROM e uso di DISTINCT.....	68
Notazione puntata e nomi di attributo ambigui.....	68
Alias e operazione di rinomina.....	68
Espressioni Booleane nella clausola WHERE.....	69
Le operazioni insiemistiche.....	69
Le interrogazioni nidificate.....	70
L'operatore IN.....	70
Interrogazioni nidificate correlate e non correlate.....	71
Esempio di violazione delle regole di visibilità.....	71
Interrogazioni nidificate correlate e uso di alias.....	71

La funzione Booleana EXISTS.....	72
INTERSECT ed EXCEPT via EXISTS.....	72
L'operatore CONTAINS (rimosso).....	73
La funzione Booleana UNIQUE.....	73
Le operazioni di join.....	73
Inner join.....	74
Outer join.....	74
Funzioni aggregate in SQL.....	74
Interrogazioni inconsistenti (mismatch).....	75
Interrogazioni con raggruppamento.....	75
Interrogazioni sintatticamente scorrette.....	75
Predicati sui gruppi.....	75
Condizioni WHERE vs. condizioni HAVING.....	76
Relazioni come valori.....	76
Altro.....	76
Usare tuple di attributi anziché singoli attributi.....	76
Operatore LIKE e pattern matching.....	77
Operatore ORDER BY.....	77
Gestione dei valori nulli.....	77
Esercizi sulle interrogazioni SQL.....	79
Esercizio 1. (1-7-19).....	79
Esercizio 2 (31-1-2014).....	80
Esercizio 3 (8-9-2017).....	81
Il linguaggio SQL: le viste.....	83
Esempio di vista.....	83
Operazioni di modifica sulle viste.....	83
Quando una vista può essere modificata?.....	84
Opzione LOCAL o CASCADE.....	84
Esempio.....	84
Tecnologia di un Database Server (centralizzato).....	86
Componenti di un database server.....	86
Le transazioni.....	86
Le istruzioni di begin e end.....	86
Le istruzioni di commit e abort.....	87
Un esempio di transazione in SQL.....	87
Le anomalie.....	88
0. Conflitti scrittura-scrittura.....	88
1. Perdita di aggiornamento.....	88
2. Lettura sporca.....	89
3. Aggiornamento fantasma.....	89
4. Letture inconsistenti.....	90
5. Inserimento fantasma.....	90
Proprietà acide delle transazioni.....	90
Atomicità (Atomicity).....	90
Consistenza (Consistency).....	91
Isolamento (Isolation).....	91
Standard SQL vs PostgreSQL.....	92
Persistenza (Durability).....	92
Gestione del buffer.....	92
Componenti di un DBMS.....	92
Organizzazione del buffer.....	93
Politiche di gestione del buffer.....	93

Direttorio e variabili di stato.....	93
Primitive per la gestione del buffer.....	93
Pre-fetching e pre-flushing.....	94
DBMS e file system.....	94
Esercizi su livelli di isolamento.....	95
Esercizio 1.....	95
Esercizio 2.....	96
Esercizio 3.....	97
Controllo della concorrenza.....	98
La nozione di schedule.....	98
La nozione di schedule seriale.....	98
La nozione di schedule serializzabile.....	99
Equivalenza di vista (VSR).....	99
Esempio.....	99
Limiti della tecnica VSR.....	99
Equivalenza rispetto ai conflitti (CSR).....	100
Il grafo dei conflitti.....	100
Limiti della tecnica CSR.....	100
Legame tra VSR e CSR.....	100
Locking a due fasi (2PL).....	101
Vincoli che caratterizzano le transazioni ben formate.....	101
Il funzionamento.....	101
Le tabelle dei conflitti.....	102
Legame tra 2PL e CSR.....	102
Dimostrazione punto 1.....	102
Dimostrazione punto 2.....	102
Locking a due fasi stretto (2PL stretto).....	103
Inserimento fantasma: lock di predicato.....	104
Controllo basato sui timestamp (TS).....	104
Esempio.....	104
Limiti del metodo TS.....	104
Quadro complessivo.....	105
Meccanismi per la gestione dei lock.....	105
Granularità dei lock.....	106
Il lock gerarchico.....	106
Deadlock (stallo).....	106
Soluzioni.....	106
Come scegliere le transazioni da uccidere.....	106
Nella pratica.....	107
Gestione dell'affidabilità.....	108
Architettura del gestore dell'affidabilità.....	108
I compiti del controllore dell'affidabilità.....	108
Organizzazione del file di log.....	109
Struttura dei record.....	109
Le primitive undo e redo.....	109
Proprietà di idempotenza.....	110
L'operazione di checkpoint.....	110
L'operazione di dump.....	110
Le regole.....	110
Write-ahead-log (WAL).....	110
Commit-precedenza.....	111
Le politiche.....	111

Gestione dei guasti.....	111
Come funzionano le procedure di ripresa?.....	112
Esercizi schedule.....	113
Esercizio 1.....	113
Esercizio 2.....	114
Esercizio 3. (22.01.2019).....	115
Esercizio 4. (3-7-2017).....	117
Organizzazione Fisica dei Dati.....	119
Record di lunghezza fissa e variabile.....	119
Esempio.....	119
Organizzazione dei record in blocchi.....	120
Record spanned e unspanned.....	121
Dimensionamento dei file.....	121
Allocazione dei blocchi di un file su disco.....	121
File header e ricerca dei record.....	122
Operazioni.....	122
Operazioni record-at-a-time.....	122
Operazioni set-at-a-time.....	122
Organizzazione dei file e metodi di accesso.....	122
File di record non ordinati (heap file).....	123
Inserimenti.....	123
Ricerche.....	123
Cancellazioni.....	123
Un altro problema.....	123
File di record ordinati (file ordinati).....	123
Vantaggi.....	124
File ordinati e ricerca binaria.....	124
Ricerca.....	125
Inserimenti.....	125
Modifica.....	125
Ricerca dei file con funzioni di hashing.....	126
Internal Hashing.....	126
Esempio.....	126
Le collisioni.....	127
External hashing.....	127
Come scegliere la grandezza dei file.....	128
Dynamic hashing.....	128
Extendible hashing.....	129
Linear hashing.....	130
Ricerca dei file con strutture ad indice per file.....	132
Indici di singolo livello.....	132
1. Indici primari.....	133
Esercizio esame.....	133
Problematiche degli indici primari.....	133
2. Indici di clustering.....	133
Problematiche degli indici di clustering.....	133
3. Indici secondari.....	134
Indici primari vs indici secondari.....	134
Esercizio esame.....	134
Indici multilivello.....	135
1. Indici multilivello statici (alberi).....	135
Esercizio esame.....	135

2. Indici multilivello dinamici (B-alberi).....	136
BST: Binary Search Tree.....	136
B-alberi.....	136
Proprietá dei BTree.....	136
Altezza e altezza massima.....	137
Operazioni sui Btree.....	137
Operazioni di ricerca.....	137
Operazioni di inserimento.....	137
Esempio di inserimento.....	138
Operazioni di cancellazione.....	139
Esercizio esame.....	140
B+ alberi.....	141
Proprietá dei BTree.....	141
La struttura dei nodi foglia.....	141
Esercizio esame.....	142
Algoritmo di inserimento.....	143
B+ alberi vs B alberi.....	144

Introduzione ai sistemi di basi di dati

Caratteristiche distintive di un database

Persistenza dei dati

Per garantire la persistenza dei dati in un database, i dati su cui si opera devono esistere sia **prima** che **dopo** l'interazione dell'utente. Per avverare ciò la base di dati risiede in modo stabile nella **memoria secondaria** e lavora dinamicamente con la memoria principale. Inoltre, esiste in modo **indipendente** dai programmi che interagiscono con essa per ottenere le informazioni di interesse o per aggiornarla attraverso inserimenti, cancellazioni e modifiche.

Mole dei dati

La gestione di **grandi quantità di dati** implica una quantità di informazioni che supera la capacità della memoria primaria disponibile. Il sistema deve quindi prendere **decisioni intelligenti** su quali dati caricare e mantenere nella memoria primaria (non è possibile caricare tutto il database in memoria principale), e deve gestire efficientemente i **frequenti trasferimenti** di dati tra memoria secondaria e memoria primaria (rispondendo a domande come: "che dati carico in memoria principale? per quanto tempo li mantengo? quando li scarico?...").

È importante notare che **nonostante i progressi** nella capacità delle memorie a stato solido utilizzate come memoria primaria, il problema dei trasferimenti di dati **persiste**. Questo è dovuto al **continuo aumento** delle dimensioni delle basi di dati nel corso degli anni, principalmente a causa della crescente complessità dei dati da gestire, come immagini, audio, filmati e così via.

Globalità dei dati

I dati sono di interesse per una **pluralità di utenti** e programmi, quindi è essenziale **disciplinarene l'utilizzo**. È necessario specificare e gestire i **diritti di accesso** ai dati, assicurando al contempo la protezione della privacy dei **dati sensibili**. Inoltre, è fondamentale risolvere eventuali **conflicti** (inconsistenze) che possono sorgere tra utenti diversi che desiderano **operare contemporaneamente** sugli stessi dati (a.e. la prenotazione di uno stesso posto per un volo aereo da parte di più utenti contemporaneamente).

Caratteristiche generali di un database

Facilità d'uso (indipendenza dei dati)

L'interazione con un database avviene a un **livello di astrazione** che corrisponde al modello dei dati utilizzato, il quale nella maggioranza dei casi è il **modello dei dati relazionale**. In questo modello, i dati sono visualizzati e organizzati come una

serie di **tabelle**, anche se sotto il livello di astrazione, nella base di dati, tali tabelle sono implementate come **file** e le operazioni eseguite su di esse sono implementate attraverso **operazioni di base** come scansioni, ordinamenti e così via.

Questo approccio consente di eseguire operazioni a un livello astratto, noto come **modello logico**, ignorando i dettagli del modello fisico sottostante. Ciò si traduce in un concetto fondamentale chiamato "indipendenza dei dati", dove le operazioni possono essere eseguite senza dover preoccuparsi dei dettagli implementativi della base di dati.

Efficienza delle operazioni di accesso ai dati (strutture di indicizzazione e ottimizzazione delle interrogazioni);

È di fondamentale importanza che la base di dati sia **efficiente** sia

- nell'uso quotidiano (ricerche, aggiunte, cancellazioni, ...)
- nei trasferimenti tra memoria primaria e secondaria (limitati il più possibile per garantire prestazioni ottimali).

Per soddisfare questa esigenza, vengono impiegate strutture di indicizzazione che consentono di organizzare e memorizzare i dati in modo ottimizzato, facilitando l'accesso rapido e efficiente ai dati richiesti.

Efficacia (convenienza)

Affinché una base di dati venga utilizzata, deve essere **efficace**.

Tuttavia, in alcuni contesti in cui è necessario avere anche una copia cartacea, una base di dati potrebbe **non risultare utile affatto**. In queste situazioni, la necessità di una copia cartacea può essere legata a specifiche esigenze normative, pratiche o di sicurezza. Ad esempio, in alcuni ambiti legali o amministrativi, potrebbe essere richiesto avere una documentazione cartacea come prova tangibile o per scopi di archiviazione a lungo termine. In tali casi, la base di dati può essere complementare alla documentazione cartacea o utilizzata per generare rapporti e analisi, ma potrebbe non sostituire completamente la necessità di una copia cartacea.

Pertanto, l'efficacia di una base di dati dipende dalla sua **capacità di soddisfare le esigenze** specifiche del contesto in cui viene utilizzata, tenendo conto di fattori come la praticità, la conformità normativa e le preferenze degli utenti.

Perché non usare il file system?

Ci sono diverse ragioni per preferire l'uso di un sistema di gestione di database (DBMS) rispetto al file system per la gestione dei dati, specialmente in contesti complessi come le applicazioni bancarie. Ecco alcuni confronti utili:

Sistemi di basi di dati o programmi

Nei sistemi di basi di dati, i dati sono organizzati in modo strutturato e sono gestiti da un DBMS che offre funzionalità avanzate come **l'indipendenza dei dati**, **l'integrità referenziale**, **la gestione delle transazioni** e **la sicurezza**. Al contrario, quando si utilizza il file system, i dati possono essere salvati in **file non strutturati** o semistrutturati, rendendo **complesso** l'accesso e la gestione dei dati in modo **coerente e sicuro**.

Sistemi di basi di dati o sistemi operativi

Mentre i sistemi operativi gestiscono principalmente l'hardware e le risorse di sistema, i sistemi di basi di dati si concentrano sulla gestione dei dati stessi. I DBMS offrono **funzionalità specifiche** per la gestione dei dati, come la **query**, la **modifica**, **l'aggiornamento** e la **sicurezza dei dati**, che non sono disponibili nei sistemi operativi.

Ci sono diversi vantaggi nell'utilizzare un DBMS rispetto al file system:

- **ridondanza ed inconsistenza** dei dati: un DBMS può garantire la coerenza dei dati attraverso vincoli di integrità e transazioni atomiche, riducendo il rischio di ridondanza e inconsistente nei dati.
- **difficoltà di accesso ai dati**: un DBMS offre un'interfaccia strutturata e standardizzata per accedere e manipolare i dati, riducendo la necessità di sviluppare (tanti) programmi ad hoc per l'accesso ai dati (filtro sesso femminile, filtro sesso femminile + età < 45, filtro femminile + età < 45 + residenti nel comune x, ...).
- **disomogeneità dei dati**: un DBMS consente di definire uno schema di dati coerente e uniforme per l'intero sistema, facilitando la gestione e l'integrazione dei dati provenienti da diverse fonti.
- **problemi di integrità dei dati**: I DBMS offrono meccanismi per garantire l'integrità dei dati attraverso vincoli di integrità e transazioni atomiche, riducendo il rischio di corruzione dei dati (a.e. in un sistema di registrazione dei voti, si vuole dare la lode solo se lo studente ha preso 30, cioè non deve essere possibile dare, a.e., 22 con lode).
- **(non) atomicità delle operazioni di accesso ai dati**: Un DBMS supporta transazioni atomiche che garantiscono che le operazioni di accesso ai dati vengano eseguite in modo coerente e affidabile, riducendo il rischio di errori e inconsistenti nei dati (a.e. un utente possiede due conti correnti e vuole muovere 5000 euro dal primo conto al secondo conto. Questa operazione deve essere fatta nella sua interezza, incrementando di 5000 euro il secondo conto corrente e successivamente decrementando di 5000 euro il primo. Non deve mai accadere che si verifichi solo una delle due operazioni);
- **anomalie causate da accessi concorrenti ai dati**: Un DBMS gestisce in modo efficace gli accessi concorrenti ai dati attraverso tecniche di controllo della concorrenza, riducendo il rischio di anomalie come la lettura sporca, l'interferenza e la perdita di aggiornamenti;
- **problemi di sicurezza/protezione dei dati**: Un DBMS offre funzionalità avanzate per garantire la sicurezza e la protezione dei dati, come il controllo degli accessi, l'autenticazione degli utenti e la crittografia dei dati, riducendo il rischio di accessi non autorizzati e violazioni della sicurezza.

Cos'è un database e cos'è un DBMS

Un database è una **collezione di dati** usata per rappresentare informazione di interesse per un dato sistema informativo, gestita da un **DataBase Management System** (DBMS). Un DBMS è lo strumento che consente di interagire con un insieme di file interconnessi, implementando tutti i punti visti sopra.

Componenti di un database

Il gestore

Il gestore di una base di dati (database manager) è il modulo software che realizza l'interfaccia con i dati fisicamente presenti nella base di dati (low-level data). Esso supporta le seguenti funzionalità:

- interazione con il file system;
- controllo e gestione dei vincoli di integrità (consistenza dei dati);
- gestione della sicurezza dei dati;
- backup e recovery in caso di malfunzionamenti;
- controllo della concorrenza.

L'amministratore

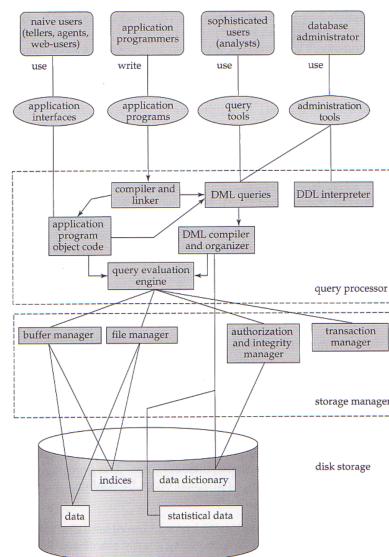
L'amministratore di una base di dati (database administrator) è una figura umana responsabile dei dati e delle relative procedure di accesso. Tale figura svolge le seguenti funzioni:

- definizione dello schema;
- definizione delle strutture di memorizzazione e dei relativi metodi di accesso;
- eventuale modifica dello schema logico e/o dell'organizzazione fisica dei dati;
- assegnazione delle autorizzazioni per l'accesso ai dati;
- specifica dei vincoli di integrità

Utenti

Possiamo distinguere più classi di utenti di una base di dati, sulla base delle diverse possibili modalità di interazione col sistema:

- utenti "naïve" (interazione trasparente con la base di dati mediante opportune interfacce applicative);
- utenti "sofisticati", che utilizzano direttamente il DML.
- utenti che sviluppano programmi applicativi scritti in un linguaggio di alto livello, tipo Cobol, Pascal, C, Java, che necessitano di accedere alla base di dati (uso del DML all'interno di linguaggi host);



Astrazioni e livelli

Le basi di dati supportano le **astrazioni sui dati**. I livelli di astrazione sono:

- il livello fisico (file, programmi. Qui lavora chi progetta database complessi);
- il livello logico/concettuale (modello relazionale, modello entità relazione);
- il [livello delle viste](#) (è un ritaglio del livello logico di interesse per una certa classe di utenti).

La differenza tra il livello logico e quello delle viste viene spiegato con un esempio: nel contesto universitario, il livello logico comprende l'insieme delle tabelle che descrivono il database dell'università ed è **unico**. Il livello delle viste è composto invece da **più possibili viste** come, per esempio:

- il livello attraverso il quale si relaziona alla base di dati il personale tecnico amministrativo (vista 1),
- il livello attraverso il quale si relazionano alla base di dati i docenti (vista 2),
- il livello attraverso il quale si relazionano alla base di dati gli studenti di un certo corso di studio (vista 3),
- ...

Quindi, ogni **classe di utenti** ha una **propria vista** con la quale interagisce con il database, **diversa** da quella di un'altra classe di utenti.

Modelli dei dati

Un modello dei dati è una **collezione di strumenti concettuali** (costrutti) per descrivere i dati, le loro relazioni e i vincoli di consistenza sui dati.

In letteratura sono stati proposti diversi modelli dei dati, in ordine cronologico:

- **modelli reticolare**: si usa un grafo, dove i nodi sono le singole entità e gli archi sono le relazioni binarie;
- **modelli gerarchico**: si usa un albero, dove in ogni livello si pone una determinata entità;
- **modelli relazionale ed entità/relazioni**: sono i modelli di riferimento;
- **modello orientato agli oggetti**;
- **modelli ibridi relazionali e orientati agli oggetti**;
- **modelli basati sulla (programmazione) logica**: utili per relazioni ricorsive (a.e. la reachability in un grafo);
- **modelli basati su XML**: utili per dati che hanno strutture non rigide (a.e. non c'è un numero fisso di attributi, ...);

Un'altra distinzione che si può fare è quella tra modelli basati su:

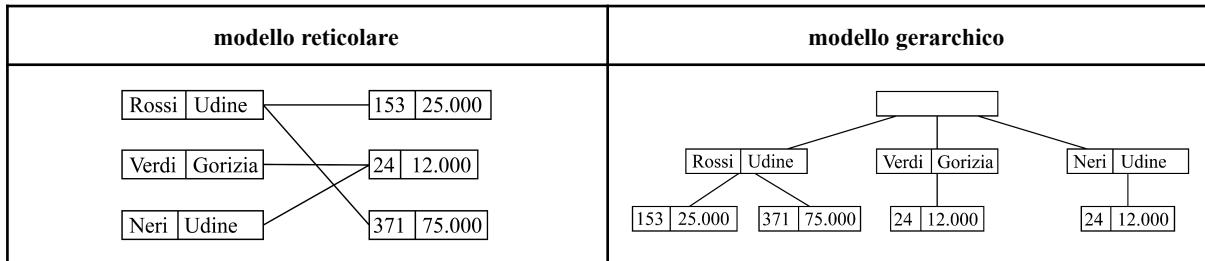
- **l'identità degli oggetti** (reticolare, gerarchico, object oriented): hanno un object identifier (o un indirizzo fisico);
- **sui valori** (entità/relazioni, relazionale, XML);

Un'ulteriore distinzione viene fatta tra modelli basati su:

- **gli oggetti** (entità/relazioni, orientato agli oggetti, XML): ammette dimensioni flessibili della rappresentazione;
- **i record** (reticolare, gerarchico, relazionale): c'è uno spazio predefinito riservato alla memorizzazione di ogni dato;

Esempio: filiale di una banca

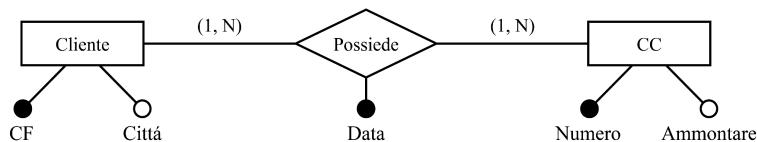
- Clienti: CF, Città, CC Posseduti,
- CC: Numero di conto, Ammontare, Possessori



Modello entità-relazioni ER

Il modello Entità-Relazione (ER, dall'inglese Entity-Relationship) è un modello concettuale che viene utilizzato nella progettazione e nell'architettura dei database per rappresentare le strutture dati in modo astratto. Sviluppato da Peter Chen nel 1976, il modello ER aiuta a definire i dati, le relazioni tra i vari dati e gli attributi in una maniera che sia sia intuitiva che sistematica, facilitando così la progettazione di database.

Dai modelli ER, è possibile derivare i **modelli fisici** dei database, che specificano come i dati sono effettivamente memorizzati in sistemi di database come MySQL, PostgreSQL, Oracle, e altri.



I costrutti fondamentali

Le entità

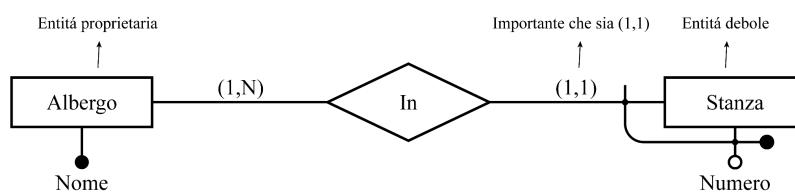
Sono le classi di oggetti che hanno un'**esistenza indipendente** e **autonoma** (un cliente e un conto corrente sono entità. L'ammontare di un conto corrente e l'età di un cliente non sono entità dato che non esistono in modo autonomo). Ogni entità ha un **nome** che la identifica in modo univoco e può assumere un valore diverso nel corso del tempo, detta **occorrenza di entità** (un'istanza). Ad esempio, l'entità "Cliente" ha come istanze, "Rossi", "Verdi" e "Neri" e tali istanze **non** vengono rappresentate nello schema entità-relazioni.

Entità deboli

Un'**entità debole** è un tipo di entità all'interno di un modello di dati che **dipende da un'altra entità**, chiamata **entità proprietaria**, per esistere. A differenza delle entità normali, un'entità debole non ha una chiave primaria che possa identificare univocamente da sola. La sua identità è strettamente legata all'entità proprietaria, e viene definita utilizzando un **identificatore parziale**, che è composto dalla chiave primaria dell'entità proprietaria insieme ad altri attributi.

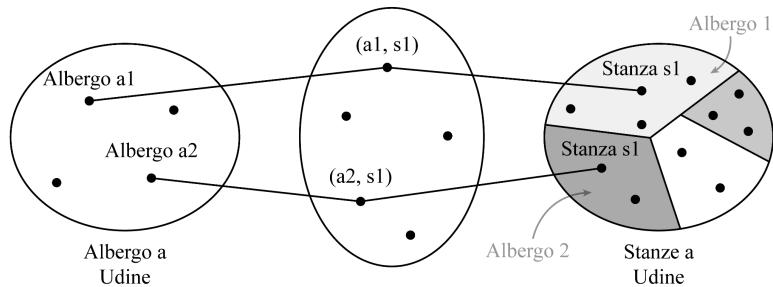
Inoltre, l'entità debole può partecipare a una relazione con l'entità proprietaria in modo **totale**, il che significa che ogni entità proprietaria deve essere associata a almeno un'entità debole, o in modo parziale, permettendo che alcune entità proprietarie non siano associate a un'entità debole.

Vediamo un esempio:



In questo caso, non ci sono attributi naturali per identificare una stanza di un albergo all'interno della città di Udine (numero da solo non significa nulla dato che non fa riferimento a nessun albergo specifico e potrebbe benissimo non essere un valore univoco dato che due alberghi potrebbero avere una stanza con lo stesso numero).

Serve dunque **partizionare** l'insieme delle istanze delle stanze in modo da dividerle a seconda di quale albergo appartengono.



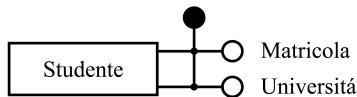
Gli attributi

Ogni entità ha degli **attributi**. Gli attributi descrivono le **proprietà elementari** di un'entità (o di una relazione). Un attributo associa a un'occorrenza, di entità o di relazione, un valore appartenente a un insieme dominio che contiene tutti i valori possibili del dominio. I domini non sono rappresentati nello schema E-R. In genere sono descritti nella documentazione del progetto.

Chiavi o identificatori interni

Ogni entità deve avere una **chiave** (detta anche identificatore interno), ossia un sottoinsieme *minimale* (più piccolo possibile) di attributi che sia univoco per ogni istanza dell'entità. Nel nostro esempio “CF” è perfetto per essere un attributo chiave, in quanto ogni cliente avrà sempre un CF univoco.

Nel caso in cui un attributo non sia in grado di garantire l'univocità da solo, si possono **unire** due o più attributi. Ad esempio, in un universo universitá, usare solo Matricola come chiave per identificare Studente sarebbe errato, in quanto, ad esempio Uniud e UniPd potrebbero avere la matricola XXXXXX assegnata a due studenti diversi. Per questo motivo si utilizza anche l'attributo Univeritá. Così facendo la chiave sarà unica anche tra universitá diverse.



Se vengono usati tutti gli attributi per formare una chiave (come nel caso qui sopra), allora essa è una **superchiave**. Se neanche tutti gli attributi formano una chiave, allora c'è un errore. In generale, se una chiave viene formata da tutti gli attributi e dunque è una **superchiave**, molto probabilmente **non è minimale** (nell'esempio sopra lo è), in quanto ci sarà qualche attributo che può essere **rimosso** dalla chiave mantenendo comunque la sua **univocità**.

Il dominio di un attributo e il valore NULL

Il dominio di un attributo **dom(A)** è l'insieme dei valori che l'attributo può assumere (a.e. parlando dell'attributo Stipendio, in un mondo non troppo cattivo il suo dominio sarà un valore maggiore di 0). Tuttavia vi possono essere istanze di entità che, rispetto ad un dato attributo, si trovano in una delle seguenti situazioni:

- possiedono un valore, ma tale valore non è noto;
- non possiedono alcun valore;
- non è noto se possiedono un valore o meno; in ogni caso, se lo possiedono, esso non è conosciuto.

Tali situazioni vengono gestite aggiungendo ad ogni dominio un valore sui *generis* detto valore NULL.

Attributi semplici e composti

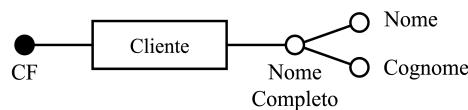
Gli **attributi semplici**, noti anche come **attributi atomici**, sono quelli che **non possono essere suddivisi ulteriormente** in parti più piccole che hanno significato all'interno del contesto del database. Esempi di attributi semplici possono includere:

- Numero di telefono
- Indirizzo email
- Data di nascita

Gli **attributi composti**, al contrario, sono quelli che **possono essere suddivisi in più attributi semplici**. Alcuni esempi di attributi composti includono:

- Nome completo (può essere suddiviso in "Nome", "Secondo Nome" e "Cognome")
- Indirizzo completo (può essere diviso in via, città, CAP e paese)
- Intervallo di date (può essere suddiviso in data di inizio e data di fine)

La suddivisione degli attributi composti in più attributi semplici può aiutare nella normalizzazione dei database, **riducendo la ridondanza** dei dati e migliorando l'efficienza dello storage.



Attributi a singolo valore o multivaleore

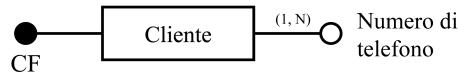
Gli attributi a **singolo valore** sono il tipo più comune di attributi nei database e rappresentano le **proprietà** che hanno un **unico valore per entità**. Ad esempio, un'entità "Persona" può avere attributi a singolo valore come:

- Numero di telefono (assumendo che si registri solo un numero principale)
- Indirizzo email
- Data di nascita

Gli attributi **multivaleore**, al contrario, possono contenere **più valori per ogni istanza** di un'entità. Un esempio classico è quando un'entità "Persona" ha un attributo "Numeri di telefono", e quella persona potrebbe avere più numeri di telefono (casa, lavoro, cellulare, ecc.). Altri esempi includono:

- Indirizzi email (una persona potrebbe avere più indirizzi email)
- Qualifiche o titoli di studio (una persona può avere più diplomi o certificati)

Per denotare la natura di attributi singoli o multivaleore si scrive usa la notazione:



Attributi obbligatori o opzionali

Gli **attributi obbligatori** sono quelli che **devono** essere forniti per ogni istanza di un'entità. Questi attributi sono essenziali e devono avere un valore valido per ogni riga nella tabella del database. Ad esempio, se stiamo progettando un database per un'applicazione di gestione degli impiegati, l'attributo "Nome" potrebbe essere obbligatorio, poiché ogni impiegato **deve** avere un nome.

Gli **attributi opzionali** non sono necessari per ogni istanza di un'entità e **possono essere lasciati vuoti** o nulli. Ad esempio, nell'applicazione di gestione degli impiegati, l'attributo "Numero di telefono cellulare" potrebbe essere opzionale, poiché alcuni impiegati potrebbero non fornire un numero di cellulare.

Questa caratteristica si ottiene impostando il numero minimo a:

- 0 per gli attributi opzionali
- 1 per gli attributi obbligatori, che devono aver almeno un'istanza



Attributi primitivi o derivati

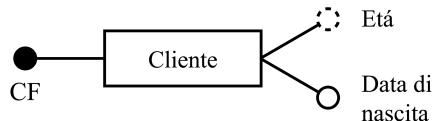
Gli **attributi primitivi** sono quegli attributi che contengono valori che **vengono inseriti nel database e non dipendono** da altri attributi per il loro valore. Esempi di attributi primitivi includono:

- Nome di una persona
- Data di nascita
- Numero di telefono
- Indirizzo email

Gli **attributi derivati** sono quelli il cui valore **viene derivato da altri attributi** o combinazioni di attributi presenti nel database. Questi attributi non sono memorizzati direttamente nel database, ma sono **calcolati quando necessario**. L'utilizzo degli attributi derivati può migliorare l'efficienza del database evitando la duplicazione dei dati e riducendo lo spazio di archiviazione necessario. Esempi di attributi derivati includono:

- Età (derivata dalla data di nascita rispetto alla data corrente)
- Anni di servizio di un dipendente (calcolati dalla data di assunzione rispetto alla data corrente)
- Totale spesa di un ordine (somma dei costi dei prodotti ordinati)

Non si deve **mai** poter inserire un valore per un attributo derivato. Piuttosto deve essere presente una **procedura** che calcola in modo automatico il suo valore. Il loro valore deve essere **aggiornato** ogniqualvolta l'informazione da cui vengono calcolati viene aggiornata.



Modelli basati sui valori

Modelli in cui gli esemplari concreti (istanze) vengono discriminati in base ai valori che essi assumono sugli attributi, si chiamano **modelli basati sui valori**. Il modello ER è un modello basato sui valori. I modelli reticolari e gerarchici non lo sono in quanto ogni nodo viene identificato attraverso l'indirizzo fisico in cui si trova.

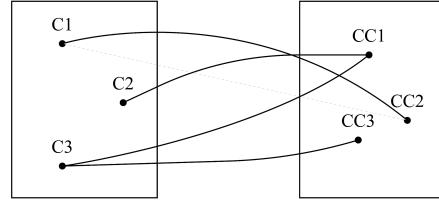
Le relazioni

Le relazioni (o associazioni) sono collegamenti logici tra una, due o più entità. Nel modello ER le relazioni hanno un **nome univoco**, in genere un sostantivo e non un verbo, che le identifica.

Le molteplicità

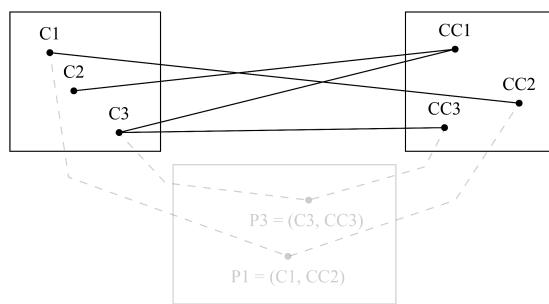
Potrebbe accadere che:

- il cliente C1 possiede il conto CC1,
- il cliente C2 possiede il conto CC1,
- il cliente C3 posseggi i conti CC1 e CC3.



Dunque, guardando questo schema, si capisce che ogni cliente deve avere almeno 1 conto e al massimo N conti. Questa informazione viene codificata inserendo la coppia (1, N) sopra l'arco tra il cliente e la relazione.

Può essere utile rappresentare graficamente anche le **istanze della relazione**, vedendole come le coppie entità-entità:



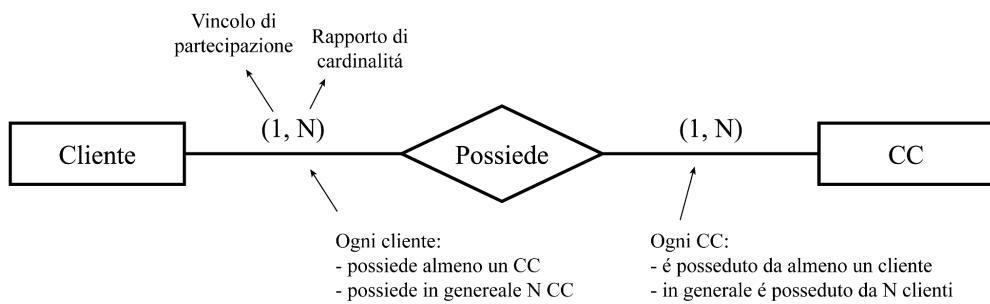
In questo modo il minimo e il massimo che vengono scritti sopra l'arco della relazione sono:

- il numero minimo di coppie in cui trovo un certo cliente,
- il numero massimo di coppie in cui trovo un certo cliente.

Duale è il ragionamento per il collegamento a destra:

- il numero minimo di coppie in cui trovo un certo conto,
- il numero massimo di coppie in cui trovo un certo conto.

Dunque:



Vincoli di partecipazione e rapporto di cardinalità

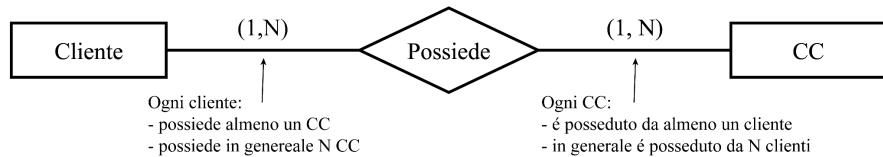
Il **vincolo di partecipazione** può assumere valori:

- 0: partecipazione **parziale** (ci sono clienti che sono clienti anche senza possedere un conto)
- 1: partecipazione **totale** (ogni cliente deve avere almeno un conto per essere cliente)

Il vincolo di partecipazione e rapporto di cardinalità assumono diversi valori:

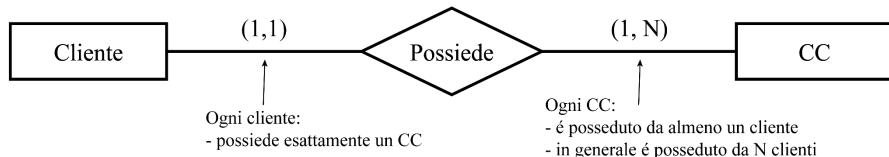
- $(_, N)-----(_, N)$: relazione molti a molti:

Questo significa che ogni cliente ha almeno 1 conto e al massimo N conti. Ogni conto è posseduto da almeno 1 cliente e al massimo da N clienti diversi.



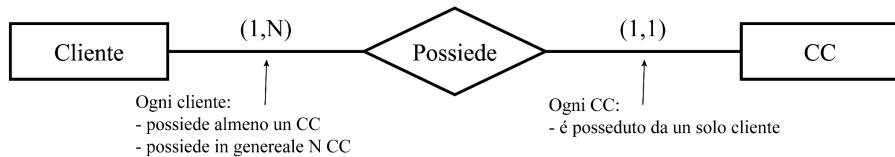
- $(_, 1)-----(_, N)$: relazione molti a uno

Questo significa che ogni cliente ha almeno 1 conto e al massimo 1 conto, dunque ha esattamente un conto. Ogni conto è posseduto da almeno un cliente e al massimo da N clienti diversi. Ossia ogni cliente ha un solo conto ma i conti possono essere condivisi tra i clienti.



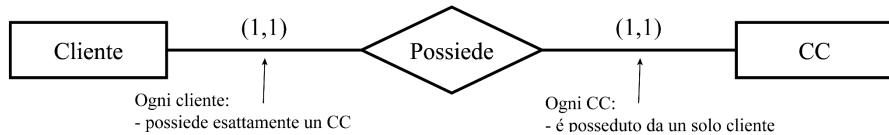
- $(_, N)-----(_, 1)$: relazione uno a molti

Questo significa che un conto corrente ha almeno un possessore e al massimo un possessore (ogni conto è riservato ad un solo cliente). Inoltre ogni cliente deve avere almeno un conto e al massimo N.



- $(_, 1)-----(_, 1)$: relazione uno a uno

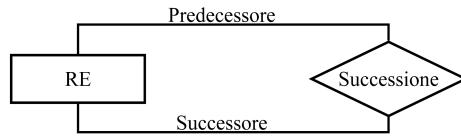
Questo significa che un cliente possiede esattamente un conto corrente e un conto corrente è posseduto solo e solamente da un cliente.



Relazioni ricorsive

Una relazione è detta **ricorsiva** se coinvolge due volte la stessa entità. È necessario e obbligatorio specificare tramite un'etichetta il **ruolo** con il quale l'entità partecipa alla relazione nei due (o più) casi.

Immaginiamo di voler modellare la relazione di successione tra regnanti:



Per stabilire i minimi e i massimi bisogna chiedersi:

- sul lato del predecessore:
 - qual è il numero minimo in cui un Re può comparire nel ruolo del Predecessore? 0
 - qual è il numero massimo in cui un Re può comparire nel ruolo del Predecessore? 1
- sul lato del successore:
 - qual è il numero minimo in cui un Re può comparire nel ruolo del Successore? 0
 - qual è il numero massimo in cui un Re può comparire nel ruolo del Predecessore? 1

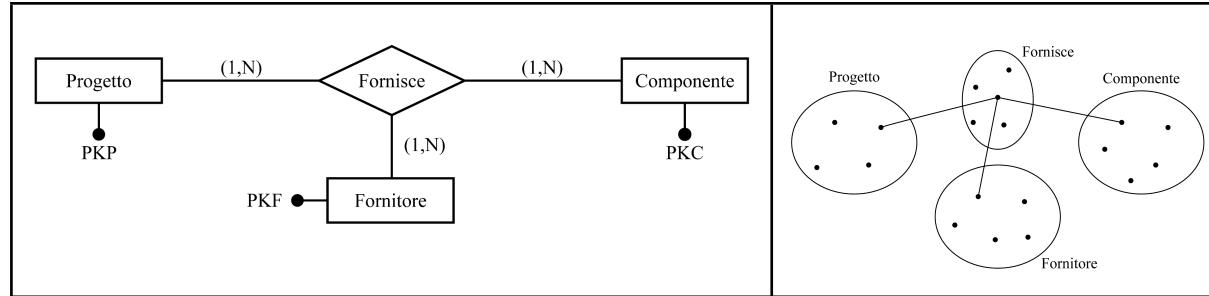
Attributi di relazione

Nel modello Entità-Relazione (ER), un attributo di relazione è una proprietà o una caratteristica che descrive un aspetto della relazione tra due o più entità. Gli attributi di relazione vengono utilizzati per aggiungere dettagli o informazioni specifiche riguardo alla natura della relazione stessa.

Ad esempio, consideriamo una relazione "Lavora" tra le entità "Impiegato" e "Progetto". Un attributo di questa relazione potrebbe essere "DataInizio" che specifica la data in cui l'impiegato ha iniziato a lavorare su un determinato progetto. Un altro esempio potrebbe essere "OreLavorate" che indica il numero di ore che l'impiegato ha dedicato al progetto.

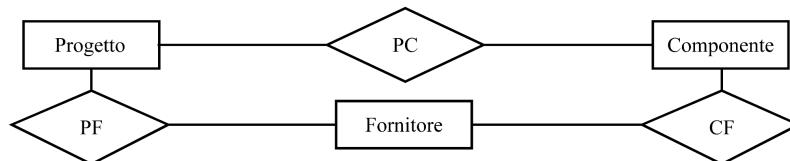
Nota: le relazioni non hanno le chiavi.

Relazioni ternarie o di grado superiore



Tuttavia non sono necessarie e possono (devono) essere tradotte in relazioni binarie.

Soluzione 1



Immaginiamo di avere:

- due istanze di Progetto {p, p'},
- due istanze di Fornitore {f, f'},

- due istanze di Componente $\{c, c'\}$,

e supponiamo che la Relazione fornisce sia:

$$\text{istanza(fornisce)} = \{(f, c, p'), (f, c', p), (f', c, p)\}$$

Ora immaginiamo di codificare questa informazione associandola alle relazioni binarie, mediante un procedimento detto **proiezione** (di relazioni su un sottoinsieme di argomenti):

$$\begin{aligned}\text{istanza}(CF) &= \{(f, c), (f, c'), (f', c)\} \\ \text{istanza}(PF) &= \{(f, p'), (f, p), (f', p)\} \\ \text{istanza}(PC) &= \{(c, p'), (c', p), (c, p)\}\end{aligned}$$

Per capire se si sono perse delle informazioni, tento di ricostruire la relazione ternaria iniziale sulla base di quelle binarie: inserirò una certa terna se, a livello della relazione binaria, ho evidenza che

- quel fornitore fornisce quella componente,
- quel fornitore fornisce quel progetto,
- quella componente è una componente di quel progetto.

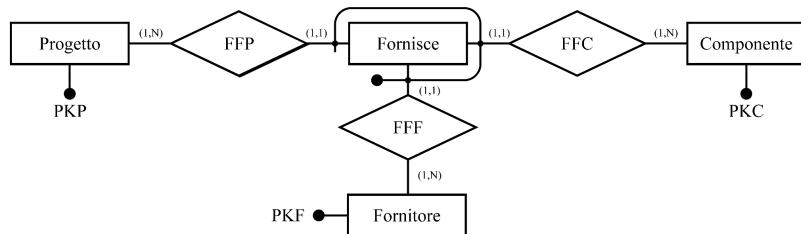
Se c'è un legame tra le coppie costruibili su una terna, allora la terna viene inserita:

$$\begin{array}{l} (f, c), (f, c'), (f', c) \\ (f, p'), (f, p), (f', p) \\ (c, p'), (c', p), (c, p) \\ \hline (f, c, p'), (f, c', p), (f', c, p) \end{array}$$

Questa operazione si chiama **natural join**. Tuttavia esce anche una quarta tripla possibile, data dalla diagonale (f, c) , (f, p) , (c, p) , quindi c'è stata una perdita di informazione. Dunque non è possibile passare dalla relazione ternaria a relazioni binarie qualora si faccia riferimento all'insieme iniziale di entità.

Soluzione 2: la reifica

La soluzione a questo problema è la **reificazione della relazione**, la quale consiste nel rendere una relazione un'entità debole, con chiave parziale vuota, che per essere identificata deve fare riferimento alle tre entità.



Soluzione 3: evitarle

Non sono così frequenti ed è molto più semplice evitarle al principio piuttosto che inserirle e reificare dopo.

Costrutti aggiuntivi

La generalizzazione

La generalizzazione in un modello ER si riferisce alla capacità di astrazione di concetti comuni tra diverse entità. Questa tecnica consente di ridurre la complessità del modello dati, identificando estraendo le caratteristiche comuni tra le entità e creando una **struttura più gerarchica** dove:

- ogni proprietà del genitore è anche una proprietà dei figli;
- ogni istanza dei figli è anche un'istanza del padre;
- ogni proprietà (attributi, relazioni, altre generalizzazioni) del genitore vengono ereditate dai figli;

Tipi di generalizzazione

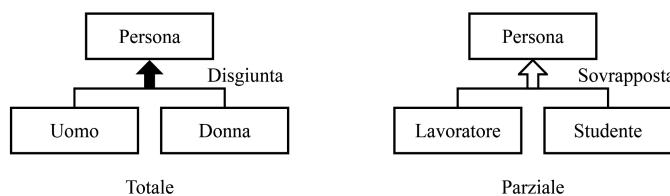
Esistono diversi tipi di generalizzazione:

- **totale o parziale**;
- **disgiunta o sovrapposta**.

Esempi: la specializzazione di persona in:

- uomo e donna è **totale e disgiunta** (tutte le persone sono o uomo o donna);
- disoccupato e lavoratore è **parziale e disgiunta** (esistono persone che non sono né lavoratori né disoccupati);
- studente e lavoratore è **parziale e con sovrapposizioni** (una persona può essere studente lavoratore);

Si rappresentano in questo modo:



Le generalizzazioni attribute-defined

L'appartenenza di un'istanza dell'entità genitore ad una o più entità figlio può essere

- **user-defined:** definita dall'utente;
- **attribute-defined:** l'appartenenza di un'istanza del genitore ad uno o più figli è stabilita in base al valore che tale istanza assume su un determinato attributo (a.e.l'appartenza di un'istanza dell'entità persona all'entità uomo o all'entità donna viene determinata sulla base del valore da essa assunto sull'attributo sesso);

Il problema dell'ereditarietà multipla

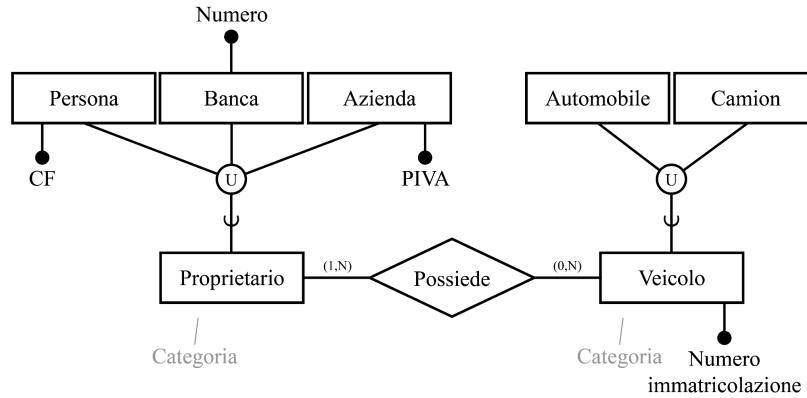
Abusando di generalizzazioni, può accadere che alcune entità appartengano a più gerarchie di specializzazione. Si passa da alberi a reticolati.

Tipi unione (o categorie)

Si consideri una base di dati per l'immatricolazione di veicoli. Si assuma che tale base di dati contenga una relazione possiede che lega i veicoli immatricolati ai proprietari. Per quanto riguarda i veicoli immatricolati, assumiamo di dover

gestire le immatricolazioni di automobili e camion. Per quanto riguarda i proprietari, assumiamo che possano essere persone, banche (ipoteca sul veicolo) e aziende.

Occorre creare un tipo di **entità** che svolga il ruolo di proprietario del veicolo e contenga istanze dei 3 tipi di entità coinvolti. A tal fine, si crea una categoria Proprietario, le cui istanze costituiscono un **sottoinsieme dell'unione delle istanze** di Persona, Banca e Azienda. Analogamente, è possibile introdurre una categoria Veicolo le cui istanze costituiscono un sottoinsieme dell'unione delle istanze di Automobile e Camion.



Automobile e Camion ereditano la chiave da Veicolo, mentre Proprietario ottiene la chiave nel momento in cui viene istanziata e viene ottenuta sulla base del tipo di proprietario.

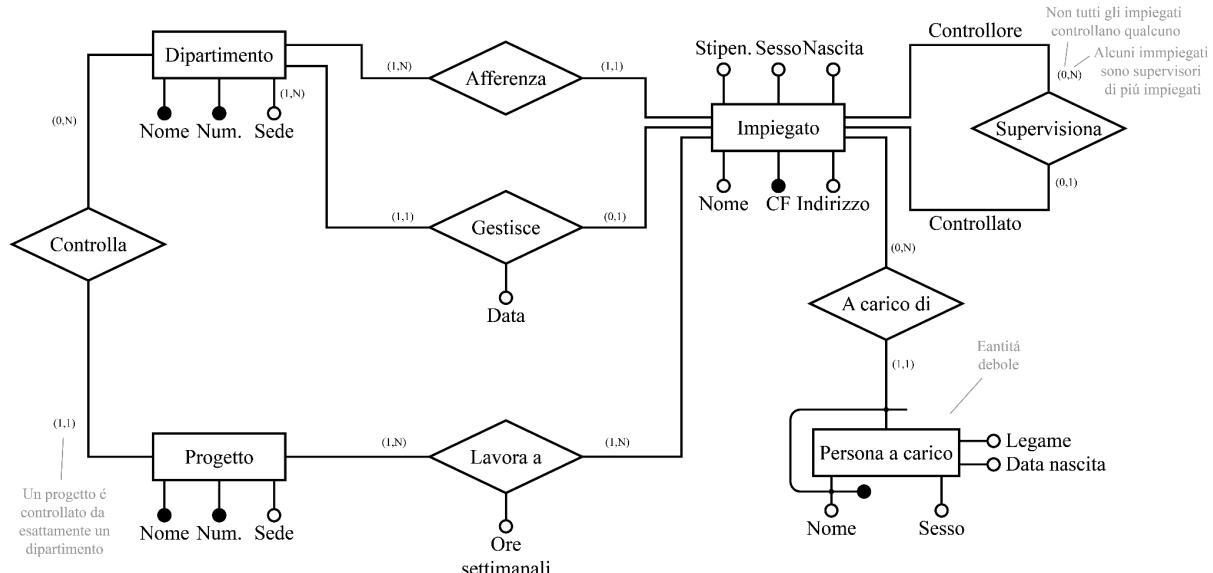
Esercizi sul ER

Si voglia sintetizzare uno schema ER che registri informazioni sugli **impiegati**, i **dipartimenti** e i **progetti** di un'azienda. Si supponga di aver raccolto i seguenti requisiti:

1. L'azienda sia organizzata in **dipartimenti**. Ogni dipartimento abbia un unico **nome**, un unico **numero** ed uno specifico impiegato che lo gestisce (**manager**). Si voglia tener traccia della **data** in cui tale impiegato ha assunto tale ruolo. Ogni dipartimento possa avere più **sedi**.
2. Un dipartimento **controlla** un certo insieme di **progetti**, ognuno dei quali sia contraddistinto da un unico **nome**, un unico **numero** e una singola **sede**.
3. Ogni **impiegato** sia contraddistinto da un **nome**, un **codice fiscale**, che lo identifica univocamente, un **indirizzo**, uno **stipendio**, un **sesso** e una **data di nascita**. Ogni impiegato riferisca ad un unico **dipartimento**, ma **possa lavorare a più progetti**, non necessariamente controllati dallo stesso dipartimento. Si voglia tener traccia del **numero di ore** per settimana che un impiegato dedica ad ogni progetto. Infine, si voglia tener traccia del **supervisore** diretto di ogni impiegato.
4. Si voglia tener traccia per motivi assicurativi delle **persone a carico** di ogni impiegato. Si voglia tener traccia del **nome** della persona a carico, del **sesso**, della **data di nascita** e del **legame con l'impiegato** (figlio, coniuge, genitore, etc.).

Osservazioni:

- l'azienda è il contesto e **non** va inserita nello schema ER (se venisse inserita, avrebbe un'unica istanza).
- "sedi" lo inserisco come attributo o come entità? non c'è alcuna informazione (attributi) per le sedi, quindi la inserisco come attributo del dipartimento. Per codificare "ogni dipartimento possa avere più sedi", sede dev'essere **multivaleore** (1,N).
- "sedi" dei progetti non sono legati a sedi del dipartimento, quindi anche qui, viene inserita come attributo.
- "Un dipartimento controlla un certo insieme di progetti", dunque il massimo è N. Gli impiegati non necessariamente lavorano ad un progetto a cui afferiscono, dunque il minimo è 0.
- "Si voglia tener traccia del **numero di ore** per settimana che un impiegato dedica ad ogni progetto" lo modello usando un attributo nella relazione Lavora a
- "si voglia tener traccia del **supervisore** diretto di ogni impiegato" è un legame tra impiegati e impiegati, quindi è una relazione ricorsiva



I cicli e i vincoli d'integrità

Bisogna fare attenzione ai **cicli**, ad esempio Dipartimento e Impiegato possono essere raggiunti seguendo due strade diverse, dunque c'è un ciclo. Sono percorsi totalmente liberi o portano ad inconsistenze?

Può accadere il seguente caso: l'impiegato Rossi afferisce al dipartimento X, Rossi lavora al progetto P, ma P non è controllato dal dipartimento X (a cui afferisce Rossi). Sarebbe però un problema solo nel caso in cui un impiegato potesse lavorare solo a progetti del proprio dipartimento, ma dato che il testo non pone questo vincolo, non c'è alcun problema in questo ciclo.

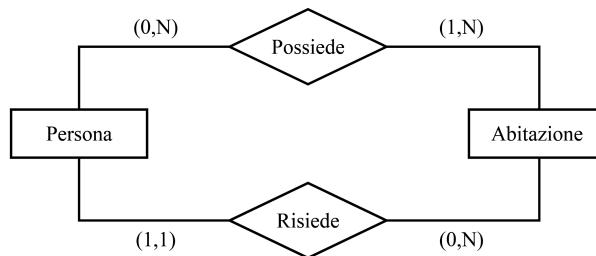
Vediamo ora un altro ciclo, il quale potrebbe dare vita a questo caso: l'impiegato Rossi afferisce al dipartimento X ma gestisce il dipartimento Y (a cui non afferisce). Questo rappresenta un'inconsistenza e quindi necessita di un **vincolo di integrità**: "Un impiegato può essere manager solo di dipartimenti a cui afferisce".

Nota: i vincoli di integrità vanno inseriti solo nel caso in cui non sia possibile codificare il vincolo nello schema

Altro esempio di ciclo non problematico

Immaginiamo la seguente situazione:

- (0, N): Una persona potrebbe non possedere alcuna abitazione oppure possederne più di una;
- (1, N): Un'abitazione ha almeno un possessore e potrebbe averne diversi;
- (1, 1): Ogni persona risiede in una e solo una abitazione;
- (0, N): Un'abitazione può non avere residenti o averne diverse.

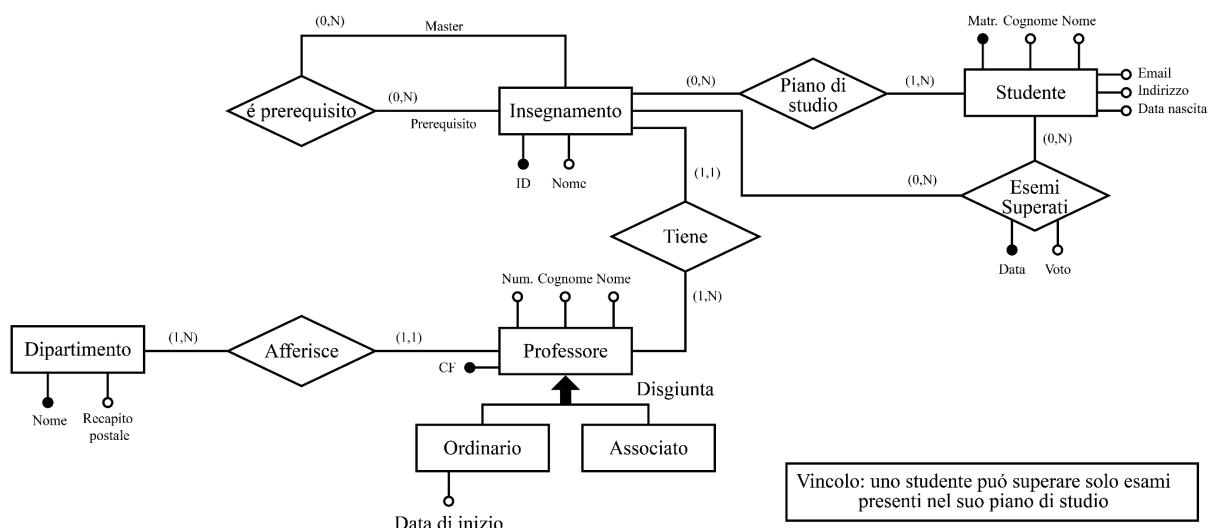


Questo è un ciclo perfettamente legittimo in quanto non ci sono inconsistenze (potrebbe essere che alcune persone risiedono in un'abitazione che non possiedono, potrebbero esserci delle persone che possiedono delle abitazioni in cui non risiedono, ...)

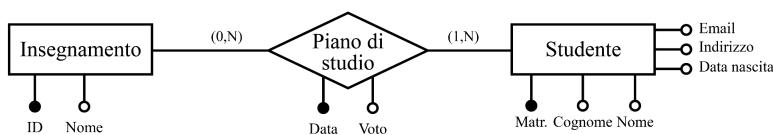
Esercizio 22 luglio 2020

Si vuole realizzare una base di dati per la gestione di informazioni relative a un **corso di studio di un'università** sulla base del seguente insieme di requisiti.

- Ogni **insegnamento** sia identificato univocamente da un **codice numerico ID** e sia caratterizzato da un **nome**, un **insieme di prerequisiti** (insieme di altri insegnamenti che devono essere già stati superati per poter sostenere l'esame dell'insegnamento) e il **professore** che tiene le lezioni (ogni insegnamento sia tenuto da un solo professore).
- Ogni **studente** sia identificato univocamente da un **numero di matricola** e sia caratterizzato da un **nome**, un **cognome**, un **indirizzo di posta elettronica**, un **recapito postale** e una **data di nascita**. Ogni studente abbia uno o più **esami** inseriti nel suo **piano degli studi** e abbia superato un certo numero di essi (non si escluda l'eventualità che non ne abbia ancora superato alcuno). Per ogni insegnamento del quale lo studente ha già superato l'esame, si registrino la **data** in cui ha sostenuto con successo l'esame e il **voto** ottenuto.
- I **professori** sono suddivisi in professori **ordinari** e professori **associati**. Ogni professore sia identificato univocamente dal suo **codice fiscale** e sia caratterizzato da un **nome**, un **cognome**, un **numero di cellulare**, il **dipartimento cui afferisce** e gli **insegnamenti** che tiene (uno o più). Ogni **dipartimento** sia identificato univocamente da un **nome** e sia caratterizzato da un **recapito postale**. Di ogni professore ordinario vogliamo registrare la **data in cui ha preso servizio** in tale veste (il primo giorno nel ruolo di professore ordinario).



Il ciclo che si è formato è pericoloso in quanto potrebbe accadere che un insegnamento non sia nel piano di studi di uno studente, ma potrebbe accadere che lo studente ha superato l'esame. Soluzione alternativa per non utilizzare il vincolo:



Esercizio 4 settembre 2019 (lez. 16)

Si vuole realizzare una base di dati per la gestione di informazioni circa un insieme di automobili caratterizzato dal seguente insieme di requisiti.

- Ogni **automobile** sia identificata univocamente dalla sua **targa** e sia caratterizzata da un **modello**, un **anno di fabbricazione**, un **colore**, un **valore di mercato** e uno o più **proprietari**. Fra le automobili, vogliamo tener traccia del sottoinsieme delle **automobili storiche** (un'auto si dice storica se sono trascorsi 25 o più anni dall'anno di fabbricazione), del sottoinsieme delle automobili **sportive**, caratterizzate dalla **velocità massima**, e del sottoinsieme delle auto **storiche sportive**.
- Ogni **modello** sia caratterizzato da un **nome**, una **casa costruttrice** e una **cilindrata**. Il nome identifichi univocamente il modello all'interno dei modelli proposti dalla casa costruttrice (non si esclude la possibilità che case costruttrici diverse propongono modelli, ovviamente diversi, con lo stesso nome).
- Ogni **casa costruttrice** sia identificata univocamente dal proprio **nome** e sia caratterizzata dall'**anno di fondazione** e da un **insieme di stabilimenti**. Una stessa **persona** può essere **presidente** di più case costruttrici. Ogni **stabilimento** è caratterizzato da un **nome**, che lo identifica univocamente nell'ambito della casa costruttrice, una **città** ove ha sede e un **numero di addetti**.

Considerazioni

- Per modellare il requisito:

“Fra le automobili, vogliamo tener traccia del sottoinsieme delle automobili storiche, del sottoinsieme delle automobili sportive, caratterizzate dalla velocità massima, e del sottoinsieme delle auto storiche sportive.”

si può avere un'entità Automobile che è la generalizzazione (parziale e con sovrapposizione) di Automobile Sportiva e Automobile Storica. A sua volta inseriamo un'entità Automobile sportiva e storica come raffinamento di Automobile Sportiva e Automobile Storica, la quale rappresenta l'intersezione delle loro istanze.

- Il presidente dev'essere per forza un'entità in quanto può essere presidente di più case costruttrici

Modello relazionale

Il modello relazionale è un modello dei dati **basato sui record** (dunque in Prima Forma Normale - 1NF) e sui **valori** (ogni esemplare concreto è identificato univocamente dai valori che assume su diversi attributi), più astratto dei modelli reticolare e gerarchico, meno astratto del modello Entità/Relazioni.

È composto da **tabelle** (dette anche relazioni). Ogni tabella rappresenta un'entità o una relazione e le colonne della tabella (attributi della relazione) sono gli **attributi** dell'entità.

Cliente			CC		
CF	Città	CC	Numero	Ammon.	Cliente

Per ogni colonna/attributo bisogna specificare l'insieme dei possibili valori che possono essere inseriti. Questo insieme è detto **dominio**.

Le tabelle (relazioni) vengono **popolate** inserendo delle righe (tuple), corrispondenti alle istanze delle entità:

Cliente			CC		
CF	Città	CC	Numero	Ammon.	Cliente
Rossi	Udine	153	153	25000	Rossi
Rossi	Udine	371	24	12000	Verdi
Verdi	Gorizia	24	24	12000	Neri
Neri	Udine	24	371	75000	Rossi

Questa forma va sotto il nome di **prima forma normale**, in cui ogni cella deve avere **una e una sola informazione** (infatti per mostrare che Rossi ha due conti sono state usate due righe nella tabella). Modelli che soddisfano questa forma sono detti **modelli basati sui record** e sono contrapposti ai modelli basati sugli oggetti.

Anche in questo modello, si deve scegliere una **chiave** (viene sottolineata) in modo da identificare **univocamente** le istanze della tabella (le singole righe). In questo caso, la chiave potrebbe essere la coppia CF+CC per la tabella Cliente e la coppia Numero+Cliente per la tabella CC.

Problematiche

Questa soluzione **non va bene** perché contiene delle **ridondanze**, ossia alcune informazioni sono ripetute in modo identico in più posti diversi e questo può dare luogo a delle **inconsistenze**.

Discrimine tra presenza o assenza di una ridondanza

È presente una ridondanza quando il valore di un certo attributo può essere determinato a partire da **altri valori** presenti nella base di dati (Nota bene: le due occorrenze di Rossi nella colonna CF non sono una ridondanza, dato che indicano due

informazioni diverse. Sono invece una ridondanza le due occorrenze di Udine, nella colonna Cittá, dato che, a fronte di modifiche nella città di residenza di Rossi, potrebbe accadere che Rossi abiti a Udine secondo la prima riga e che magari Rossi abiti a Gorizia secondo la seconda riga).

Il **sistema** garantisce l'**unicità delle chiavi**: immaginiamo che sia presente solo la prima riga Rossi, Udine, 153. Se un utente tenta di inserire Rossi, Pordenone, 371. Il sistema controlla il valore della chiave e cerca una riga che contenga Rossi, ..., 371 nella tabella. Nel momento in cui non trova nessuna riga con questi valori accetta l'inserimento.

La soluzione

La soluzione è dividere le tabelle, usando una tabella per la relazione (in questo caso viene anche aggiunto l'attributo della relazione).

Cliente		Possiede			CC	
CF	Cittá	CF	Numero	Data	Numero	Ammon.
Rossi	Udine	Rossi	153	...	153	25000
Verdi	Gorizia	Rossi	371	...	24	12000
Neri	Udine	Verdi	24	...	371	75000
		Neri	24	...		

La nuova tabella "Possiede", avrà come colonne le due chiavi delle entità che la compongono e tali chiavi vengono dette **chiavi esterne (FK)** e fanno riferimento alle chiavi primarie delle entità.

Costrutti fondamentali

Dominio, attributo, relazione, tupla

Il **dominio** è un'insieme di valori **atomici** (non ci sono attributi multivalue e/o composti). È possibile definire un dominio **globalmente**, dunque a se stante rispetto alle tabelle.

Attributi

Gli **attributi** possono avere o non avere un nome: se privo di nome, l'attributo è identificato dalla sua posizione (a.e. seconda colonna rappresenta il CF). È obbligatorio avere la possibilità di **rinominare** gli attributi, specialmente per avere dei vantaggi durante le operazioni di join tra le tabelle che usano i nomi.

Relazioni

Una relazione (tabella) ha un **nome**. Lo **schema di una relazione** $R(A_1, \dots, A_n)$ definisce la sua struttura, dove il grado di una relazione corrisponde al numero dei suoi attributi (n). Un'istanza di relazione è indicata con $r(R)$ o $r \in R$ dove r rappresenta un insieme di righe $t = \langle v_1, \dots, v_n \rangle$, con $v_i \in \text{dom}(A_i)$ per $1 < i < n$. Il numero di tuple di una relazione, o meglio di una istanza di una relazione viene chiamato **cardinalità**.

Nel contesto del modello relazionale, le relazioni sono caratterizzate da diverse proprietà fondamentali. Tra queste, vi è l'organizzazione delle tuple all'interno di una relazione, che può essere considerata come un insieme ordinato di righe. Inoltre, all'interno di ciascuna tupla, i valori possono essere disposti in due modalità:

- come un insieme di coppie attributo-valore, quando gli attributi sono distinti e identificati
- come una sequenza ordinata di valori quando gli attributi sono anonimi.

Questa organizzazione riflette la struttura intrinseca dei dati all'interno del modello relazionale. Oltre a ciò, la gestione dei valori delle tuple, garantendo la conformità alla Prima Forma Normale (1NF), richiede la gestione degli attributi composti e multivalore, come definito nel modello di Entità-Relazione (ER). Infine, vanno considerati anche i vincoli di integrità referenziale, nonché il concetto di chiave primaria e chiave esterna, che contribuiscono alla coerenza e all'integrità dei dati all'interno del database.

Schema e istanza di una base di dati

Uno **schema di una base di dati relazionale** è un insieme di schemi di relazione $\{R_1, \dots, R_n\}$ + insieme di vincoli di integrità. Un'**istanza di una base di dati relazionale** è l'insieme di istanze di relazione $\{r_1, \dots, r_n\}$ che soddisfano i vincoli di integrità.

I vincoli di integrità possono essere:

Intra-relazionali

Sono vincoli che coinvolgono:

- un singolo valore di una singola tupla:
 - vincolo di **dominio**: gli attributi delle tuple devono esistere nel dominio;
 - vincolo **NOT NULL**: utile per certi attributi che non possono mai assumere il valore NULL;
- più valori di una singola tupla: a.e. nella tabella:

ESAMI(STUDENTE, MATERIA, DATA, VOTO, LODE)

la LODE può essere assegnata solo se il VOTO è 30.
- più tuple di una singola tabella:
 - vincolo di **chiave**: specifica delle chiavi candidate (le nozioni di superchiave, di chiave candidata e di chiave primaria).

Al vincolo di chiave viene affiancato il vincolo di integrità dell'entità: il valore della chiave primaria non può essere NULL

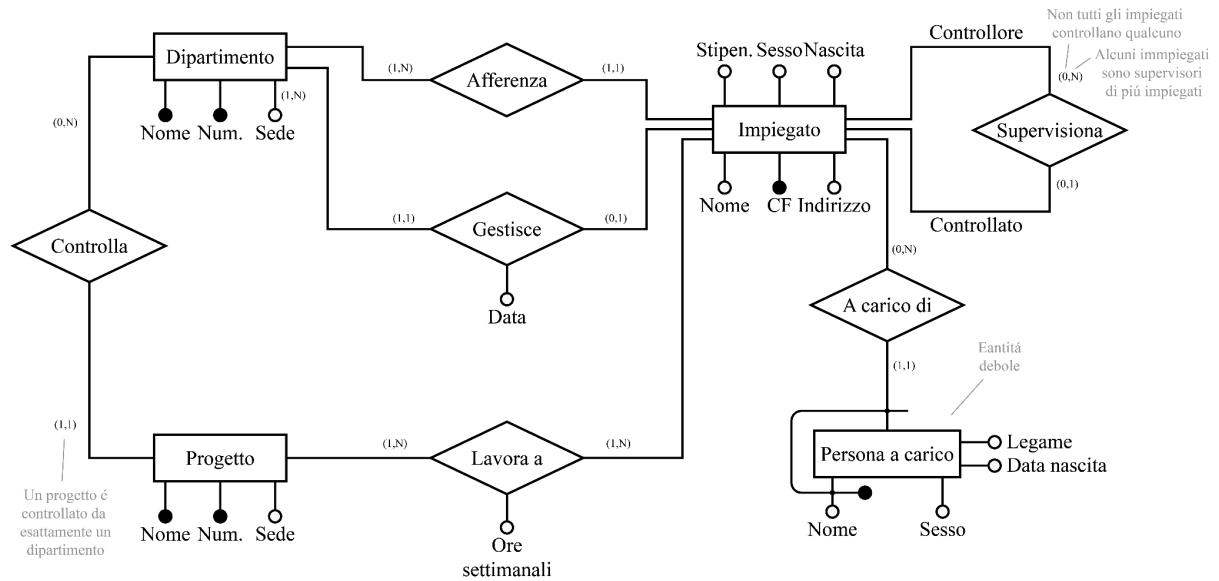
Inter-relazionali

Sono vincoli che coinvolgono tuple appartenenti a relazioni diverse. Un esempio è il vincolo di **integrità referenziale** (o di **chiave esterna**). un insieme di attributi FK di uno schema di relazione R_1 è una **chiave esterna** rispetto ad uno schema di relazione R_2 se

- gli attributi in FK hanno lo stesso dominio degli attributi che costituiscono la chiave primaria PK di R_2 (in verità, è sufficiente che soddisfano il vincolo di unicità);
- date $r_1 \in R_1$ e $r_2 \in R_2$, per ogni $t_1 \in r_1$ se $t_1[FK] \neq NULL$ (su nessun attributo di FK t_1 assume il valore NULL), allora esiste $t_2 \in r_2$ tale che $t_1[FK] = t_2[FK]$.

Il vincolo di chiavi esterne permette di garantire il significato della relazione di afferenza dello schema ER.

Un esempio di schema relazionale



```

IMPIEGATO(NOME_INIZIALE, COGNOME, CF, DATA_NASCITA, INDIRIZZO, SESSO, STIPENDIO, SUPERVISORE, DIP)
DIPARTIMENTO(DNUMERO, DNOME, MANAGER, DATA_INIZIO)
SEDI_DIPARTIMENTO(DNUMERO, DSEDE)
PROGETTO(PNUMERO, PNOME, DNUM, PSEDE)
LAVORA_A(IMP, PROGETTO, ORE_SETTIMANA)           (relazioni molti a molti vanno esplicitate a parte)
PERSONA_A_CARICO(IMP, NOME, SESSO, DATA_NASCITA, LEGAME)
  
```

Legenda: le CHIAVI PRIMARIE sono sottolineate; le CHIAVI ESTERNE sono in corsivo

Chiavi esterne

- SUPERVISORE è chiave esterna di IMPIEGATO rispetto a IMPIEGATO
- DIP è chiave esterna di IMPIEGATO rispetto a DIPARTIMENTO
- MANAGER è chiave esterna di DIPARTIMENTO rispetto a IMPIEGATO
- DNUMERO è chiave esterna di SEDI_DIPARTIMENTO rispetto a DIPARTIMENTO
- DNUM è chiave esterna di PROGETTO rispetto a DIPARTIMENTO
- IMP è chiave esterna di LAVORA_A rispetto a IMPIEGATO
- PROGETTO è chiave esterna di LAVORA_A rispetto a PROGETTO
- IMP è chiave esterna di PERSONA_A_CARICO rispetto a IMPIEGATO

Logical design

L'obiettivo della progettazione logica è quello di **costruire uno schema relazionale** che rappresenti correttamente ed efficientemente tutte le informazioni descritte da uno schema Entità-Relazione prodotto durante la fase di progettazione concettuale.

Questo non è solo una semplice traduzione da un modello all'altro per due ragioni principali:

- Non tutti i costrutti del modello Entità-Relazione possono essere tradotti naturalmente nel modello relazionale;
- Lo schema deve essere ristrutturato in modo da rendere l'esecuzione delle operazioni il più efficiente possibile.

Le fasi

È generalmente utile dividere la progettazione logica in due fasi:

1. **Ristrutturazione dello schema ER**, basata su criteri per l'**ottimizzazione** dello schema e **semplificazione**. Uno schema ER può essere ristrutturato per ottimizzare due parametri:

- **costo di un'operazione** (valutato in termini del numero di occorrenze di entità e relazioni che vengono visitate per eseguire un'operazione sul database);
- **requisiti di archiviazione** (valutati in termini del numero di byte necessari per memorizzare i dati descritti dallo schema).

Per studiare questi parametri, è necessario conoscere:

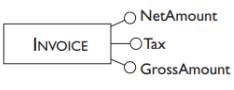
- il volume dei dati (numero di righe nelle tabelle);
- le caratteristiche dell'operazione (interrogazione? aggiornamento? con che frequenza?).

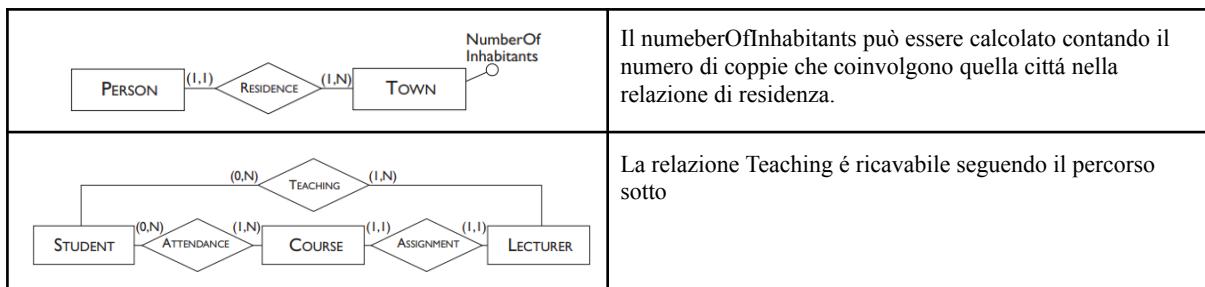
Concretamente, le fasi da seguire sono le seguenti:

1. Analisi delle ridondanze;
2. Rimozione generalizzazioni;
3. Partizionamento/unione di entità e relazioni;
4. Selezione chiavi primarie;
2. **Traduzione nel modello logico**, basata sulle caratteristiche del modello logico (nel nostro caso, il modello relazionale).

1. Analisi delle ridondanze

Una **ridondanza** in uno schema concettuale corrisponde a una informazione che può essere derivata (cioè ottenuta da una serie di operazioni di recupero) da altri dati. Uno schema Entità-Relazione può contenere varie forme di ridondanza.

	2 attributi mi permettono di conoscere il terzo.
	L'attributo totalAmount può anche essere calcolato sommando tutti i prezzi dei prodotti.



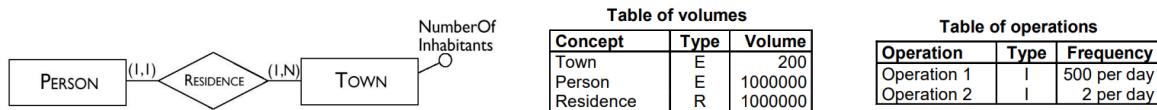
La presenza di una informazione derivata in un database presenta:

- **un vantaggio:**
 - una riduzione del numero di accessi necessari per ottenere l'informazione derivata;
- **alcuni svantaggi:**
 - un maggiore requisito di archiviazione (che spesso è un costo trascurabile)
 - la necessità di effettuare operazioni aggiuntive per mantenere aggiornati i dati derivati.

La decisione di mantenere o eliminare una ridondanza viene presa confrontando il costo delle operazioni che coinvolgono le informazioni ridondanti e lo spazio di archiviazione necessario, nel caso di presenza o assenza di ridondanza.

Esempio di analisi delle ridondanze

- Operazione 1: Aggiungere una nuova persona con la città di residenza della persona.
- Operazione 2: Stampare tutti i dati di una città (incluso il numero di abitanti).



Siccome dominano gli aggiornamenti rispetto agli inserimenti, forse conviene rimuovere la ridondanza. Tuttavia bisogna valutare anche i volumi: senza ridondanza, ogni volta che si fa l'operazione 2, sarà necessario interrogare ogni volta 5000 (una città ha mediamente 5000 abitanti) coppie della relazione residenza.

Per risolvere proseguiamo con lo studio dei due casi usando le tabelle:

Con ridondanza

Operation 1			
Concept	Type	Accesses	Type
Person	Entity	1	W
Residence	Relationship	1	W
Town	Entity	1	R
Town	Entity	1	W

Operation 2

Concept	Type	Accesses	Type
Town	Entity	1	R

Senza ridondanza

Operation 1			
Concept	Type	Accesses	Type
Person	Entity	1	W
Residence	Relationship	1	W

Operation 2

Concept	Type	Accesses	Type
Town	Entity	1	R
Residence	Relationship	5000	R

Operazione 1:

Operazione 1:

- Inserisco la nuova persona
- Inserisco la nuova coppia nella relazione di Residenza
- Leggo il valore dell'attributo numero di abitanti
- Aggiornare l'attributo numero di abitanti

Operazione 2:

- Leggo città

- Inserisco la nuova persona
- Inserisco la nuova coppia nella relazione di Residenza

Operazione 2:

- Leggo città
- 5000 letture sulla relazione residenza

- Presenza di ridondanza.

- L'operazione 1 richiede un totale di 1500 accessi in scrittura e 500 accessi in lettura al giorno.
- Il costo dell'operazione 2 è quasi trascurabile.
- Contando due volte gli accessi in scrittura, abbiamo un totale di 3500 accessi al giorno.

- Assenza di ridondanza.

- L'operazione 1 richiede un totale di 1000 accessi in scrittura al giorno.
- Tuttavia, l'operazione 2 richiede un totale di 10000 accessi in lettura al giorno.
- Contando due volte gli accessi in scrittura, abbiamo un totale di 12000 accessi al giorno.

Dunque vale la pena mantenere i dati ridondanti.

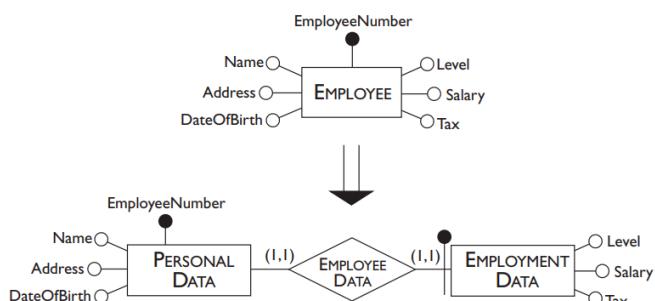
2. Rimozione delle generalizzazioni

Per rimuovere una generalizzazione in uno schema ER, è necessario decomporre la struttura gerarchica in **entità distinte** o **modificare il modello** in modo che non sia più necessaria la generalizzazione. Ci sono tre alternative:

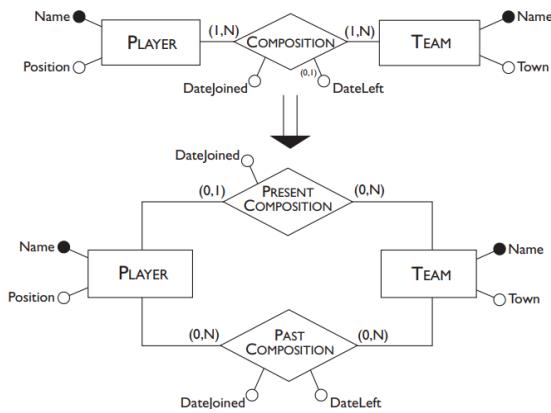
1. Butto via i figli e tengo il genitore, che eredita le informazioni che avevano i figli (utile nei casi in cui le operazioni non sfruttano direttamente i figli);
2. Butto via il genitore e tengo i figli, che ereditano le informazioni che aveva il genitore (utile nei casi in cui le operazioni non sfruttano direttamente il genitore);
3. Tengo sia genitore che figli e li lego attraverso qualcosa.
4. Elimino parte dei figli;

3.1. Partizionamento delle entità

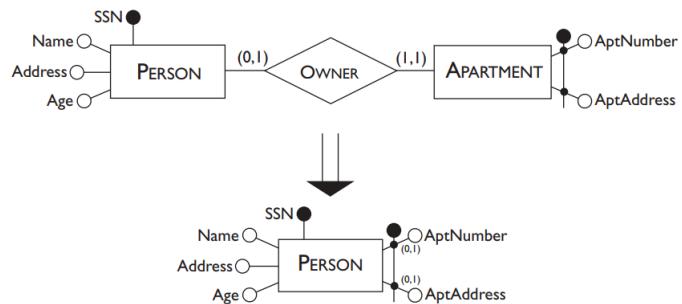
Può essere comodo dividere entità molto ricche di attributi inserendo un'entità devole che li mantenga. Ad esempio:



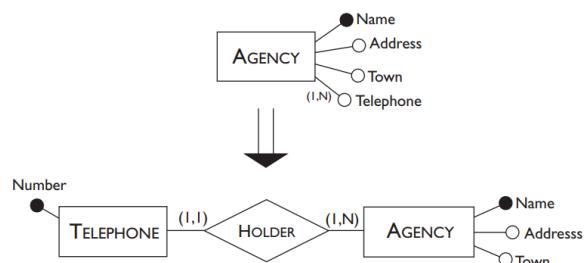
3.2. Partizionamento delle relazioni



3.3. Unione delle entità



3.4. Rimozione attributi multivalore



4. Scelta della chiave primaria

- I criteri per questa decisione sono i seguenti:
 - Gli attributi con valori nulli non possono costituire identificatori primari.
 - Uno o pochi attributi sono preferibili a molti attributi.
 - Un identificatore interno con pochi attributi è preferibile a uno esterno (potrebbe coinvolgere molte entità).
 - Un identificatore utilizzato da molte operazioni per accedere alle occorrenze di un'entità è preferibile ad altri.
- A questo punto, se nessuno degli identificatori candidati soddisfa i requisiti sopra elencati, è possibile introdurre un ulteriore attributo nell'entità. Questo attributo conterrà valori speciali (spesso chiamati codici) generati esclusivamente allo scopo di identificare le occorrenze dell'entità.

Esempio

Modello ER

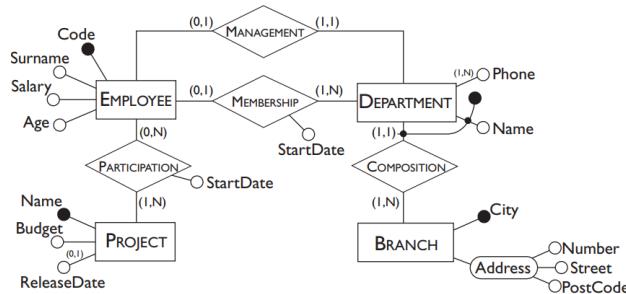


Tabella dei volumi e delle operazioni

È importante mantenere la coerenza con le **regole di consistenza**: se a.e. c'è un vincolo (1,1) che lega department alla relazione management, e viene detto che il numero di dipartimenti è 80, allora necessariamente anche nella tabella management dovranno esserci 80 istanze.

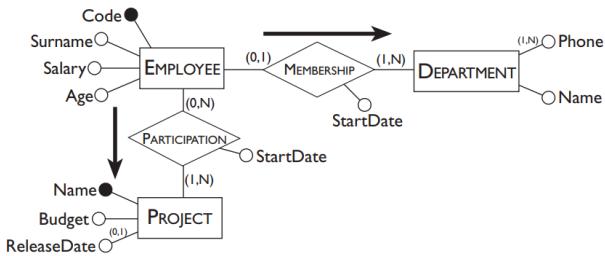
Table of volumes		
Concept	Type	Volume
Branch	E	10
Department	E	80
Employee	E	2000
Project	E	500
Composition	R	80
Membership	R	1900
Management	R	80
Participation	R	6000

Table of operations		
Operation	Type	Frequency
Operation 1	I	50 per day
Operation 2	I	100 per day
Operation 3	I	10 per day
Operation 4	B	2 per day

La tabella delle operazioni contiene le operazioni che si vogliono fare nel database: possono essere di tipo Interactive (quando viene fatta la richiesta, deve essere subito soddisfatta) o Batch (non c'è fretta nel dare la risposta). Ad esempio:

- op. 1: Assegna un impiegato ad un progetto (Inserimento);

- op. 2: Trova i dati di un impiegato del dipartimento dove lavora e dei progetti ai quali partecipa (Interrogazione);
 - vado nella tabella impiegato a prendere un'occorrenza di impiegato;
 - vado nella tabella dipartimento a prendere un'occorrenza di dipartimento;
 - Per farlo devo leggere un'istanza della relazione di afferenza;
 - vado nella tabella progetto a recuperare i progetti a cui lavora



Concept	Type	Accesses	Type
Employee	Entity	1	R
Membership	Relationship	1	R
Department	Entity	1	R
Participation	Relationship	3	R
Project	Entity	3	R

Traduzione di schemi ER in schema relazionale

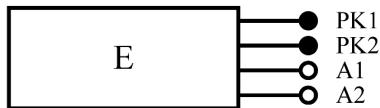
Entità



$E(\underline{PK}, A1, A2)$

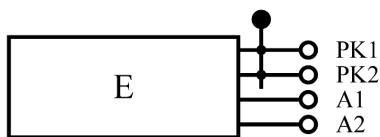
Il vincolo NOT NULL
imposte che l'attributo
non sia nullo

Mettere NOT NULL ad A2
arebbe sbagliato in quanto
nel ER, c'è presente (0, ...)

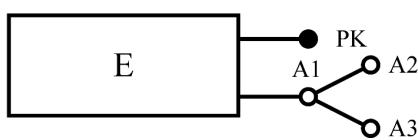


$E(\underline{PK1}, \underline{PK2}, A1, A2)$

NOT NULL
UNIQUE
Per garantire i vincoli di
chiave anche a PK2



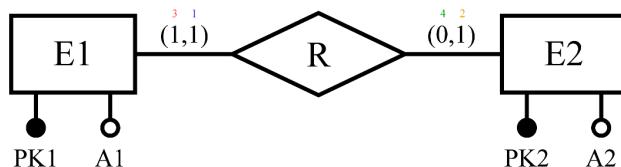
$E(\underline{PK1}, \underline{PK2}, A1, A2)$



$E(\underline{PK}, A2, A3)$ oppure $E(\underline{PK}, A1)$

Relazioni

Casi 1 a 1

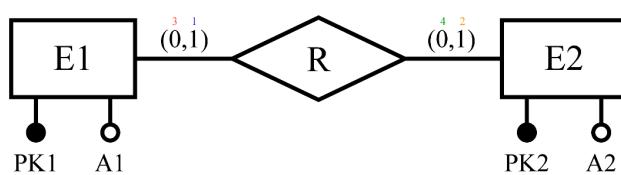


$E1(\underline{PK1}, A1, \underline{PK2})$

1 UNIQUE
3 NOT NULL
FOREIGN KEY

$E2(PK2, A2)$

2 GARANTITA DAL
VINOLO UNICO
DELLA CHIAVE



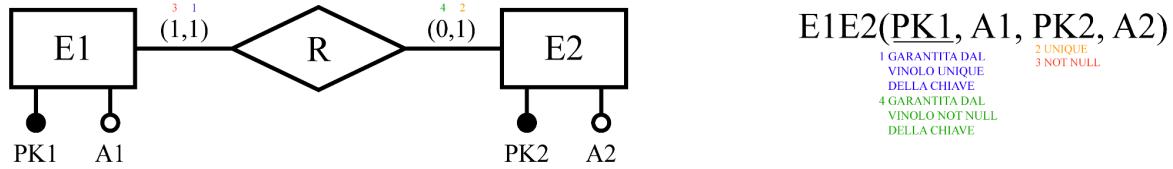
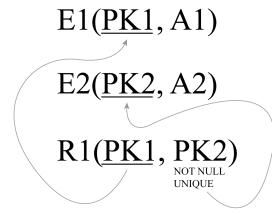
$E1(\underline{PK1}, A1, \underline{PK2})$

1 UNIQUE
FOREIGN KEY

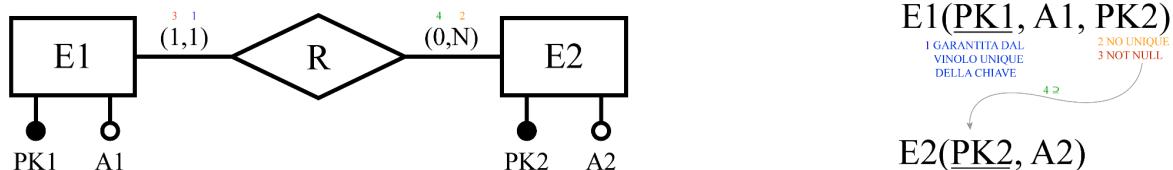
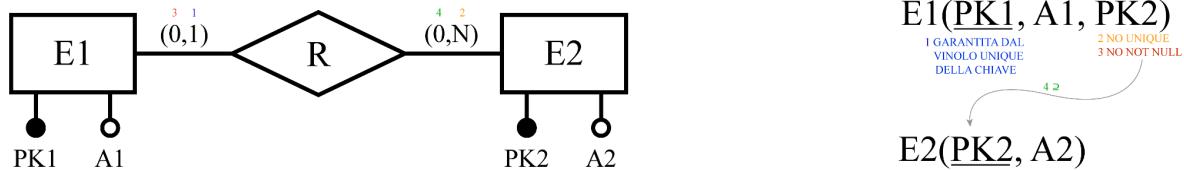
$E2(PK2, A2)$

2 GARANTITA DAL
VINOLO UNICO
DELLA CHIAVE

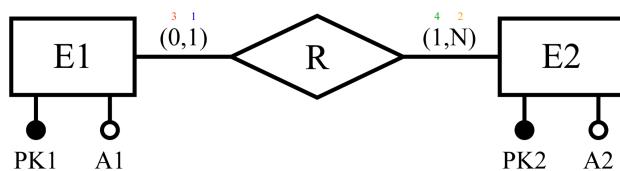
Nel caso in cui la partecipazione di E1 e E2 è molto bassa, si può usare la soluzione:



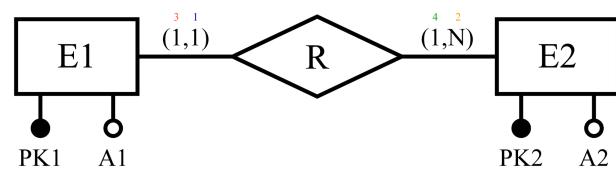
Casi 1 a N



In questo caso non è possibile forzare il vincolo 4 direttamente dallo schema. Occorre imporre in maniera esplicita.



Anche in questo caso non è possibile forzare il vincolo 4 direttamente dallo schema. Occorre imporre in maniera esplicita.



Casi N a N

...

Forme normali e normalizzazioni

Una forma normale è una **proprietà** di un database relazionale, concretamente è una **struttura** (delle tabelle) che rispetta certe proprietà che garantisce l'**assenza di ridondanze**. Quando una relazione non è normalizzata (cioè non soddisfa una forma normale), presenta ridondanze e comportamenti indesiderati durante le operazioni di aggiornamento.

La **normalizzazione** è una procedura che permette di trasformare gli schemi non normalizzati in nuovi schemi in cui è garantita la soddisfazione di una forma normale.

Esempio di relazione con anomalie

Immaginiamo che la chiave della tabella sia la coppia Employee + Project. Notiamo come ci sia una ridondanza ad esempio per Green e per il suo Salary: potrebbe accadere che venga aggiunta una nuova riga per Green con un Salary diverso da 55.

Employee	Salary	Project	Budget	Function
Brown	20	Mars	2	technician
Green	35	Jupiter	15	designer
Green	35	Venus	15	designer
Hoskins	55	Venus	15	manager
Hoskins	55	Jupiter	15	consultant
Hoskins	55	Mars	2	consultant
Moore	48	Mars	2	manager
Moore	48	Venus	15	designer
Kemp	48	Venus	15	designer
Kemp	48	Jupiter	15	manager

Perché non va bene?

La **forma normale** è una struttura delle tabelle, nella quale il rispetto del vincolo di chiave (e dei vincoli imposti sulle tabelle), garantiscono il rispetto del significato delle tabelle. In questo caso non è così dato che:

- Il valore dello stipendio di ciascun dipendente è ripetuto in tutti i tuple ad esso correlati: quindi c'è una ridondanza.
- Se lo stipendio di un dipendente cambia, dobbiamo modificare il valore in tutti i tuple corrispondenti. Questo problema è noto come anomalia di aggiornamento.
- Se un dipendente smette di lavorare su tutti i progetti ma non lascia l'azienda, tutti i tuple corrispondenti vengono eliminati e quindi, anche le informazioni di base, come nome e stipendio, vengono perse. Questo problema è noto come anomalia di cancellazione.
- Se abbiamo informazioni su un nuovo dipendente, non possiamo inserirle finché il dipendente non viene assegnato a un progetto. Questo è noto come anomalia di inserimento.

Il problema è l'esistenza di **dipendenze funzionali** da parte di attributi che non fanno parte delle chiavi che non sono dipendenze funzionali dall'intera chiave ma solo da porzioni di essa.

La soluzione è la **decomposizione** della tabella.

Le dipendenze funzionali

Dato:

- una relazione r su uno schema R(X)
- due sottoinsiemi non vuoti Y e Z degli attributi X,

diciamo che c'è una **dipendenza funzionale** su r tra Y e Z, se, per ogni coppia di tuple t_1 e t_2 di r che hanno gli stessi valori sugli attributi Y, t_1 e t_2 hanno anche gli stessi valori degli attributi Z. Una dipendenza funzionale tra gli attributi Y e Z è indicata dalla notazione $Y \rightarrow Z$. In altre parole:

se due righe coincidono sull'antecedente (a.e. Green), allora devono coincidere anche sul conseguente (a.e. 35)

Nell'esempio sopra, c'è una dipendenza funzionale $\text{Employee} \rightarrow \text{Salary}$, dove Y = Employee e Z = Salary, dato che, a.e. $t_1 = \text{Green}$ e $t_2 = \text{Green}$ hanno gli stessi valori sugli attributi Y = Green, t_1 e t_2 hanno anche gli stessi valori degli attributi Z = 35.

Le dipendenze funzionali banali e non

Diciamo quindi che una dipendenza funzionale $Y \rightarrow Z$ è non banale se nessun attributo in Z appare tra gli attributi di Y.

- $\text{Employee} \rightarrow \text{Salary}$ è una dipendenza funzionale non banale.
- $\text{Employee Project} \rightarrow \text{Project}$ è una dipendenza funzionale banale.

Assiomi di Armstrong

Gli assiomi di Armstrong sono un insieme di regole o proprietà che permettono di dedurre ulteriori dipendenze funzionali da un insieme dato di dipendenze funzionali in un database. Queste regole sono utilizzate nell'algoritmo di chiusura di Armstrong, che è un metodo per determinare tutte le dipendenze funzionali implicite da un insieme di dipendenze funzionali date. Gli assiomi di Armstrong sono:

1. **Riflessività**: Se Y è un sottoinsieme di X, allora X determina Y. (Se $X \rightarrow Y$, allora $X \rightarrow Y$);
2. **Transitività**: Se X determina Y e Y determina Z, allora X determina Z. (Se $X \rightarrow Y$ e $Y \rightarrow Z$, allora $X \rightarrow Z$);
3. **Aumento**: Se X determina Y e X determina Z, allora X determina YZ. (Se $X \rightarrow Y$ e $X \rightarrow Z$, allora $X \rightarrow YZ$);

Queste regole possono essere applicate in modo iterativo per derivare tutte le dipendenze funzionali implicite da un insieme di dipendenze funzionali dato.

Boyce–Codd Normal Form (BCNF)

La Boyce-Codd Normal Form (BCNF) è una **forma normale** in teoria dei database relazionali.

Una relazione si trova in BCNF se ogni dipendenza funzionale non banale $X \rightarrow Y$, l'antecedente è una superchiave ossia contiene la chiave K della relazione.

Nell'esempio sopra, questa condizione non è soddisfatta in quanto, nella dipendenza:

- **Employee → Salary**: Employee non rispetta questa condizione (non contiene la chiave K della relazione)
- **Project → Budget**: Project non rispetta questa condizione (non contiene la chiave K della relazione)

Note

- Relazioni (Tabelle) con due attributi sono sempre in BCNF;
- Una relazione in BCNF non contiene ridondanze: non è possibile

Grazie al primo punto di questa sezione, è **sempre** possibile ricondursi alla BCNF, iterando un appropriato numero di volte la decomposizione delle tabelle fino ad avere tabelle con soli due attributi, che, per banalità, sono in BCNF.

Esempio

No BCNF	BCNF
---------	------

Employee	Salary	Project	Budget	Function
Brown	20	Mars	2	technician
Green	35	Jupiter	15	designer
Green	35	Venus	15	designer
Hoskins	55	Venus	15	manager
Hoskins	55	Jupiter	15	consultant
Hoskins	55	Mars	2	consultant
Moore	48	Mars	2	manager
Moore	48	Venus	15	designer
Kemp	48	Venus	15	designer
Kemp	48	Jupiter	15	manager

Employee	Salary
Brown	20
Green	35
Hoskins	55
Moore	48
Kemp	48

Project	Budget
Mars	2
Jupiter	15
Venus	15

Employee	Project	Function
Brown	Mars	technician
Green	Jupiter	designer
Green	Venus	designer
Hoskins	Venus	manager
Hoskins	Jupiter	consultant
Hoskins	Mars	consultant
Moore	Mars	manager
Moore	Venus	designer
Kemp	Venus	designer
Kemp	Jupiter	manager

Quella a destra è in BCNF perché, tenendo a mente che la chiave è Employee + Project:

- **Employee Project → Function:** l'antecedente Employee Project è la chiave;
- **Project → Budget:** l'antecedente Project è la chiave;
- **Employee → Salary:** l'antecedente Employee è la chiave;

Scomposizioni in BCNF problematiche: Lossless decomposition

Supponendo la chiave come Employee Project, facciamo la decomposizione in BCNF della seguente tabella. In alcuni casi, questa trasformazione può comportare la **perdita di informazioni** e la conseguente **mancanza di possibilità di ricostruire** completamente i dati originali dalla relazione trasformata.

Tabella di partenza

Employee	Project	Branch
Brown	Mars	Chicago
Green	Jupiter	Birmingham
Green	Venus	Birmingham
Hoskins	Saturn	Birmingham
Hoskins	Venus	Birmingham

Tabella in BCNF

Employee	Branch
Brown	Chicago
Green	Birmingham
Hoskins	Birmingham

Project	Branch
Mars	Chicago
Jupiter	Birmingham
Saturn	Birmingham
Venus	Birmingham

Tabella ricostruita

Employee	Project	Branch
Brown	Mars	Chicago
Green	Jupiter	Birmingham
Green	Venus	Birmingham
Hoskins	Saturn	Birmingham
Hoskins	Venus	Birmingham
Green	Saturn	Birmingham
Hoskins	Jupiter	Birmingham

Una decomposizione di una tabella r in X_1 e X_2 è detta **lossless decomposition** se l'operazione di join (fusione) sulle proiezioni di r di X_1 e X_2 è uguale ad r stessa.

Condizione sufficiente per avere una lossless decomposition

- Sia r una relazione su X e siano X_1 e X_2 due sottoinsiemi di X tali che $X_1 \cup X_2 = X_0$ (sia quindi $X_0 = X_1 \cap X_2$);
- Se r soddisfa la dipendenza funzionale $X_0 \rightarrow X_1$ o la dipendenza funzionale $X_0 \rightarrow X_2$, allora la scomposizione di r su X_1 e X_2 è lossless.

Tabella di partenza

Employee	Project	Branch
Brown	Mars	Chicago
Green	Jupiter	Birmingham
Green	Venus	Birmingham
Hoskins	Saturn	Birmingham
Hoskins	Venus	Birmingham

Tabella in BCNF

Employee	Branch
Brown	Chicago
Green	Birmingham
Hoskins	Birmingham

Employee	Project
Brown	Mars
Green	Jupiter
Green	Venus
Hoskins	Saturn
Hoskins	Venus

Tabella ricostruita

Employee	Project	Branch
Brown	Mars	Chicago
Green	Jupiter	Birmingham
Green	Venus	Birmingham
Hoskins	Saturn	Birmingham
Hoskins	Venus	Birmingham

Un altro problema con la nuova decomposizione sorge quando si considerano gli aggiornamenti al database. Ad esempio, supponiamo di voler inserire una nuova tupla che specifica la partecipazione dell'impiegato di nome Armstrong, che lavora a Birmingham, nel progetto Mars. Nella relazione originale, questo aggiornamento sarebbe immediatamente identificato come illegale perché causerebbe una violazione della dipendenza tra gli attributi Progetto e Filiale. Tuttavia, nelle relazioni decomposte, diventa difficile individuare eventuali violazioni di dipendenza poiché gli attributi Progetto e Filiale sono stati separati in relazioni diverse.

Questa separazione oscura la dipendenza originale, rendendo più difficile imporre vincoli e garantire l'integrità dei dati durante gli aggiornamenti.

Preservazione delle dipendenze

Una decomposizione **preserva le dipendenze** se ciascuna delle dipendenze funzionali dello schema originale coinvolge attributi che compaiono tutti insieme in uno degli schemi decomposti. È ovviamente auspicabile che una decomposizione preservi le dipendenze poiché, in questo modo, è possibile garantire, sullo schema decomposto, il soddisfacimento degli stessi vincoli dello schema originale.

Vediamo ora un caso problematico: assumiamo che siano definite le seguenti dipendenze:

- **Manager → Branch**: ogni manager lavora presso una specifica filiale;
- **Project Branch → Manager**: ogni progetto ha più manager responsabili per esso, ma in diverse filiali, e ogni manager può essere responsabile per più di un progetto; tuttavia, per ogni filiale, un progetto ha solo un manager responsabile per esso.

Manager	Project	Branch
Brown	Mars	Chicago
Green	Jupiter	Birmingham
Green	Mars	Birmingham
Hoskins	Saturn	Birmingham
Hoskins	Venus	Birmingham

La relazione presenta delle **ridondanze pericolose** dato che non è nella forma normale di Boyce-Codd (il lato sinistro della prima dipendenza non è una superchiave). Dunque si potrebbe pensare di **decomporre**, avendo poi due tabella.

Tuttavia, **non è possibile** ottenere una buona decomposizione di questa relazione: la dipendenza **Project Branch → Manager** coinvolge tutti gli attributi e quindi nessuna decomposizione è in grado di preservarla e andrebbe persa.

Possiamo quindi affermare che a volte la forma normale di Boyce-Codd non può essere raggiunta.

La Terza Forma Normale

La Terza Forma Normale (3NF) è una forma normale più **debole** rispetto alla BCNF perché non richiede che ogni dipendenza funzionale non banale coinvolga solo superchiavi, come richiesto dalla BCNF. Questo significa che una relazione può essere in 3NF anche se contiene dipendenze funzionali non banali che coinvolgono attributi non chiave.

Asserisce volgarmente che la tabella:

- o soddisfa la condizione imposta dalla BCNF
- o ...

Una relazione r si trova in 3NF se soddisfa almeno una delle due condizioni:

- X contiene una chiave K di r;
- ogni attributo in Y è contenuto in almeno una chiave di r;

Rivediamo perché questa definizione soddisfa l'esempio sopra: ricordiamo che le due chiavi sono Project + Branch e Manager + Project:

- **Project Branch → Manager:** Soddisfa BCNF quindi apposto;
- **Manager → Branch:** non soddisfa BCNF ma c'è un unico attributo nel conseguente Branch e Branch fa parte della chiave Project + Branch

La Terza Forma Normale è meno restrittiva della Forma Normale di Boyce-Codd e per questo motivo non offre le stesse garanzie di qualità per una relazione; ha però il vantaggio di essere **sempre raggiungibile**.

Decomposizione in 3NF

Per decomporre una tabella in Terza Forma Normale, è necessario introdurre una tabella per ogni dipendenza funzionale. Ad esempio, data la tabella con chiave Employee + Project:

Employee	Salary	Project	Budget	Function
Brown	20	Mars	2	technician
Green	35	Jupiter	15	designer
Green	35	Venus	15	designer
Hoskins	55	Venus	15	manager
Hoskins	55	Jupiter	15	consultant
Hoskins	55	Mars	2	consultant
Moore	48	Mars	2	manager
Moore	48	Venus	15	designer
Kemp	48	Venus	15	designer
Kemp	48	Jupiter	15	manager

con le dipendenze:

- Employee → Salary;
- Project → Budget;
- Employee Project → Function;

Verranno introdotte tre tabelli:

- una contenente Employee e Salary;
- una contenente Project e Budget;
- una contenente Employee Project e Function;

Mantenendo l'antecedente (Employee, Project , Employee Project) come chiave, le dipendenze sono state preservate.

Esempio

Dipendenze funzionali:

- **Manager → Branch Division:** ogni manager lavora in una filiale e gestisce una divisione.
- **Branch Division → Manager:** per ogni filiale e divisione c'è un unico manager.
- **Project Branch → Division:** per ogni filiale, un progetto è allocato a una singola divisione ed ha un solo manager responsabile.

Tabella problematica

Manager	Project	Branch	Division
Brown	Mars	Chicago	1
Green	Jupiter	Birmingham	1
Green	Mars	Birmingham	1
Hoskins	Saturn	Birmingham	2
Hoskins	Venus	Birmingham	2

Tabella in 3NF ✨

Manager	Branch	Division
Brown	Chicago	1
Green	Birmingham	1
Hoskins	Birmingham	2

Project	Branch	Division
Mars	Chicago	1
Jupiter	Birmingham	1
Mars	Birmingham	1
Saturn	Birmingham	2
Venus	Birmingham	2

La decomposizione è senza perdita e le dipendenze sono preservate. Questo esempio dimostra che spesso la difficoltà nel raggiungere la forma normale di Boyce-Codd potrebbe essere dovuta a un'analisi dell'applicazione insufficientemente accurata.

Algebra relazionale

- Proprietà di chiusura dell'algebra relazionale

L'algebra relazionale è un linguaggio **procedurale** i cui operatori ricevono in **ingresso** una o più **relazioni** e producono in **uscita** una **relazione**. Ciò consente di comporre gli operatori dell'algebra relazionale per formulare interrogazioni complesse.

- L'esecuzione di una interrogazione (query) **non modifica il contenuto della base di dati** (differentemente da quanto accade con gli aggiornamenti (update)).
- Gli operatori di base dell'algebra relazionale selezionano/estraggono le informazioni di interesse **presenti** nella base di dati. Non avviene nessuna derivazione.

Limiti dell'algebra relazionale

1. Supporta solo relazioni **finite** (**no complementazione**):

Immaginiamo di avere una relazione R con gli attributi A_i , ossia $R(A_1, A_2, \dots, A_n)$. Chiaramente ci saranno delle istanze $r \in R$ costruite prendendo ogni A_i all'interno del proprio dominio $\text{dom}(A_i)$ e quindi si ha che:

$$r \subseteq \text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n)$$

Nel concreto immaginiamo di avere una tabella **PIACE** con colonne **BAMBINO**, **GUSTO**, **GELATO**. I valori degli attributi vengono pescati rispettivamente dai propri domini, quindi, a.e. l'attributo **GUSTO** viene pescato nell'insieme finito $\text{dom}(\text{GUSTO})$.

PIACE(BAMBINO, GUSTO, GELATO)

Se si volesse trovare quali sono i gusti che non piacciono ai bambini, si potrebbe creare una nuova tabella **NON_PIACE** attraverso un'operazione di **complementazione** rispetto all'universo $\text{dom}(\text{BAMBINO}) \times \text{dom}(\text{GUSTO})$.

NON_PIACE(BAMBINO, GUSTO, GELATO)

In questo caso, l'operazione è fattibile perché sia $\text{dom}(\text{BAMBINO})$ che $\text{dom}(\text{GUSTO})$ sono di **cardinalità finita**. Tuttavia, se anche solo uno dei due domini per i quali si tenta di fare la complementazione fosse infinito, questa operazione non si potrebbe fare perché potrebbe **violare le condizioni di finitezza** delle relazioni.

Per questo motivo in questi casi, è meglio usare la **differenza insiemistica**.

2. Solo **informazione positiva e completa**:

Immaginiamo di avere una gara con tre atleti A , B e C senza parimeriti e immaginiamo di voler registrare l'informazione "ordine di arrivo" con la tabella "<"

É_ARRIVATO_PRIMA_DI(PRIMA, DOPO)

Che contiene le istanze $\{(B, A), (C, A)\}$ (dunque B è arrivato prima di A e C è arrivato prima di A) ma se proseguissi con il complementare (fattibile dato che l'universo è finito) avrei la tabella:

É_ARRIVATO_DOPPIO_DI(PRIMA, DOPO)

Che conterebbe le istanze $\{(A, B), (A, C), (B, C), (C, B)\}$ e si avrebbe una inconsistenza.

3. Non si possono esprimere **chiusure transitive** (no ricorsione).

Ad esempio, in questo **database**, non è possibile scrivere un'interrogazione del tipo "dato un impiegato tornare tutti i suoi supervisori diretti o indiretti". Altre interrogazioni che non possono essere scritte sono:

- dato un oggetto composto, quali sono le componenti diretti o indiretti che lo formano;
- dato un fiume, quali sono i suoi affluenti diretti o indiretti;
- dato un nodo in un grafo, quali sono i nodi raggiungibili a partire da quel nodo;
- dato un aeroporto, quali sono gli aeroporti che posso raggiungere in una o più tratte;

Le operazioni di base

Le operazioni di base (o primitive) comprendono:

1. Operazioni di selezione (σ)

Sintassi: $T \leftarrow \sigma_{cond}(R)$

Significato: prendi la tabella R e, riga per riga, vedi se l'attributo STIPENDIO è maggiore di 50.000. Se è così restituisci quell'impiegato nella tabella risultato T, altrimenti buttalo via.

Note:

- $cond$ può essere qualsiasi espressione booleana a.e.: $T \leftarrow \sigma_{STIPENDIO > 50.000 \text{ AND GENERE} = M}(R)$;
- Questa operazione non cambia il **grado** (numero degli attributi);
- La **cardinalità** di R sarà sempre maggiore o uguale rispetto alla cardinalità di T;
- L'**indice di selettività** (cardinalita(T)/cardinalità(R)) si abbassa e l'ottimizzatore gode, dunque si cerca sempre di applicare per prime le condizioni più selettive (quelle che tagliano più roba);
- Vale la proprietà di **commutatività**: $\sigma_{cond2}(\sigma_{cond1}(R)) = \sigma_{cond1}(\sigma_{cond2}(R))$;

proiezione (π)

Sintassi: $T \leftarrow \pi_{<\text{lista di attributi}>}(R)$

Significato: per ogni riga di R restituisci solo le colonne definite in $<\text{lista di attributi}>$ eliminando le altre.

Esempio: data la tabella EMPLOYEE(ID, NAME, DEPARTMENT, SALARY), puoi utilizzare l'operazione di proiezione per estrarre solo alcune colonne, ad esempio solo ID e NAME, ignorando DEPARTMENT e SALARY.

Note:

- grado(T) \leq grado(R);
- card(T) \leq card(R);
- non vale sempre la commutatività;

2. Operazioni di rinomina (ρ)

$T \leftarrow \sigma_{DIP = 5}(IMPIEGATO)$

$R \leftarrow \pi_{CF, STIPENDIO}(T)$

Data la tabella IMPIEGATO(NOME5, COGNOME5, CF5, STIPENDIO5, ...), Se si volesse assegnare come nuovi nomi degli attributi di IMPIEGATO nella relazione T i nomi originali con 5 alla fine:

$T(NOME5, COGNOME5, CF5, STIPENDIO5, ...) \leftarrow \sigma_{DIP = 5}(IMPIEGATO)$

Notazione alternativa: $\rho_{(NOME5, COGNOME5, CF5, STIPENDIO5, ...)}(T)$

3. Operazione di unione insiemistica (\cup)

L'operazione di unione consente di combinare due relazioni **compatibili** ovvero con

1. **Stesso grado:** le due relazioni devono avere lo stesso numero di attributi;
2. **Stesso significato per gli attributi;**
3. **Stesso nome per gli attributi** (SQL non ha questo vincolo, in algebra relazionale è aggirabile con la rinnomina);

in una nuova relazione che contiene **tutte** le tuple presenti in entrambe le relazioni di partenza, eliminando duplicati.

La sintassi è questa:

$$R \leftarrow T_1 \cup T_2$$

Esempio

Determinare il CF di tutti gli impiegati che o lavorano per il dipartimento 5, o sono il supervisore (diretto) di un impiegato che lavora per il dipartimento 5.

1. Estraggo gli impiegati che sono nel dipartimento 5, filtro le colonne mantenendo solo quella relativa ai CF e salvo in T_1

$$T_1 \leftarrow \pi_{CF} (\sigma_{DIP=5}(IMPIEGATO))$$

2. Estraggo gli impiegati che sono nel dipartimento 5, filtro le colonne mantenendo solo quella relativa ai supervisori e rinomino tale colonna in "CF" (era SUPERVISORE prima). Poi salvo in T_2

$$T_2(CF) \leftarrow \pi_{SUPERVISORE} (\sigma_{DIP=5}(IMPIEGATO))$$

3. Dato che T_1 e T_2 soddisfano le tre proprietà di compatibilità, posso fare l'unione

$$R \leftarrow T_1 \cup T_2$$

4. Operazione di differenza insiemistica (\setminus o -)

La differenza è un operatore insiemistico binario dell'algebra relazionale che contiene le tuple presenti nella relazione r1 ma non nella relazione r2.

Sintassi: $R \leftarrow CANDIDATI - NO_GOOD$

- Non vale la commutatività;
- Non vale l'associatività;
- Il grado delle due tabelle non varia;
- Cardinalità di T \leq Cardinalità di R;

Esempio

Trovare tutti gli impiegati senza persone a carico.

Risulta facile trovare gli impiegati che hanno delle persone a carico. Una volta fatto ciò, si può fare la differenza insiemistica, tra la tabella iniziale e quella contenente solo impiegati che hanno persone a carico.

1. Costruiamo l'insieme dei candidati

$$CANDIDATI \leftarrow \pi_{CF}(IMPIEGATO)$$

2. Cerco i **no good**, ossia, impiegati che hanno persone a carico. Una volta fatto, faccio una riconomina per soddisfare le tre proprietà di compatibilità

$$NO_GOOD(CF) \leftarrow \pi_{IMP}(PERSONA_A_CARICO)$$

3. Ora procedo con la differenza insiemistica

$$R \leftarrow CANDIDATI - NO_GOOD$$

5. Operazione di prodotto cartesiano (\times)

Il prodotto cartesiano è un operatore binario insiemistico dell'algebra relazionale che, date due relazioni S e T , crea una relazione composta da **tutte le combinazioni possibili** delle tuple della prima con le tuple della seconda. Il simbolo del prodotto cartesiano è \times .

$$R \leftarrow S \times T$$

Esempio

Ho due tabelle S e T .

S		T	
a	b	c	d
a	100	a	100
b	300	b	200
c	400	f	400
d	200	g	120
e	150		

Calcolo il prodotto cartesiano delle due tabelle.

$$R \leftarrow S \times T$$

Il prodotto cartesiano è una tabella (relazione) composta da tutte le tuple della prima relazione combinate con tutte le relazioni della seconda.

$S \times T$

a	b	c	d
a	100	a	100
a	100	b	200
a	100	f	400
a	100	g	120
b	300	a	100
b	300	b	200
b	300	f	400
b	300	g	120
c	400	a	100
c	400	b	200
c	400	f	400
c	400	g	120
d	200	a	100
d	200	b	200
d	200	f	400
d	200	g	120
e	150	a	100
e	150	b	200
e	150	f	400
e	150	g	120

Il conflitto di nomi negli attributi

Se le relazioni S e T hanno degli attributi con lo stesso nome, per convenzione nel prodotto cartesiano i campi corrispondenti sono indicati senza alcun nome.

S

a	b
a	100
b	300
c	400
d	200
e	150

T

c	b
a	100
b	200
f	400
g	120

S x T

a		c	
a	100	a	100
a	100	b	200
a	100	f	400
a	100	g	120
b	300	a	100
b	300	b	200
b	300	f	400
b	300	g	120
c	400	a	100
c	400	b	200
c	400	f	400
c	400	g	120
d	200	a	100
d	200	b	200
d	200	f	400
d	200	g	120
e	150	a	100
e	150	b	200
e	150	f	400
e	150	g	120

Per risolvere il conflitto di nomi ci sono due soluzioni:

1. Riconoscere gli attributi di origine in base alla loro **posizione** nello schema e rinominarli (**ex post**).
2. Rinominare gli attributi con lo stesso nome nello schema di origine **prima del prodotto** cartesiano (**ex ante**).

Operazioni derivate

1. Operazioni di theta join

Il theta join, anche conosciuto come **join condizionale** o join con predicato, è un tipo di operazione di join nei database relazionali in cui viene specificata una **condizione di join esplicita** (diversamente dal natural join che utilizza automaticamente colonne con lo stesso nome).

Il join tra due relazioni R e S equivale a una **selezione sul prodotto cartesiano** delle relazioni $R \times S$:

$$R \bowtie_{\langle cond \rangle} S = \sigma_{\langle cond \rangle}(R \times S)$$

Nel theta join, puoi specificare una condizione arbitraria che **determina** le righe da unire tra due tabelle. Questa condizione può coinvolgere qualsiasi combinazione di colonne e operatori logici. Ecco come funziona in modo più dettagliato:

- Consideriamo due tabelle, Tabella A e Tabella B.
- Viene specificata una condizione di join usando un predicato (ad esempio, uguaglianza, confronto, ecc.).
- Il theta join seleziona le righe da entrambe le tabelle che soddisfano la condizione specificata e le unisce.

Caso particolare: equi join

Se la condizione è composta da **operatori di uguaglianza** ($=$), eventualmente in congiunzione (and) tra loro, il join condizionale è detto **equi join**. Di fatto l'equi join è simile a un join naturale in cui nella condizione di join sono ammessi gli attributi con nomi diversi e le costanti.

Esempio

Ho due tabelle Azienda e Edificio.

Azienda			Edificio	
ufficio	direzione	dipendenti	servizio	piano
contabilità	amministrativo	20	generale	1
ced	generale	15	personale	2
buste paga	personale	10	amministrativo	3
assunzioni	personale	5	marketing	4
vendite	marketing	30		

Utilizzo l'equi-join (theta join con un uguaglianza come condizione) per associare l'ufficio al piano dell'edificio in cui si trova. La condizione di join è l'uguaglianza direzione=servizio.

$$R \leftarrow AZIENDA \bowtie_{\langle direzione = servizio \rangle} EDIFICO$$

E quello che ottengo è:

ufficio	direzione	dipendenti	servizio	piano
contabilità	amministrativo	20	amministrativo	3
ced	generale	15	generale	1
buste paga	personale	10	personale	2
assunzioni	personale	5	personale	2
vendite	marketing	30	marketing	4

Esempio 2

Sulla base delle due tabelle Azienda e Edificio del precedente esempio, si vuole trovare il piano degli uffici con più di 10 dipendenti. La condizione di join è dunque (direzione=servizio) and (dipendenti>10). Non essendo la condizione composta da una congiunzione di uguaglianze, si tratta di un theta join.

$$R \leftarrow AZIENDA \bowtie_{<\text{direzione} = \text{servizio} \wedge \text{dipendenti} > 10} EDIFICO$$

Il risultato finale è la seguente tabella:

ufficio	direzione	dipendenti	servizio	piano
contabilità	amministrativo	20	amministrativo	3
ced	generale	15	generale	1
vendite	marketing	30	marketing	4

2. Operazioni di natural join

Il natural join è un tipo di join usato nei database relazionali che automaticamente crea un join tra due tabelle basandosi sulle colonne che hanno lo **stesso nome** e lo stesso tipo di dati in entrambe le tabelle. La condizione principale del natural join è che i valori nelle colonne con nomi uguali devono essere identici per poter unire le righe delle due tabelle.

$$A \bowtie B$$

Ecco come funziona in termini più dettagliati:

- Si considerano due tabelle, diciamo Tabella A e Tabella B.
- Ogni tabella ha delle colonne, alcune delle quali possono avere nomi in comune con colonne nell'altra tabella.
- Il natural join compara le righe di A e B.

Per ogni coppia di righe (una da A e una da B), il join includerà nella tabella risultante quelle coppie di righe dove i valori nelle colonne con lo stesso nome sono uguali.

Un esempio semplice può aiutare a chiarire:

Supponiamo che Tabella A abbia le colonne ID e Nome, e Tabella B abbia le colonne ID e Età. Un natural join tra A e B produrrà una tabella con colonne ID, Nome, e Età, ma solo per quelle righe dove il valore di ID corrisponde tra le due tabelle originali.

$$A \bowtie B$$

Caso particolare senza attributi in comune

C'è un **caso particolare** quando R ed S non hanno attributi con lo stesso nome ed eseguono un natural join.

$$T \leftarrow R \bowtie S$$

Il natural join diventa, in questo caso, un **prodotto cartesiano**.

Caso particolare con tutti gli attributi in comune

Un altro **caso particolare** si ha con due tabelle che hanno gli stessi attributi. ad esempio $R(A, B)$ e $S(A, B)$. La relazione risultato è

$$R(A, B) \bowtie S(A, B) = T(A, B) = R(A, B) \cap S(A, B)$$

Dunque un'**intersezione** tra R ed S .

Esempio di natural join

Ho due tabelle R e T .

R			T	
a	b	c		
1	a	d	b	
3	c	c	a	100
4	d	f	d	200
5	d	b	f	400
6	e	f	g	120

Il risultato finale è una tabella con lo schema composto dagli attributi delle due relazioni e le tuple (righe) che hanno lo stesso valore nell'attributo in comune.

$$F \leftarrow R \bowtie T$$

E quello che ottengo è:

a	b	c	d
1	a	d	100
4	d	f	200
5	d	b	200

La colonna dell'attributo in comune (b) è presente una sola volta nella tabella del join naturale.

3. Operazione di divisione

La divisione è un operatore binario dell'algebra relazionale che contiene le tuple $\langle x \rangle$ della prima relazione A tali che per ogni tupla $\langle y \rangle$ della seconda relazione B ci sia una tupla $\langle x, y \rangle$ nella prima relazione A.

$$A/B = \{\langle x \rangle \mid \forall \langle y \rangle \in B, \langle x, y \rangle \in A\}$$

Esempio

Ho una relazione A di impiegati e una relazione B degli uffici di un'azienda.

A

impiegato	ufficio
Rossi	marketing
Rossi	vendite
Bianchi	marketing

B

ufficio
marketing
vendite

Per trovare gli impiegati che lavorano in tutti i reparti si ricorre alla divisione A/B. Nella tabella A c'è solo un valore <impiegato> tale che ogni <impiegato, ufficio> è compreso in A.

A

impiegato	ufficio
Rossi	marketing
Rossi	vendite
Bianchi	marketing

B

ufficio
marketing
vendite

Il risultato è una tabella contenente l'impiegato che lavora in entrambi i reparti. In questo caso è l'impiegato Rossi.

impiegato
Rossi

4. Operazione di semi join sinistro (e destro)

Esempio

Riportare le informazioni disponibili nella tabella IMPIEGATO sui manager.

$$R \leftarrow IMPIEGATO \bowtie_{CF = MANAGER} DIPARTIMENTO$$

R contiene tutti e soli gli attributi di IMPIEGATO. Definita in questo modo:

$$R \leftarrow \pi_{<\text{attributi di impiegato}>} (IMPIEGATO \bowtie_{CF = MANAGER} DIPARTIMENTO))$$

Operazioni addizionali

1. Operazione di proiezione generalizzata

È come una proiezione standard, solo che invece di avere un attributo (o più) ha una **funzione**. Ad esempio, data la tabella $\text{IMPIEGATO}(\text{CF}, \text{STIPENDIO}, \text{DEDUZIONE}, \text{ANNI_SERVIZIO})$,

$$\text{REPORT}(\text{CF}, \text{STIPENDO_NETTO}, \text{BONUS}) \leftarrow \pi_{\text{CF}, \text{STIPENDIO} - \text{DEDUZIONI}, 2000 \cdot \text{ANNI_SERVIZIO}}(\text{IMPIEGATO})$$

2. Funzioni aggregate

Le funzioni aggregate in algebra relazionale sono funzioni che **operano su un insieme di valori** all'interno di una relazione e **producono un singolo risultato**. Queste funzioni sono utilizzate per calcolare statistiche, come somme, medie, conteggi, valori massimi o minimi, su un insieme di valori. Alcune delle funzioni aggregate comuni includono:

- **SUM**: Calcola la somma dei valori in un insieme di tuple.
- **AVG**: Calcola la media dei valori in un insieme di tuple.
- **COUNT**: Conta il numero di tuple in un insieme di tuple (utile solo quando occorre contare in modo unbounded).
- **MAX**: Restituisce il valore massimo in un insieme di tuple (si può fare a meno).
- **MIN**: Restituisce il valore minimo in un insieme di tuple (si può fare a meno).

Funzioni aggregate max e min

Immaginiamo di voler trovare gli impiegati che percepiscono lo stipendio massimo Approccio 1: con la funziona maximum:

$$T \leftarrow F_{\text{MAXIMUM STIPENDIO}}(\text{IMPIEGATO})$$

$$R \leftarrow \pi_{\text{CF}}(T \bowtie_{\text{MAX} = \text{STIPENDIO}} \text{IMPIEGATO})$$

Approccio 2: senza la funziona maximum (tieni conto che è una condizione universale, quindi passare per i NOGOOD):

$$\text{IMPIEGATO1}(\text{CF1}, \dots) \leftarrow \text{IMPIEGATO}$$

$$\text{NOGOOD} \leftarrow \pi_{\text{CF}}(\text{IMPIEGATO} \bowtie_{\text{STIPENDIO} < \text{STIPENDIO1}} \text{IMPIEGATO1})$$

$$R \leftarrow \pi_{\text{CF}}(\text{IMPIEGATO}) - \text{NOGOOD}$$

3. Operazione di outer join

Il join esterno mi permette di mantenere le righe di una tabella o entrambe le tabelle. Esistono tre versioni del join esterno:

- **left outer**: Mantiene le righe della 1^a tabella a cui aggiunge le righe della 2^a che soddisfano la condizione di join.
- **right outer**: Mantiene le righe della 2^a tabella a cui aggiunge le righe della 1^a che soddisfano la condizione di join.
- **full outer join**: Mantiene tutte le tuple di entrambe le tabelle, estendendole con valori nulli se necessario.

Esempio

Ho due tabelle Persone e Paternità.

Persone

Nome	Reddito
Mario	25000
Giovanni	15000
Paolo	30000
Giuseppe	20000
Maria	35000

Paternità

Padre	Figlio
Giovanni	Maria
Mario	Paolo
Mario	Giuseppe

Voglio mostrare il nome, il reddito e, se presente, il nome del padre della persona. Per non escludere le righe della prima tabella senza corrispondenza nella seconda utilizzo il LEFT OUTER JOIN. Il join esterno genera una tabella in output con tutti i record della prima tabella.

LEFT JOIN Nome=Figlio

Nome	Reddito	Padre
Mario	25000	
Giovanni	15000	
Paolo	30000	Mario
Giuseppe	20000	Mario
Maria	35000	Giovanni

Esercizi sull'algebra relazionale

Esercizio 1.

Trovare impiegati che afferiscono allo stesso dipartimento.

Utilizziamo coppie ordinate

1. Costruisco una copia di IMPIEGATO chiamata IMPIEGATO1, rinominando tutti gli attributi aggiungendo un 1 alla fine

$$IMPIEGATO1(CF1, NOME1, \dots) \leftarrow IMPIEGATO$$

2. Faccio il prodotto cartesiano, ottenendo tutti i possibili accoppiamenti di impiegati (cioè ogni impiegato sta in coppia con ogni impiegato, incluso se stesso e incluse coppie del tipo (A, B) e (B, A))

$$T \leftarrow IMPIEGATO1 \times IMPIEGATO$$

3. Ora seleziono le coppie in cui il valore DIPARTIMENTO è uguale a quello di DIPARTIMENTO1, in questo modo inseriamo in T_1 solo le coppie di due impiegati che stanno nello stesso dipartimento.

$$T_1 \leftarrow \sigma_{DIP = DIP1}(T)$$

4. In questo modo però avremo anche le coppie del tipo (A, A) e (A, B) e (B, A), estrarre solo le coppie necessarie

$$R \leftarrow \sigma_{CF < CF1}(T_1)$$

Esercizio 2.

Restituire tutti gli impiegati con due o più persone a carico

$$PERSONA_A_CARICO1(IMP1, \dots) \leftarrow PERSONA_A_CARICO$$

$$R \leftarrow \pi_{IMP}(\sigma_{IMP = IMP1 \wedge NOME >> NOME1}(PERSONA_A_CARICO \times PERSONA_A_CARICO1))$$

Esercizio 3.

Restituire tutti gli impiegati con tre o piú persone a carico

$PERSONA_A_CARICO1(IMP1, \dots) \leftarrow PERSONA_A_CARICO$

$PERSONA_A_CARICO2(IMP2, \dots) \leftarrow PERSONA_A_CARICO$

$R \leftarrow \pi_{IMP}(\sigma_{IMP1 = IMP2 \wedge NOME1 <> NOME2 \wedge NOME2 <> NOME}(\sigma_{IMP = IMP1 \wedge NOME <> NOME1}(PERSONA_A_CARICO \times PERSONA_A_CARICO1)) \times PERSONA_A_CARICO2)$

Esercizio 4. (7-2-19)

Dato il seguente schema relazionale relativo a film e attori:

FILM(CodiceFilm, Titolo, Regista, Anno);
ATTORE(CodiceAttore, Cognome, Nome, Sesso, DataNascita, Nazionalità);
INTERPRETAZIONE(Film, Attore, Personaggio);

Ogni film è identificato univocamente da un codice e sia caratterizzato da un titolo, da un regista e dall'anno in cui è uscito. Per semplicità, si assuma che ogni film sia diretto da un unico regista e che un regista sia identificato univocamente dal suo Cognome. Non si escluda la possibilità che film diversi abbiano lo stesso titolo (è questo il caso, ad esempio, dei remake). Ogni attore sia identificato univocamente da un codice e sia caratterizzato da un nome, un cognome, un sesso, una data di nascita e una nazionalità. Più attori possono recitare in uno stesso film e un attore possa recitare in più film. Per semplicità, si assume che in un film un attore possa interpretare un solo personaggio.

Definire preliminarmente le chiavi primarie, le eventuali altre chiavi candidate e, se ve ne sono, le chiavi esterne delle relazioni date.

Chiavi **primarie**:

- CodiceFilm per FILM;
- CodiceAttore per ATTORE;
- Film + Attore per INTERPRETAZIONE;

Chiavi **esterne**:

- FILM è chiave esterna per INTERPRETAZIONE rispetto FILM;
- ATTORE è chiave esterna per INTERPRETAZIONE rispetto ATTORE;

Successivamente, formulare opportune interrogazioni in algebra relazionale che permettano di determinare (senza usare l'operatore di divisione e usando solo se necessario le funzioni aggregate):

Gli attori che hanno recitato in tutti i film diretti dal regista Antonioni;

Seleziono i film di Antonioni

$FilmAntonioni(Film) \leftarrow \pi_{CodiceFilm}(\sigma_{Regista = 'Antonioni'}(Film))$

Siamo di fronte ad una **condizione universale**, dovrò dunque passare attraverso una negazione e poi una differenza insiemistica: costruisco un insieme di coppie tale che per ogni attore accoppio quel attore con tutti i film di Antonioni.

$Requisiti(FILM, ATTORE) \leftarrow FilmAntonioni \times \pi_{CodiceAttore}(ATTORE)$

Ora ho la tabella Requisiti in cui ho legato ogni attore con i film di Antonioni. Costruisco una tabella StatoDiFatto, in cui riporto esattamente per ogni attore i film in cui ha recitato.

$$StatoDiFatto \leftarrow \pi_{FILM, ATTORE} (INTERPRETAZIONE)$$

Ora è necessario fare la differenza insiemistica per trovare gli attori che non soddisfano almeno un requisito

$$NOGOOD \leftarrow \pi_{Attore} (Requisiti - StatoDiFatto)$$

$$Candidati(ATTORE) \leftarrow \pi_{CodiceAttore} (ATTORE)$$

$$R \leftarrow Candidati - NOGOOD$$

Gli attori che hanno recitato in al più due (0, 1, 2) film diretti dal regista Allen;

Seleziono da FILM tutti e soli i film in cui il regista è Allen, selezionare nella tabella INTERPRETAZIONE tutte le righe in cui il Film è uno di Allen

$$FilmAllen \leftarrow \pi_{Film, Attore} (\sigma_{REGISTRA = 'ALLEN'} (FILM) \bowtie_{CodiceFilm = Film} INTERPRETAZIONE)$$

Ora la tabella FilmAllen contiene coppie film-attore in cui il film è un film di Allen. Ora cerchiamo tutti gli attori che hanno recitato in almeno tre film con Allen e li togliamo dall'insieme di tutti gli attori.

$$FilmAllen1(FILM1, ATTORE1) \leftarrow FilmAllen$$

$$FilmAllen2(FILM2, ATTORE2) \leftarrow FilmAllen$$

$$NOGOOD \leftarrow \pi_{ATTORE} (FilmAllen \bowtie_{Attore = Attore 1 and Film <> Film1} FilmAllen1) \bowtie_{Attore 2 = Attore and Film <> Film2 and Film1 <> Film2} FilmAllen2$$

$$Candidati (ATTORE) \leftarrow \pi_{CodiceAttore} (ATTORE)$$

$$R \leftarrow Candidati - NOGOOD$$

Se fosse stato esattamente 2 sarebbe stato almeno 2 - almeno 3

Le coppie di attori tali che esista un film in cui il primo ha recitato e il secondo no, e viceversa."

Assumiamo la completezza dell'informazione positiva registrata in INTERPRETAZIONE (se un attore ha recitato in un film lo trovo scritto in INTERPRETAZIONE). Costruiamo una tabella NON_INTERPRETAZIONE che dà l'informazione negativa. Devo dunque costruire un universo rispetto il quale fare poi la differenza insiemistica.

$$UNIVERSO(FILM, ATTORTE) \leftarrow \pi_{CodiceFilm} (FILM) \times \pi_{CodiceAttore} (ATTORE)$$

$$NON_INTERPRETAZIONE(NFILM, NATTORE) \leftarrow UNIVERSO - \pi_{Film, Attore} (INTERPRETAZIONE)$$

Rossi ha recitato e Verdi non ha recitato per il film FILM = NFILM:

$$Cond1 \leftarrow \pi_{Attore, NAttore} (INTERPRETAZIONE \bowtie_{FILM = NFILM} NON_INTERPRETAZIONE)$$

Verdi ha recitato e Rossi non ha recitato per il film FILM = NFILM:

$$Cond2(ATTORE, NATTORE) \leftarrow \pi_{NAttore, Attore}(NON_INTERPRETAZIONE \bowtie_{FILM = NFILM} INTERPRETAZIONE)$$

La condizione $Attore < Nattore$ consente di eliminare una delle due coppie:

$$R \leftarrow \sigma_{Attore < Nattore} Cond1 \cap Cond2$$

Esercizio 5. (8-9-17)

Sia dato il seguente schema relazionale relativo agli insegnamenti offerti da un corso di laurea a ciclo unico di durata quinquennale (questo è il caso, ad esempio, del corso di laurea in giurisprudenza) di una data università:

```
STUDENTI(Matricola, NomeStudente, CognomeStudente, AnnoImmatricolazione);
INSEGNAMENTI(CodiceInsegnamento, NomeInsegnamento, CodiceDocente, Area, AnnoCorso);
ESAMI(Studente, Insegnamento, Voto, Lode).
```

STUDENTI			
Matricola	NomeStudente	CognomeStudente	AnnoImmatricolazione
123456	Giovanni	Rossi	2017
234567	Maria	Bianchi	2018
345678	Marco	Verdi	2019

INSEGNAMENTI				
CodiceInsegnamento	NomeInsegnamento	CodiceDocente	Area	AnnoCorso
INF101	Basi di Dati	DOC001	Database	1
MAT202	Analisi Matematica II	DOC002	Matematica	2
EC0303	Economia Politica	DOC003	Economia	3

ESAMI			
Studente	Insegnamento	Voto	Lode
123456	INF101	25	false
123456	MAT202	28	false
234567	INF101	30	true
234567	MAT202	26	false
345678	EC0303	22	false

Si assuma che ogni studente sia identificato univocamente dalla matricola e sia caratterizzato dal nome, dal cognome e dall'anno di immatricolazione (ad esempio, l'anno di immatricolazione degli studenti che si iscrivono al primo anno del corso di laurea nell'anno accademico 2017/18 è il 2017). Si assuma che ogni insegnamento sia identificato univocamente da un codice e sia caratterizzato dal codice del docente che lo tiene, dall'area disciplinare in cui si colloca (ad esempio, l'insegnamento di Complementi di basi di dati si colloca nell'area Basi di dati) e dall'anno di corso in cui si tiene (l'attributo Anno assume il valore 1 per gli insegnamenti del primo anno, il valore 2 per gli insegnamenti del secondo anno e così via—assumiamo che ogni insegnamento sia associato ad uno ed un solo anno). Si assuma che vi possano essere più insegnamenti con lo stesso nome (ad esempio, due diversi insegnamenti di algoritmi e strutture dati distinti mediante il loro codice). Si assuma, inoltre, che uno stesso docente possa tenere più insegnamenti e che possano essere offerti più insegnamenti relativi alla stessa area disciplinare. Si assuma, infine, che la relazione esami registri i voti degli studenti relativi agli insegnamenti dei quali hanno superato l'esame (voti compresi tra 18 e 30). L'attributo Lode può assumere i valori vero e falso. Il valore vero può essere associato solo al voto 30. Si noti come la base di dati possa contenere studenti che non hanno registrato alcun esame ed insegnamenti il cui esame non è stato superato da alcuno studente.

Definire preliminarmente le chiavi primarie, le eventuali altre chiavi candidate e, se ve ne sono, le chiavi esterne delle relazioni date.

Chiavi **primarie**:

- Matricola
- CodiceInsegnamento
- Studente Insegnamento

Chiavi esterne:

- Studente è chiave esterna rispetto a Matricola
- Insegnamento è chiave esterna rispetto a CodiceInsegnamento

Successivamente, formulare opportune interrogazioni in algebra relazionale che permettano di determinare (senza usare l'operatore di divisione e usando solo se necessario le funzioni aggregate):

Gli studenti che hanno registrato solo esami relativi ad insegnamenti del 1 e 2 anno

La parola solo denota un universale (per ogni studente, dovrebbe vedere quali sono gli esami che ha superato e verificare che tutti siano esami del 1 o 2 anno). Conviene passare attraverso la negazione della condizione (studenti che hanno registrato un esame di un anno maggiore o uguale a 3 e toglierli).

$$Candidati \leftarrow \pi_{\text{Matricola}}(\text{STUDENTI})$$

Prendo gli insegnanti, seleziono quelli degli anni di corso 3, 4 ,5 ... e faccio un join con la tabella ESAMI. Poi filtro e mantengo solo la colonna relativa agli studenti.

$$\begin{aligned} NOGOOD(\text{Matricola}) &\leftarrow \pi_{\text{Studente}}(\text{ESAMI} \bowtie_{\text{Insegnamento} = \text{CodiceInsegnamento}} \sigma_{\text{AnnoCorso} \geq 3}(\text{INSEGNAMENTI})) \\ R &\leftarrow \text{Candidati} - NOGOOD \end{aligned}$$

Per ogni area disciplinare, l'insegnamento (gli insegnamenti se più di uno) col maggior numero di esami registrati;

$$AIS \leftarrow \pi_{\text{Area}, \text{Insegnamento}, \text{Studente}}(\text{ESAMI} \bowtie_{\text{Insegnamento} = \text{CodiceInsegnamento}} \text{INSEGNAMENTI})$$

$$AINS(\text{Area}, \text{Insegnamento}, \text{NumeroEsami}) \leftarrow \pi_{\text{Area}, \text{Insegnamento}} F_{\text{COUNT Studente}} AIS$$

Così ottengo la seguente tabella:

AINS		
Area	Insegnamento	NumeroEsami
...
...

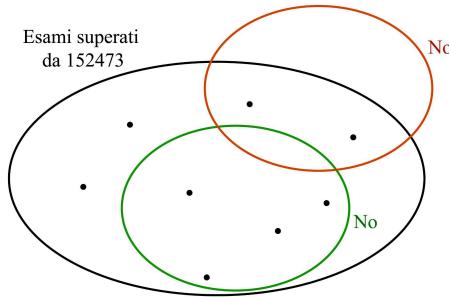
Creo una copia e, per trovare l'insegnamento con il maggior numero di esami registrati procedo come prima: passo al caso negativo e determino quali sono gli insegnamenti che non hanno registrato il numero massimo di esami in una certa area:

$$AINS1(\text{Area1}, \text{Insegnamento1}, \text{NumeroEsami1}) \leftarrow AINS$$

$$NOGOOD \leftarrow \pi_{\text{Area}, \text{Insegnamento}} (AINS \bowtie_{\text{NumeroEsami} < \text{NumeroEsami1} \wedge \text{Area} = \text{Area1}} AINS1)$$

$$R \leftarrow \pi_{\text{Area}, \text{Insegnamento}} (AINS) - NOGOOD$$

Gli studenti che hanno superato l'esame di un sottoinsieme proprio degli insegnamenti dei quali lo studente con matricola 152473 ha superato l'esame.



Troviamo innanzitutto gli esami superati da 152473

$$Esami152473 \leftarrow \pi_{Insegnamento} (\sigma_{Studente = 152473} (ESAMI))$$

Costruisco una tabella in cui associo ad ogni studente tutti gli esami svolti da 152473

$$Requisiti(Studente, Insegnamento) \leftarrow \pi_{Matricola} (Studenti) \times Esami152473$$

$$StatoDiFatto \leftarrow \pi_{Studente, Insegnamento} (ESAMI)$$

Devo verificare che ci sia almeno un esame che ha fatto 152473 che il candidato non ha fatto

$$Candidati \leftarrow \pi_{Studente} (Requisiti - StatoDiFatto)$$

Da questi candidati devo togliere gli studenti che hanno superato esami che 152473 non ha superato

$$NOGOOD \leftarrow \pi_{Studente} (StatoDiFatto - Requisiti)$$

E infine

$$R \leftarrow Candidati - NOGOOD$$

Calcolo relazionale

Il calcolo relazionale è una famiglia di linguaggi di interrogazione **dichiarativi** basati sul calcolo dei predicati del primo ordine.

Prenderemo in considerazione due linguaggi:

1. **Calcolo relazionale basato su domini** (DRC - Domain Relational Calculus);
2. **Calcolo relazionale basato su tuple** (TRC - Tuple Relational Calculus);

Entrambi i tipi di calcolo relazionale sono equivalenti dal punto di vista espressivo, il che significa che qualsiasi interrogazione esprimibile in uno può essere espressa nell'altro. Tuttavia, ciascuno può essere più intuitivo o naturale a seconda del contesto.

Le interrogazioni in linguaggio SQL

Un'interrogazione SQL può contenere fino a **6 clausole**, delle quali solo le prime 2 sono obbligatorie:

- `SELECT <lista di attributi> (target list)`
- `FROM <lista di tabelle>`
- `WHERE <condizione>`
- `GROUP BY <lista di attributi (di raggruppamento)>`
- `HAVING <condizione (di raggruppamento)>`
- `ORDER BY <lista di attributi>`

Le prime 3 clausole costituiscono il blocco fondamentale (detto mapping).

Il blocco fondamentale SELECT-FROM-WHERE

SELECT

Specifica gli attributi e/o le funzioni (in generale, le espressioni) il cui valore viene restituito dalla valutazione dell'interrogazione (schema della relazione risultato). Può essere specializzata in più modi (ad esempio, uso della parola chiave DISTINCT).

FROM

Specifica tutte le relazioni cui occorre accedere per recuperare l'informazione richiesta dall'interrogazione (non quelle coinvolte in eventuali interrogazioni nidificate).

WHERE

Specifica le condizioni (espressioni Booleane) per la selezione delle tuple delle relazioni indicate nella clausola FROM (condizioni di selezione e condizioni di join; in SQL-2 è possibile spostare le condizioni di join nella clausola FROM).

Esempi

Esempio 1. Trovare lo stipendio degli impiegati il cui cognome è Bianco.

```
SELECT STIPENDIO AS STIPENDIOBIANCO  
FROM IMPIEGATO  
WHERE COGNOME = 'BIANCO'
```

Esempio 2. Recuperare tutta l'informazione relativa agli impiegati il cui cognome è Bianco.

```
SELECT *  
FROM IMPIEGATO  
WHERE COGNOME = 'BIANCO'
```

Esempio 3. Trovare lo stipendio mensile degli impiegati il cui cognome è Bianco.

```
SELECT STIPENDIO / 12 AS STIPENDIOMENSILE  
FROM IMPIEGATO  
WHERE COGNOME = 'BIANCO'
```

Esempio 4. Per ogni impiegato, recuperare il nome del dipartimento per cui lavora.

```
SELECT CF, DNAME  
FROM IMPIEGATO, DIPARTIMENTO  
WHERE DIP = DNUMERO
```

Sottoblocco SELECT-FROM e uso di DISTINCT

Esempio 5. Recuperare lo stipendio di ogni impiegato.

```
SELECT CF, STIPENDIO  
FROM IMPIEGATO
```

Esempio 6. Determinare le diverse fasce di stipendio degli impiegati dell'azienda.

```
SELECT DISTINCT STIPENDIO  
FROM IMPIEGATO
```

Notazione puntata e nomi di attributo ambigui

Utile quando ha attributi con lo stesso nome in tabelle diverse.

Esempio 7. Per ogni impiegato, identificato da nome e cognome, recuperare i numeri dei progetti a cui lavora.

```
SELECT IMPIEGATO.CF, IMPIEGATO.NOME, IMPIEGATO.COGNOME, L.PROGETTO  
FROM IMPIEGATO, LAVORA_A AS L  
WHERE CF = IMP
```

Esempio 8. Recuperare nome e data di nascita delle persone a carico di ogni impiegato.

```
SELECT CF, P.NOME, P.DATA_NASCITA  
FROM IMPIEGATO, PERSONA_A_CARICO P  
WHERE CF = IMP
```

Alias e operazione di rinomina

Esempio 9. Recuperare nome e cognome dei supervisori degli impiegati che lavorano per il dipartimento 10.

```
SELECT S.NOME, S.COGNOME  
FROM IMPIEGATO AS I S  
WHERE I.DIP = 10 AND I.SUPERVISORE = S.CF
```

Esempio 10. Trovare i dipartimenti che hanno almeno una sede in comune.

```
SELECT DISTINCT S1.DNUMERO, S2.DNUMERO  
FROM SEDI_DIPARTIMENTO S1 S2  
WHERE S1.DSEDE = S2.DSEDE AND S1.DNUMERO < S2.DNUMERO
```

Nota. DISTINCT opera a livello di tuple (non di singola componente) e non potrebbe essere diversamente in quanto SQL è orientato alle tuple.

Espressioni Booleane nella clausola WHERE

Esempio 11. Recuperare nome e cognome degli impiegati di sesso maschile che guadagnano più di 40000 euro (si assuma che non vi siano omonimie).

```
SELECT NOME, COGNOME  
FROM IMPIEGATO  
WHERE SESSO = 'M' AND STIPENDIO > 40000
```

Esempio 12. Determinare gli impiegati di cognome BIANCO che lavorano per il dipartimento 2 o il dipartimento 3.

```
SELECT CF  
FROM IMPIEGATO  
WHERE COGNOME = 'BIANCO' AND (DIP = 2 OR DIP = 3)
```

Le operazioni insiemistiche

Le **operazioni insiemistiche** sono:

- UNION (unione);
- EXCEPT (differenza insiemistica);
- INTERSECT (intersezione);

I **duplicati** vengono rimossi, a meno che non venga richiesto in modo esplicito attraverso la **parola chiave ALL**.

Esempio 13. Selezionare i nomi di tutti gli impiegati e di tutte le persone a carico.

```
SELECT NOME  
FROM IMPIEGATO  
UNION  
SELECT NOME  
FROM PERSONA_A_CARICO
```

Se gli attributi hanno **nome diverso**, l'unione può comunque essere effettuata e il risultato normalmente usa i nomi del **primo operando** (il nome degli attributi della relazione risultato può dunque variare al variare dell'ordine degli operandi).

Esempio 14. Selezionare i nomi e i cognomi di tutti gli impiegati.

```
SELECT NOME  
FROM IMPIEGATO  
UNION  
SELECT COGNOME  
FROM IMPIEGATO
```

Esempio 15. Selezionare i nomi e i cognomi di tutti gli impiegati del Dipartimento numero 10, mantenendo gli eventuali duplicati.

```
SELECT NOME  
FROM IMPIEGATO  
WHERE DIP = 10  
UNION ALL  
SELECT COGNOME  
FROM IMPIEGATO  
WHERE DIP = 10
```

Esempio 16. Selezionare i cognomi degli impiegati che sono anche nomi (di qualche impiegato).

```
SELECT COGNOME  
FROM IMPIEGATO  
INTERSECT  
SELECT NOME  
FROM IMPIEGATO
```

Esempio 17. Selezionare i nomi degli impiegati che non sono anche cognomi (di qualche impiegato).

```
SELECT NOME  
FROM IMPIEGATO  
EXCEPT  
SELECT COGNOME  
FROM IMPIEGATO
```

Le interrogazioni nidificate

Nella clausola WHERE, SQL consente di confrontare, mediante i normali operatori di confronto, un **valore** (eventualmente ottenuto quale risultato di un'espressione valutata su una singola tupla) col **risultato** dell'esecuzione di una interrogazione (nidificata) SQL. Nel caso più semplice tale valore è il valore di uno specifico attributo.

Per confrontare un singolo valore con un insieme di valori (risultato della valutazione di un'interrogazione) si possono usare le parole chiave

- **ANY** (equivolentemente **SOME**): la tupla soddisfa la condizione se il confronto (con l'operatore specificato) tra il valore che l'attributo/i assume sulla tupla e **almeno uno** degli elementi restituiti dalla valutazione dell'interrogazione nidificata risulta VERO.
- **ALL**: la tupla soddisfa la condizione se il confronto (con l'operatore specificato) tra il valore che l'attributo/i assume sulla tupla e **ciascuno degli elementi** restituiti dalla valutazione dell'interrogazione nidificata risulta VERO.

Nota Bene: è richiesta la compatibilità di dominio tra l'attributo/i restituito dall'interrogazione e l'attributo/i con cui avviene il confronto.

Esempio 18. Selezionare gli impiegati che afferiscono ad un dipartimento con una sede a Pordenone.

```
SELECT *
FROM IMPIEGATO
WHERE DIP = ANY( SELECT DNUMERO
                   FROM SEDI_DIPARTIMENTO
                   WHERE DSEDE = 'PORDENONE' )
```

Esempio 19. Selezionare il codice fiscale degli impiegati che hanno lo stesso cognome di un impiegato che lavora per il dipartimento 10.

```
SELECT CF
FROM IMPIEGATO
WHERE COGNOME = ANY( SELECT COGNOME
                       FROM IMPIEGATO
                       WHERE DIP = 10)
```

Esempio 20. Selezionare tutti i dipartimenti nei quali non lavora alcun impiegato che guadagna più di 80000 euro.

```
SELECT DNUMERO
FROM DIPARTIMENTO
WHERE DNUMERO <> ALL ( SELECT DIP
                           FROM IMPIEGATO
                           WHERE STIPENDIO > 80000)
```

Esempio 21. Selezionare l'impiegato/i che percepisce lo stipendio massimo.

```
SELECT CF
FROM IMPIEGATO
WHERE STIPENDIO >= ALL ( SELECT STIPENDIO
                           FROM IMPIEGATO)
```

L'operatore IN

Si noti che se un'interrogazione nidificata restituisce un unico valore, è indifferente usare la parola chiave ANY o la parola chiave ALL (la parola chiave può anche essere omessa).

Per controllare l'appartenenza (rispettivamente, la non appartenenza) di un elemento/valore ad un insieme di elementi/valori restituiti da un'interrogazione nidificata, si possono utilizzare gli operatori IN (equivalente a = ANY) e NOT IN (equivalente a <> ALL).

Esempio 18'. Selezionare gli impiegati che afferiscono ad un dipartimento con una sede a Pordenone.

```
SELECT *
FROM IMPIEGATO
WHERE DIP IN ( SELECT DNUMERO
                  FROM SEDI_DIPARTIMENTO
                  WHERE DSEDE = 'PORDENONE' )
```

Interrogazioni nidificate correlate e non correlate

In tutte le interrogazioni nidificate sin qui viste, l'interrogazione nidificata può essere eseguita **una sola volta** (prima di analizzare le tuple dell'interrogazione esterna): il risultato della valutazione dell'interrogazione nidificata non dipende dalla specifica tupla presa in esame dall'interrogazione esterna (interrogazione nidificata non correlata all'interrogazione esterna). Tuttavia, l'interrogazione nidificata deve poter far riferimento alle tuple delle tabelle dichiarate nell'interrogazione esterna: il risultato della valutazione dell'interrogazione nidificata dipende dalla specifica tupla presa in esame dall'interrogazione esterna. Occorre un **meccanismo per il passaggio di binding**: attributi di tabelle dichiarate nell'interrogazione più esterna devono poter essere richiamati dall'interrogazione più interna.

L'ordine di valutazione è il seguente:

Per ogni tupla dell'interrogazione più esterna, prima viene valutata l'**interrogazione nidificata**, poi viene valutata la **condizione della clausola WHERE** dell'interrogazione più esterna (valutazione che coinvolge il risultato dell'interrogazione nidificata).

Tale processo può essere ripetuto un **numero arbitrario di volte**, pari al numero di interrogazioni nidificate l'una nell'altra in un'interrogazione (problemi di leggibilità delle interrogazioni).

Regole di visibilità (o scoping): gli attributi di una tabella sono utilizzabili solo nell'ambito dell'interrogazione in cui la tabella è dichiarata o di un'interrogazione nidificata (a qualsiasi livello) all'interno di essa.

```
SELECT *  
FROM IMPIEGATO  
WHERE DIP IN ( SELECT DNUMERO  
                  FROM SEDI_DIPARTIMENTO  
                  WHERE DSEDE = 'PORDENONE' )
```

Un attributo definito in IMPIEGATO può essere usato al **livello in cui è definito** (a.e. qui viene usato DIP a livello di IMPIEGATO), oppure possono essere utilizzati a **livello dell'interrogazione nidificata**.

Un attributo definito in SEDI_DIPARTIMENTO può essere ugualmente usato al livello in cui è definito e nelle sue sottointerrogazioni, dunque non sarà usabile nelle interrogazioni più esterne.

Uso degli alias: ci possono essere delle ambiguità quando sono presenti più attributi con lo stesso nome, uno che compare in una tabella dichiarata nell'interrogazione esterna, uno che compare in una tabella dichiarata nell'interrogazione nidificata. Quando c'è una situazione di questo tipo, posso disambiguare in 2 modi:

1. notazione puntata;
2. rinomina delle tabelle e notazione puntata;

Esempio di violazione delle regole di visibilità

```
SELECT CF  
FROM IMPIEGATO  
WHERE DIP IN ( SELECT DNUMERO  
                  FROM DIPARTIMENTO AS D1, SEDI_DIPARTIMENTO AS S1  
                  WHERE D1.DNUMERO = S1.DNUMERO AND D1.DNOME = 'RICERCA') OR  
DIP IN ( SELECT DNUMERO  
                  FROM SEDI_DIPARTIMENTO AS D2  
                  WHERE S1.DSEDE = D2.DSEDE)
```

Tale soluzione è scorretta perché la variabile di tupla S1 non è visibile nella seconda interrogazione nidificata.

Interrogazioni nidificate correlate e uso di alias

Esempio 23. Determinare nome e cognome di tutti gli impiegati che hanno una persona a carico del loro sesso con lo stesso nome.

```
SELECT NOME, COGNOME  
FROM IMPIEGATO I  
WHERE CF IN ( SELECT IMP  
                  FROM PERSONA_A_CARICO  
                  WHERE I.NOME = NOME AND I.SESSO = SESSO)
```

Esempio 24. Selezionare gli impiegati che percepiscono uno stipendio diverso da tutti gli altri impiegati del loro dipartimento.

```

SELECT CF
FROM IMPIEGATO AS I
WHERE STIPENDIO NOT IN (SELECT STIPENDIO
                        FROM IMPIEGATO
                        WHERE CF <> I.CF AND DIP = I.DIP)

```

La funzione Booleana EXISTS

SQL introduce una funzione Booleana (funzione EXISTS) che consente di verificare se il risultato di **un'interrogazione nidificata correlata è vuoto o meno**.

- Relazione risultato non vuota: EXISTS
- Relazione risultato vuota: NOT EXISTS

Dato che non siamo interessati alle specifiche tuple restituite dall'interrogazione nidificata, ma solo alla presenza/assenza di tuple, la clausola select dell'interrogazione nidificata assume di norma la forma: **SELECT ***. Sarebbe inutile scrivere altro e sarebbe indice di non aver capito.

Esempio 25. Selezionare gli impiegati che non hanno persone a carico.

```

SELECT CF
FROM IMPIEGATO
WHERE NOT EXISTS ( SELECT *
                     FROM PERSONA_A_CARICO
                     WHERE IMPIEGATO.CF = PERSONA_A_CARICO.IMP)

```

Selezioniamo tutti i campi degli impiegati (**SELECT * FROM IMPIEGATO**). La subquery nella clausola **WHERE NOT EXISTS** controlla poi se ci sono record nella tabella **PERSONA_A_CARICO** dove il codice fiscale dell'impiegato (**CF**) corrisponde al campo **IMP** (identificatore dell'impiegato nella tabella **PERSONA_A_CARICO**). Se non esistono tali record, l'impiegato viene incluso nel risultato finale.

Esempio 26. Restituire il nome e il cognome dei manager che hanno almeno una persona a carico.

```

SELECT NOME, COGNOME
FROM IMPIEGATO
WHERE EXISTS ( SELECT *
                  FROM DIPARTIMENTO
                  WHERE CF = MANAGER) AND
EXISTS ( SELECT *
                  FROM PERSONA_A_CARICO
                  WHERE CF = IMP)

```

INTERSECT ed EXCEPT via EXISTS

Dallo studio del calcolo relazionale su tuple con dichiarazioni di range sappiamo che gli **operatori insiemistici** di intersezione e differenza, a differenza dell'operatore di unione, **non sono strettamente necessari**. Vediamo come possono essere definiti in SQL mediante la funzione Booleana EXISTS.

Siano date due relazioni R(A, B) e S(C, D):

Intersezione	
<pre> SELECT A, B FROM R INTERSECT SELECT C, D FROM S </pre>	<pre> SELECT A, B FROM R WHERE EXISTS (SELECT * FROM S WHERE C = A AND D = B) </pre>

Differenza insiemistica	
<pre> SELECT A, B FROM R </pre>	<pre> SELECT A, B FROM R </pre>

<pre> EXCEPT SELECT C, D FROM S </pre>	<pre> WHERE NOT EXISTS (SELECT * FROM S WHERE C = A AND D = B) </pre>
--	--

L'operatore **CONTAINS** (rimosso)

L'implementazione originaria di SQL (System R) prevedeva un operatore CONTAINS che consentiva di stabilire se un insieme era o meno contenuto in un altro.

Esempio 27. Trovare gli impiegati che lavorano a tutti i progetti controllati dal dipartimento 10.

```

SELECT  CF
FROM    IMPIEGATO WHERE ( SELECT  PROGETTO
                           FROM   LAVORA_A
                           WHERE   CF = IMP)
                           CONTAINS (
                           (SELECT  PNUMERO
                           FROM   PROGETTO
                           WHERE   DNUM = 10)

```

CONTAINS è stato successivamente rimosso da SQL. Come esprimere l'operatore CONTAINS?

```

SELECT  CF
FROM    IMPIEGATO
WHERE   NOT EXISTS (   SELECT  *
                           FROM   PROGETTO
                           WHERE   DNUM = 10
                           AND
                           NOT EXISTS (   SELECT  *
                           FROM   LAVORA_A
                           WHERE   CF = IMP
                           AND
                           PNUMERO = PROGETTO))

```

La funzione Booleana UNIQUE

La funzione Booleana UNIQUE consente di verificare che nel risultato di un'interrogazione nidificata non siano presenti dei duplicati.

Esempio 28. Trovare gli impiegati che non hanno a carico due o più persone dello stesso sesso.

```

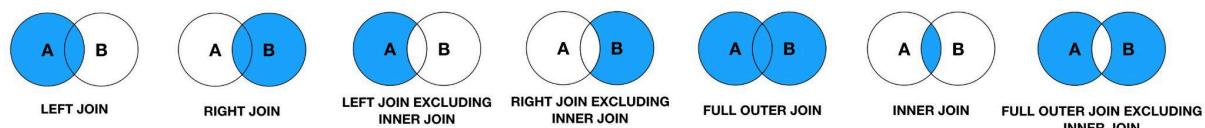
SELECT  CF
FROM    IMPIEGATO
WHERE   UNIQUE ( SELECT SESSO
                           FROM PERSONA_A_CARICO
                           WHERE CF = IMP)

```

Le operazioni di join

SQL-2 consente di specificare una tabella che si ottiene come risultato di un'operazione di join nella clausola FROM (separazione tra condizioni di selezione, nella clausola WHERE, e condizioni di join, nella clausola FROM). Tipi di join:

- INNER JOIN (o semplicemente JOIN)
- LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN (la parola OUTER si può omettere)
- NATURAL JOIN
- NATURAL LEFT/RIGHT/FULL OUTER JOIN



Inner join

Esempio 29. Recuperare nome, cognome e indirizzo degli impiegati che afferiscono al dipartimento Ricerca.

```
SELECT NOME, COGNOME, INDIRIZZO  
FROM (IMPIEGATO JOIN DIPARTIMENTO  
      ON DIP = DNUMERO)  
WHERE DNAME = 'RICERCA'
```

Esempio 30. Per ogni progetto con sede Tolmezzo, restituire il numero del progetto, il dipartimento che lo controlla e il cognome del manager di tale dipartimento.

```
SELECT PNUMERO, DNUMERO, COGNOME  
FROM ((PROGETTO JOIN DIPARTIMENTO ON DNUM = DNUMERO)  
      JOIN IMPIEGATO ON MANAGER = CF)  
WHERE PSEDE = 'TOLMEZZO'
```

Outer join

Esempio 31. Restituire nome e cognome di ogni impiegato e del relativo supervisore diretto (se esiste).

```
SELECT I.NOME AS NOMEIMP, I.COGNOME AS COGNOMEIMP, S.NOME AS NOMESUP, S.COGNOME AS COGNOMESUP  
FROM (IMPIEGATO AS I LEFT OUTER JOIN IMPIEGATO AS S ON I.SUPERVISORE = S.CF)
```

Esempio 32. Restituire tutti i dipartimenti con gli eventuali progetti da essi controllati.

```
SELECT PNUMERO, DNUMERO  
FROM (PROGETTO RIGHT OUTER JOIN DIPARTIMENTO ON DNUM = DNUMERO)
```

Funzioni aggregate in SQL

SQL fornisce alcune funzioni built-in quali COUNT, SUM, AVG, MAX e MIN (alcune funzioni per il calcolo della varianza, mediana, etc. sono state aggiunte in SQL-99).

Esempio 33. Determinare la somma totale degli stipendi pagati dall'azienda ai suoi impiegati, lo stipendio medio, lo stipendio massimo e lo stipendio minimo.

```
SELECT SUM(STIPENDIO), AVG(STIPENDIO), MAX(STIPENDIO), MIN(STIPENDIO)  
FROM IMPIEGATO
```

Nell'esempio precedente, tutte le funzioni aggregate prendono in considerazione tutti i valori diversi dal valore nullo dell'attributo STIPENDIO (lo stesso effetto si otterrebbe utilizzando la parola chiave **ALL**).

L'operatore COUNT consente di **contare il numero di righe di una tabella** (opzione *), il numero di valori distinti assunti dall'attributo/i cui viene applicata la funzione (parola chiave DISTINCT) e il numero di righe che assumono un valore diverso dal "valore" NULL sull'attributo/i cui viene applicata la funzione (parola chiave ALL).

Esempio 34. Determinare il numero di impiegati del dipartimento 3.

```
SELECT COUNT (*)  
FROM IMPIEGATO  
WHERE DIP = 3
```

Esempio 35. Determinare il numero di fasce diverse di stipendio degli impiegati dell'azienda.

```
SELECT COUNT(DISTINCT STIPENDIO)  
FROM IMPIEGATO
```

Interrogazioni inconsistenti (mismatch)

Le funzioni aggregate **non forniscono un meccanismo di selezione**. Esse vengono applicate ad un insieme (di valori) e restituiscono un singolo valore. Ne segue che la clausola SELECT non può contenere sia un attributo (o più), che genera un valore per ogni tupla selezionata, sia una funzione aggregata (o più), che restituisce un singolo valore per l'intero insieme di tuple. Un esempio di interrogazione inconsistente:

```
SELECT NOME, COGNOME, MAX(STIPENDIO)
FROM IMPIEGATO, SEDI_DIPARTIMENTO
WHERE DIP=DNUMERO AND DSEDE = 'TRIESTE'
```

Interrogazioni con raggruppamento

Funzioni aggregate e clausola GROUP BY, che consente di **partizionare le tuple di una relazione**.

Esempio 36. Per ogni dipartimento, determinare la somma degli stipendi degli afferenti.

```
SELECT DIP, SUM(STIPENDIO)
FROM IMPIEGATO
GROUP BY DIP
```

Prende la tabella impiegato e fa una partizione delle tuple di IMPIEGATO sulla base dei dipartimenti (ossia avrà un sottoinsieme di impiegati per il dipartimento 1, uno per il dipartimento 2, ...). Infine ritorna, per ognuna di queste classi, la somma degli stipendi.

In presenza della clausola GROUP BY, la clausola SELECT può contenere solo funzioni aggregate e (un sottoinsieme de) gli attributi di raggruppamento.

Interrogazioni sintatticamente scorrette

Esempio 37. Per ogni dipartimento, restituire il numero di afferenti e il manager

soluzione scorretta

```
SELECT DIP, COUNT(*), MANAGER
FROM IMPIEGATO, DIPARTIMENTO
WHERE DIP = DNUMERO
GROUP BY DIP
```

soluzione corretta

```
SELECT DIP, COUNT(*), MANAGER
FROM IMPIEGATO, DIPARTIMENTO
WHERE DIP = DNUMERO
GROUP BY DIP, MANAGER
```

Predicati sui gruppi

La clausola HAVING può essere utilizzata per **restringersi alle sole classi della partizione** (indotta dagli attributi della clausola GROUP BY) che **soddisfano una determinata condizione**.

Esempio 38. Selezionare tutti e soli i dipartimenti che spendono più di 1000000 di euro in stipendi per i loro afferenti (riportando tale spesa).

```
SELECT DIP, SUM(STIPENDIO) AS TOTALESTIPENDI
FROM IMPIEGATO
GROUP BY DIP
HAVING SUM(STIPENDIO) > 1000000
```

La clausola **HAVING** viene applicata ad ogni classe della partizione **DIP**. L'ordine dell'esecuzione della query è il seguente.

- Prendo la tabella **IMPIEGATO**;
- Partiziono rispetto a **DIP**;
- Per ogni classe della partizione faccio il test **SUM(STIPENDIO) > 1000000**, tagliando fuori i **DIP** che non soddisfano;
- Restituisco il risultato.

Condizioni WHERE vs. condizioni HAVING

Vengono valutate **prima** le condizioni specificate dalla clausola **WHERE**, per restringere l'insieme delle tuple; **poi**, una volta partizionato l'insieme di tuple così ottenuto, vengono valutate le condizioni specificate dalla clausola **HAVING** per eliminare alcune classi della partizione.

Esempio 39. Per ogni dipartimento con più di 5 afferenti, determinare il numero di afferenti che guadagnano più di 60000 euro.

soluzione scorretta

```
SELECT DIP, COUNT(*)  
FROM IMPIEGATO  
WHERE STIPENDIO > 60000  
GROUP BY DIP  
HAVING COUNT(*) > 5
```

soluzione corretta

```
SELECT DIP, COUNT(*)  
FROM IMPIEGATO  
WHERE STIPENDIO > 60000 AND  
DIP IN (SELECT DIP  
        FROM IMPIEGATO  
        GROUP BY DIP  
        HAVING COUNT(*) > 5)  
GROUP BY DIP
```

Relazioni come valori

Esempio 40. Determinare cognome e nome degli impiegati che hanno due o più persone a carico (usando le funzioni aggregate).

```
SELECT COGNOME, NOME  
FROM IMPIEGATO  
WHERE (SELECT COUNT(*)  
      FROM PERSONA_A_CARICO  
      WHERE CF = IMP) ≥ 2
```

Altro

Usare tuple di attributi anziché singoli attributi

Nei confronti col risultato di interrogazioni nidificate, si possono usare tuple di attributi/valori anziché singoli attributi/valori.

Esempio 41. Selezionare gli impiegati che dedicano ad un progetto lo stesso numero di ore che vi dedica l'impiegato MNTGVN89S14J324Q.

```
SELECT DISTINCT IMP  
FROM LAVORA_A  
WHERE (PROGETTO, ORE_SETTIMANA) IN  
      (SELECT PROGETTO, ORE_SETTIMANA  
       FROM LAVORA_A)
```

```
WHERE IMP = 'MNTGVN89S14J324Q')
```

Nella clausola WHERE, possono essere introdotti in modo **esplicito** insiemi di valori.

Esempio 42. Selezionare gli impiegati che lavorano per i progetti 1, 2 o 3.

```
SELECT DISTINCT IMP
FROM LAVORA_A
WHERE PROGETTO IN {1, 2, 3}
```

Operatore LIKE e pattern matching

Pattern matching su stringhe: l'operatore LIKE. Vengono utilizzati due caratteri riservati: %, che rimpiazza un numero arbitrario (0 o più) di caratteri, e _, che rimpiazza un singolo carattere.

Esempio 43. Selezionare gli impiegati che risiedono in una (parte di) città con Codice di Avviamento Postale 33210.

```
SELECT NOME, COGNOME
FROM IMPIEGATO
WHERE INDIRIZZO LIKE '%33210%'
```

Esempio 44. Selezionare gli impiegati nati negli anni sessanta.

```
SELECT NOME, COGNOME
FROM IMPIEGATO
WHERE DATA_NASCITA LIKE '_ _ 6 - - - - -'
```

Se % o _ devono essere utilizzati come elementi di una stringa, ogni loro occorrenza deve essere preceduta da un **carattere di escape**, che viene specificato alla fine della stringa utilizzando la parola chiave ESCAPE.

Esempi: la sequenza 'AB_CD%\EF'ESCAPE'\ rappresenta la stringa 'AB_CD%EF' in quanto il carattere \ è specificato quale carattere di escape.

Per dati di tipo stringa, si può utilizzare l'**operatore di concatenazione** // per concatenare due stringhe.

Nei confronti, si può utilizzare l'**operatore BETWEEN** che consente di esprimere alcune condizioni in modo più compatto.

Esempio 45. Selezionare gli impiegati del dipartimento 3 che guadagnano tra i 30000 e i 40000 euro (estremi compresi)

```
SELECT *
FROM IMPIEGATO
WHERE (STIPENDIO BETWEEN 30000 AND 40000) AND DIP=3
```

Operatore ORDER BY

Le tuple appartenenti al risultato di un'interrogazione possono essere **ordinate**, in modo crescente o decrescente, sulla base di uno o più dei loro attributi usando la clausola ORDER BY.

Esempio 46. Restituire l'elenco degli impiegati e dei progetti ai quali lavorano, ordinati sulla base del dipartimento di afferenza (in modo decrescente) e, all'interno di ogni dipartimento, in ordine alfabetico (in modo crescente) rispetto a cognome e nome.

```
SELECT DNOME, COGNOME, NOME, PNOME
FROM IMPIEGATO, DIPARTIMENTO, PROGETTO, LAVORA_A
WHERE DIP = DNUMERO AND CF=IMP AND PROGETTO = PNUMERO
ORDER BY DNOME DESC, COGNOME ASC, NOME ASC
```

L'ordine ascendente è il default (ASC può essere omessa nella clausola ORDER BY).

Gestione dei valori nulli

Per selezionare le tuple che assumono (risp., non assumono) il valore NULL su un certo attributo, SQL fornisce il predicate IS NULL (risp., IS NOT NULL), che restituisce il valore vero se e solo se l'attributo ha valore nullo (risp., non ha valore nullo).

Sintassi: ATTRIBUTO IS [NOT] NULL

Esempio 47. Selezionare tutti gli impiegati dei quali non è noto lo stipendio.

```
SELECT *
```

```
FROM      IMPIEGATO  
WHERE    STIPENDIO IS NULL
```

Osservazione. SQL-89 usa la logica a 2 valori; SQL-2 la logica a tre valori (true, false, unknown)

Esercizi sulle interrogazioni SQL

Esercizio 1. (1-7-19)

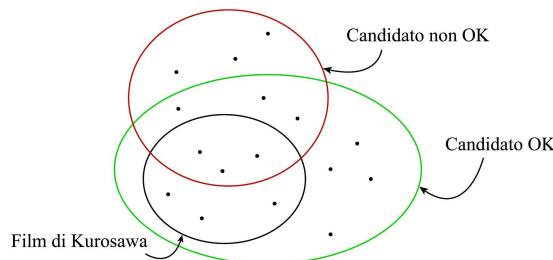
Sia dato il seguente schema relazionale relativo a film e attori:

```
FILM(CodiceFilm, Titolo, Regista, Anno);
ATTORE(CodiceAttore, Cognome, Nome, Sesso, DataNascita, Nazionalità);
INTERPRETAZIONE(Film, Attore, Ruolo);
```

1. Gli attori che hanno recitato solo in film di Kurosawa (**caso universale** → **not exists**);

```
SELECT CodiceAttore
FROM ATTORE A
WHERE NOT EXISTS (
    SELECT *
    FROM FILM F, INTERPRETAZIONE I
    WHERE F.Regista = 'Kurosawa' AND
        F.CodiceFilm = I.Film AND
        I.Attore = A.CodiceAttore);
```

2. Gli attori che hanno recitato in tutti i film di Kurosawa (**inclusione insiemistica** = non esiste un film di Kurosawa per il quale il candidato non ha recitato);



```
SELECT CodiceAttore
FROM ATTORE A0
WHERE NOT EXISTS (
    SELECT *
    FROM FILM F1
    WHERE F1.Regista = 'Kurosawa' AND
        NOT EXISTS (
            SELECT *
            FROM INTERPRETAZIONE I1
            WHERE I1.Attore = A0.CodiceAttore AND
                I1.Film = F1.CodiceFilm))
```

3. Gli attori che hanno recitato in tutti e soli i film di Kurosawa (**and delle due precedenti soluzioni**)

```
SELECT CodiceAttore
FROM ATTORE AS A0
WHERE NOT EXISTS (
    SELECT *
    FROM FILM F, INTERPRETAZIONE I
    WHERE F.Regista = 'Kurosawa' AND
        F.CodiceFilm = F.Film AND
        I.Attore = A.CodiceAttore
    AND
    NOT EXISTS (
        SELECT *
        FROM FILM F1
        WHERE F1.Regista = 'Kurosawa' AND
            NOT EXISTS (
                SELECT *
                FROM INTERPRETAZIONE I1
                WHERE I1.Attore = A0.CodiceAttore AND
                    I1.Film = F1.CodiceFilm)))
```

Esercizio 2 (31-1-2014)

Dato il database:

```
ISPETTORE(CF_Ispettore, Grado, AnnoNascita)
HA_CONTROLLATO(Ispettore, Azienda, Anno, Valutazione)
SI_TROVA_IN(Azienda, Città)
```

1. Le aziende che sono state controllate da un unico ispettore (eliminiamo dall'insieme di tutte le aziende candidate quelle che hanno avuto controlli da parte di almeno due ispettori);

```
SELECT DISTINCT Azienda
FROM HA_CONTROLLATO AS HC
WHERE NOT EXISTS (
    SELECT *
    FROM HA_CONTROLLATO
    WHERE Azienda = HC.Azienda AND
        Ispettore <> HC.Ispettore)
```

2. Gli ispettori che, nell'anno 2013, hanno controllato una o due aziende (bisogna escludere i casi 0 e i casi 3, 4, 5, ...);

```
SELECT Ispettore
FROM HA_CONTROLLATO AS HC
WHERE HC.Anno = '2013'
EXCEPT
SELECT HC1.Ispettore
FROM HA_CONTROLLATO AS HC1, HC2, HC3
WHERE HC1.Anno = '2013' AND HC2.Anno = '2013' AND HC3.Anno = '2013' AND
    HC1.Ispettore = HC2.Ispettore AND HC2.Ispettore = HC3.Ispettore AND
    HC1.Azienda <> HC2.Azienda AND HC2.Azienda <> HC3.Azienda AND HC1.Azienda <> HC3.Azienda
```

3. Gli ispettori che hanno controllato esattamente le stesse aziende (uguaglianza insiemistica $I1 \subseteq I2 \wedge I2 \subseteq I1$).

```
SELECT I1.CF_Ispettore, I2.CF_Ispettore
FROM ISPETTORE AS I1 I2
WHERE I1.CF_Ispettore < I2.CF_Ispettore AND
    NOT EXISTS(
        SELECT *
        FROM HA_CONTROLLATO HC1
        WHERE I1.CF_Ispettore = HC1.Ispettore AND
            NOT EXISTS(
                SELECT *
                FROM HA_CONTROLLATO HC2
                WHERE I2.CF_Ispettore = HC2.Ispettore AND
                    HC1.Azienda = HC2.Azienda))
    AND
    NOT EXISTS(
        SELECT *
        FROM HA_CONTROLLATO HC1
        WHERE I2.CF_Ispettore = HC1.Ispettore AND
            NOT EXISTS(
                SELECT *
                FROM HA_CONTROLLATO HC2
                WHERE I1.CF_Ispettore = HC2.Ispettore AND
                    HC1.Azienda = HC2.Azienda))
```

Esercizio 3 (8-9-2017)

```
STUDENTI(Matricola, NomeStudente, CognomeStudente, AnnoImmatricolazione);
INSEGNAMENTI(CodiceInsegnamento, NomeInsegnamento, CodiceDocente, Area, AnnoCorso);
ESAMI(Studente, Insegnamento, Voto, Lode);
```

(a) gli studenti che hanno registrato solo esami relativi ad insegnamenti del biennio iniziale del corso di laurea (caso universale → not exists);

-- non esiste un esame con i2.annodicorso < 2 che lo studente candidato ha superato

```
create view esamipassati as
select e1.studente, e1.insegnamento, i1.annodicorso
from esami e1, insegnamenti i1
where e1.insegnamento = i1.codice;

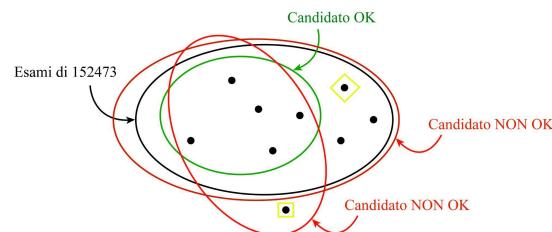
select distinct studente
from esamipassati e1
where not exists (
    select *
    from esamipassati e2
    where e1.studente = e2.studente
    and e2.annodicorso > 2
);
```

(b) per ogni area disciplinare, l'insegnamento (gli insegnamenti se più di uno) col maggior numero di esami registrati;

```
create view esamiregistrati as
select i.materia, count(*) as esamiregistrati
from insegnamenti i, esami e
where i.codice = e.insegnamento group by i.materia;

select *
from esamiregistrati e1
where not exists (
    select *
    from esamiregistrati e2
    where e1.materia <> e2.materia
    and e2.esamiregistrati > e1.esamiregistrati
);
```

(c) gli studenti che hanno superato l'esame di un sottoinsieme proprio degli insegnamenti dei quali lo studente con matricola 152473 ha superato l'esame.



Agisco così:

- ROMBO: cerco un esame che 152473 ha fatto e il candidato no;
- QUADRATO: cerco che non esista un esame che 152473 non ha fatto e il candidato si;

```
-- esiste un esame che 'Studente_3' ha superato e il candidato no
select *
from esami e1
where exists (
    select *
    from esami e2
    where e2.studente = 'Studente_3'
    and not exists (
        select *
        from esami e3
        where e3.insegnamento = e2.insegnamento
        and e1.studente = e3.studente
    )
)
-- non esiste un esame che 'Studente_3' non ha fatto e il candidato si
and not exists (
    select *
    from esami e2
    where e2.studente = e1.studente
    and not exists (
        select *
        from esami e3
        where e3.studente = 'Studente_3'
        and e3.insegnamento = e2.insegnamento
    )
);

```

Trovi altro qui: <https://github.com/6eero/Bachelor/tree/master/Database-and-Laboratory/Interrogazioni>

Il linguaggio SQL: le viste

Le viste in SQL, conosciute anche come "views", sono una sorta di "finestre virtuali" su uno o più tavole di un database. Esse sono utilizzate per diversi scopi, come **semplificare la complessità** delle operazioni sui dati, **personalizzare la visualizzazione dei dati per diversi utenti**, e proteggere i dati stessi **limitando l'accesso a specifiche parti** di una tabella.

Funzionalità delle viste:

1. **Astrazione:** Le viste possono nascondere la complessità delle query, esponendo all'utente solo i dati necessari.
2. **Sicurezza:** Limitando l'accesso a specifiche colonne o righe, le viste possono proteggere dati sensibili.
3. **Riutilizzabilità:** Le query che sono utilizzate frequentemente possono essere convertite in viste.
4. **Personalizzazione:** Diversi utenti possono avere viste personalizzate in base alle loro esigenze e permessi.

Le viste vengono definite in SQL nel modo seguente.

```
CREATE VIEW NomeVista [(ListaAttributi)] AS  
SELECT ...  
FROM ...  
WHERE ...
```

L'interrogazione deve restituire un insieme di attributi compatibile con gli attributi dichiarati nello schema della vista

Esempio di vista

Immaginiamo di avere una tabella Impiegati con i campi ID, Nome, Cognome e Reparto e Stipendio. Se vogliamo che un manager del reparto HR possa vedere solo i nomi e i reparti degli impiegati, ma non i loro stipendi, potremmo creare una vista come segue:

```
CREATE VIEW VistaImpiegatiHR AS  
SELECT Nome, Cognome, Reparto  
FROM Impiegati;
```

Questa vista, *VistaImpiegatiHR*, include solo le colonne Nome, Cognome e Reparto della tabella Impiegati. Un utente con accesso a questa vista **non sarà in grado di vedere lo stipendio** degli impiegati, il che può essere un requisito per la protezione dei dati sensibili.

Se si volesse trovare solo gli impiegati di nome Giorgio, continuando a non mostrare la colonna Stipendio, si potrebbe **riutilizzare la precedente view** in questo modo:

```
CREATE VIEW VistaImpiegatiHRSenzaReparto AS  
SELECT Nome, Cognome  
FROM VistaImpiegatiHR  
WHERE Nome = 'Giorgio'
```

Operazioni di modifica sulle viste

Su alcune viste si possono effettuare operazioni di modifica che vengono trasformate in operazioni corrispondenti sulle tavole di base da cui tali viste dipendono. Non sempre è possibile ricondurre in modo univoco le modifiche sulle viste a modifiche delle tavole di base.

Esempio.

```
CREATE VIEW TABELLA_LAVORA_A_RIVISTA AS
    SELECT NOME, COGNOME, PNOME, ORE_SETTIMANA
    FROM IMPIEGATO, PROGETTO, LAVORA A
    WHERE CF = IMP AND PNUMERO = PROGETTO

UPDATE TABELLA_LAVORA_A_RIVISTA
SET PNOME = 'PROGETTOB'
WHERE COGNOME = 'ROSSI' AND NOME = 'MARIO' AND PNOME = 'PROGETTOA'
```

Quando una vista può essere modificata?

Una vista è aggiornabile solo quando una sola tupla di ciascuna tabella di base corrisponde ad una tupla della vista (ciò, ad esempio, implica che la vista non può coinvolgere funzioni aggregate).

I sistemi commerciali sono più restrittivi: una vista è aggiornabile solo se è definita su una sola tabella (alcuni sistemi richiedono anche che gli attributi della vista contengano la chiave primaria della tabella e che a tutti gli attributi della vista sia associato il vincolo NOT NULL ma non un valore di default). La clausola CHECK OPTION può essere utilizzata solo nel contesto di tali viste: essa specifica che sono consentiti aggiornamenti solo delle tuple della vista e impone che dopo la loro esecuzione tali tuple continuino ad appartenere alla vista (una tupla puo' essere rimossa dalla vista se ad una degli attributi della vista viene assegnato un valore che rende falsa la condizione della clausola WHERE).

Esempio. Non è possibile operare una riduzione dello stipendio di un impiegato del Dipartimento 5 da un valore superiore a 10000 euro ad un valore minore o uguale a 10000 euro.

Opzione LOCAL o CASCADE

Nel caso una vista sia definita in termini di altre viste,

- **opzione LOCAL:** il controllo sul mantenimento dell'appartenenza delle tuple alla vista viene fatto solo all'ultimo livello (la modifica non deve provocare una violazione della condizione che definisce la vista piu' esterna);
- **opzione CASCADED** (opzione di default): il controllo sul mantenimento dell'appartenenza delle tuple alla vista viene fatto a tutti i livelli (si controlla che le tuple su cui si apportano le modifiche non scompaiono dalla vista per effetto della violazione di una qualsiasi delle condizioni delle viste coinvolte).

Esempio

```
CREATE VIEW IMPIEGATI5(CF5, NOME5, COGNOME5, STIP5) AS
    SELECT CF, NOME, COGNOME, STIPENDIO
    FROM IMPIEGATO
    WHERE DIP = 5 AND STIPENDIO > 10000

CREATE VIEW IMPIEGATI5POVERI AS
    SELECT *
    FROM IMPIEGATI5
    WHERE STIPENDIO < 25000
    WITH CHECK OPTION
```

Un assegnamento del valore 8000 euro ad una delle tuple della vista IMPIEGATI5POVERI è accettato con l'opzione LOCAL, ma è rifiutato con l'opzione CASCADING (default). Una modifica che assegna ad una tupla della vista il valore 40000 euro è rifiutato con entrambe le opzioni

Tecnologia di un Database Server (centralizzato)

Componenti di un database server

Componenti di un server per la gestione dei dati:

1. **ottimizzatore** (o gestore delle interrogazioni): traduce un'interrogazione da SQL ad una strategia di accesso dati, ossia, stabilisce quali strutture dati e quali algoritmi usare;
2. **gestore dei metodi di accesso ai dati**: trasforma le richieste contenute nei comandi di alto livello (SQL) in operazioni di lettura e scrittura di dati in memoria secondaria (è il modulo che si interfaccia con le primitive di R/W del os);
3. **buffer manager**: gestisce (più efficacemente possibile) il trasferimento effettivo delle pagine della base di dati da/da memoria secondaria a/da memoria principale, interagendo col gestore della memoria secondaria;
4. **controllore della concorrenza**: gestisce gli accessi concorrenti alla base di dati;
5. **controllo dell'affidabilità**: garantisce il buon funzionamento del sistema in presenza di malfunzionamenti e guasti.

Le transazioni

Una transazione è una **sequenza** di operazioni di **lettura e scrittura**. Ecco un esempio:

`t1: r1(x) r1(y) w1(x) w1(y)`

Letture/scritture eseguite dalla stessa transazione sono contraddistinte dal medesimo **indice** (indice che identifica la transazione). Si noti come in tale formalizzazione non compaiono le operazioni di manipolazione dei dati eseguite dalle transazioni. Dal punto di vista della teoria del controllo della concorrenza, infatti, ogni transazione è un **oggetto sintattico**, del quale si conoscono unicamente le azioni di ingresso/uscita (lettura/scrittura).

Le istruzioni di begin e end

Per definire una transazione ci sono due comandi per l'**incapsulamento** (per dire dove inizia e finisce la transazione):

- **begin** (o start) transaction (bot);
- **end** transaction (eot);

Il comando end transaction è di norma implicito. Se viene omesso il comando begin/start transaction, il sistema assume che ogni singola istruzione costituisca una transazione (modalità autocommit). E' ciò che accade usualmente negli ambienti interattivi.

Le istruzioni di commit e abort

Due istruzioni particolari:

- **commit** (commit work): rende definitivo quanto avvenuto nella transazione. Dev'essere posta alla fine della trans.;
- **abort** (rollback work): "disfa" quanto prodotto dalla transazione fino a quel punto;

L'istruzione **commit work** forza una conclusione positiva della transazione, con la registrazione **permanente** di tutti gli aggiornamenti nella base di dati. In transazioni più articolate, che utilizzano strutture di controllo, ci possono essere dei casi in cui, dopo aver effettuato parte delle operazioni, il verificarsi di alcune **condizioni indesiderate** impone l'annullamento degli aggiornamenti effettuati, col conseguente **ripristino della situazione precedente** l'(inizio dell')esecuzione della transazione. Tale obiettivo può essere raggiunto tramite l'esecuzione dell'istruzione **rollback work**.

Un esempio di transazione in SQL

Immaginiamo di voler specificare una transazione che trasferisce 10000 euro dal conto corrente numero 43719 al conto corrente numero 65286. In SQL, una tale transazione può essere formulata nel seguente modo:

```
start transaction;

update ContoCorrente set Saldo = Saldo + 10000 where NumConto = 65286;
update ContoCorrente set Saldo = Saldo - 10000 where NumConto = 43719;

commit work;
```

Non si vuole infatti eseguire tale operazione come due operazioni separate (sottrazione del denaro dal conto A e incremento del denaro sul conto B), ma bensì vuole essere un'operazione **unica e atomica**. Al termine della transazione, se tutto è andato a buon fine, avviene il commit.

Questo esempio però, esegue il commit a prescindere dall'esito delle operazioni e questo è sicuramente problematico. Se un'operazione va a buon fine e l'altra fallisce è impensabile pensare di poter fare un commit. Per questo motivo si utilizza una **struttura di controllo**, che si occupa di gestire i due casi:

```
start transaction;

update ContoCorrente set Saldo = Saldo + 10000 where NumConto = 65286;
update ContoCorrente set Saldo = Saldo - 10000 where NumConto = 43719;

select Saldo into A from ContoCorrente where NumConto = 43719;

if A ≥ 0
    then commit work;

rollback work;
```

Le anomalie

Le **transazioni concorrenti** possono causare diversi tipi di anomalie. Per illustrare le prime due anomalie, faremo riferimento alle seguenti due semplici transazioni (dalla medesima struttura):

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

Si vorrebbe che le due transazioni venissero eseguite una dopo l'altra ma questo raramente accade. Nascono dunque diversi problemi. Nota: $r_1(x)$ Significa lettura della transazione 1 su x

0. Conflitti scrittura-scrittura

Avviene quando si tenta di inserire tuple con lo stesso valore della chiave.

Transazione 1	Transazione 2
<pre>start transaction; insert into Account values ('Bud', 90); commit;</pre>	<pre>start transaction; insert into Account values ('Bud', 110); commit;</pre>

1. Perdita di aggiornamento

Avviene quando l'aggiornamento di t_2 viene perso perché sovrascritto da t_1 .

Risolta usando **qualsiasi** isolation level.

Transazione 1	Transazione 2
<pre>bot r1(x) x := x + 1 w1(x) abort</pre>	<pre>bot r2(x) x := x + 1 w2(x) commit</pre>

2. Lettura sporca

Avviene quando t_2 legge qualcosa scritto ma non committato da t_1 (che subito dopo abortisce).

Risolta usando isolation level **read committed**, **repeatable read** e **serializable**.

Transazione 1	Transazione 2
bot	
r1(x)	
x := x + 1	
w1(x)	
	bot
	r2(x)
	x := x + 1
	w2(x)
	commit
	abort

3. Aggiornamento fantasma

Risolta usando isolation level **repeatable read** e **serializable**.

Transazione 1	Transazione 2
bot	
s := 0	
r1(x); r1(y)	
s := s + x	
s := s + y	
	bot
	r2(z); z := z + 10000
	r2(y); y := y - 10000
	w2(y)
	w2(z)
	commit
r1(z)	
s := s + z	
commit	

4. Letture inconsistenti

Avviene quando due letture dello stesso dato danno valori diversi: tra le due letture da parte di t_1 , t_2 scrive e committa il valore della variabile letta da t_1 .

Risolta usando isolation level **repeatable read** e **serializable**.

Transazione 1	Transazione 2
	bot
r1(x)	
	r2(x)
	x := x + 1
	w2(x)
	commit
r1(x)	
commit	

5. Inserimento fantasma

Consideriamo una transazione che valuta un **valore aggregato** relativo all'insieme di tutti gli elementi che **soddisfano un dato predicato** di selezione (esempio: voto medio studenti del secondo anno di un certo corso di laurea). Se tale valore aggregato viene calcolato **due volte** e tra la prima e la seconda volta viene inserito/modificato/cancellato qualcosa da parte di un'altra transazione il risultato potrebbe essere diverso. Questa è una condizione che **non** si vuole avere ed è nota come inserimento fantasma.

Si noti come per evitare tale anomalia non sia sufficiente far riferimento solo ai dati già presenti nella base di dati (l'anomalia dipende da eventuali inserimenti intervenuti tra le due valutazioni).

Risolta solo usando isolation level **serializable**.

Proprietà acide delle transazioni

L'acronimo ACID sta per Atomicità, Coerenza, Isolamento e Durabilità (in inglese: Atomicity, Consistency, Isolation, Durability). Queste proprietà sono essenziali per garantire la correttezza e l'affidabilità delle transazioni in un database, specialmente in ambienti in cui molti utenti accedono e modificano contemporaneamente i dati.

Atomicità (Atomicity)

Questa proprietà assicura che ogni transazione sia trattata come un'**unità indivisibile** (atomica), che viene completata nella sua interezza oppure non viene eseguita affatto. Se una parte di una transazione fallisce (per esempio, a causa di un errore di sistema), allora l'intera transazione viene annullata (**rollback**). La corretta esecuzione dell'operazione di commit fissa l'istante atomico/divisibile in cui la transazione va a buon fine (prima di essa ogni guasto provoca un rollback, dopo può richiedere un redo).

Diverse ragioni possono portare all'**aborto di una transazione** (i meccanismi che realizzano l'abort sono sempre gli stessi):

- Esecuzione del **comando rollback work**: un suicidio autonomamente deciso dalla transazione;
- Il **sistema uccide la transazione** se capisce (attraverso timeout) che la transazione non è in grado di completarsi;
- Varie transazioni possono essere uccise a seguito di un **guasto di sistema**;

Di norma, l'esecuzione delle transazioni **va a buon fine** (terminano con un commit); in casi sporadici (ad esempio, malfunzionamenti o situazioni impreviste), terminano con un abort.

Consistenza (Consistency)

La consistenza garantisce che una transazione porti il database **da uno stato valido a un altro stato valido**, mantenendo **l'integrità dei dati**. In altre parole, l'esecuzione della transazione **non deve violare i vincoli di integrità** definiti sulla base di dati, ad esempio i vincoli di chiave unica o di integrità referenziale.

La verifica dei vincoli di integrità può essere di tipo

- **immediato**: effettuata durante l'esecuzione della transazione, rimuovendo gli effetti dell'esecuzione della transazione che provocano la violazione dei vincoli, senza forzare l'abort della transazione;
- **differito**: effettuata a conclusione della transazione dopo che è stata richiesta l'esecuzione del commit; in questo caso, se un vincolo è stato violato, l'istruzione commit work non va a buon fine e gli effetti della transazione vengono annullati;

Isolamento (Isolation)

L'isolamento si riferisce alla capacità di una transazione di operare in modo **indipendente dalle altre transazioni** e di non essere influenzata da modifiche concorrenti al database. Ci sono diversi livelli di isolamento nello standard SQL:

1. **read uncommitted**: non pone alcun vincolo sui lock;
2. **read committed**: lock condiviso in lettura (no due fasi). 2pl sulla scrittura;
3. **repeatable read**: 2PL Stretto (acquisizione lock, coiit, rilascio lock);
4. **serializable**: 2PL Stretto + lock di predicato.

Livello di isolamento	Perdita di aggiornamento	Letture sporche	Aggiornamento fantasma Lettura inconsistente	Inserimento fantasma
read uncommitted	non possibile	possibile	possibile	possibile
read committed	non possibile	non possibile	possibile	possibile
repeatable read	non possibile	non possibile	non possibile	possibile
serializable	non possibile	non possibile	non possibile	non possibile

Per utilizzare i livelli su PostgreSQL, la sintassi è la seguente:

```
start transaction;
set transaction isolation level {read uncommitted | read committed | repeatable read | serializable};
...
```

Se l'opzione `set transaction isolation level` viene omessa, il livello di default è `read committed`.

Standard SQL vs PostgreSQL

In PostgreSQL ci sono alcune differenze rispetto allo standard SQL:

Livello di isolamento	Perdita di aggiornamento	Letture sporche	Aggiornamento fantasma Lettura inconsistente	Inserimento fantasma
read uncommitted	<u>possibile</u>	possibile	possibile	possibile
read committed	<u>possibile</u>	non possibile	possibile	possibile
repeatable read	non possibile	non possibile	non possibile	<u>non possibile</u>
serializable	non possibile	non possibile	non possibile	non possibile

Persistenza (Durability)

La durabilità assicura che una volta che una transazione è stata completata, i cambiamenti che essa ha introdotto nel database siano permanenti e sopravvivano a eventuali guasti del sistema. Anche in caso di crash del sistema, i dati non verranno persi.

Gestione del buffer

Il buffer in un sistema di gestione di database (DBMS) è una **porzione di memoria centrale** preallocata e **condivisa fra le varie transazioni**. Questo componente è diventato sempre più importante, data la diminuzione dei costi delle memorie, portando all'allocazione di buffer di dimensioni sempre maggiori. In alcuni casi, l'intera base di dati può essere gestita in memoria centrale.

Componenti di un DBMS

I componenti principali di un DBMS coinvolti nella gestione delle interrogazioni e nell'accesso alla memoria secondaria includono

- il **gestore delle interrogazioni**: ottimizza le interrogazioni;
- il **gestore dei metodi di accesso**: traduce le richieste in operazioni di lettura e scrittura;
- Il **gestore del buffer**:
 - mantiene temporaneamente porzioni della base di dati in memoria centrale per l'efficienza del DBMS;
 - carica/scarica le pagine dalla memoria secondaria alla memoria principale/dalla memoria principale;
 - accede alle pagine presenti nel buffer;
 - invia al gestore della memoria secondaria le richieste di lettura e scrittura fisica;
 - Per il controllo della concorrenza, il gestore del buffer interagisce con lo scheduler (lock manager).
- il **gestore della memoria secondaria**;

Organizzazione del buffer

L'organizzazione del buffer avviene in **pagine**, che corrispondono a uno o più **blocchi** di memoria secondaria. Il gestore del buffer gestisce operazioni come il caricamento e lo scaricamento delle pagine tra memoria secondaria e memoria principale, garantendo l'efficienza del sistema.

La tempistica delle letture/scrittura sulla base di dati da parte del gestore del buffer non necessariamente coincide con quella delle corrispondenti richieste:

- in caso di **lettura**, se la pagina/blocco è già presente nel buffer, non è necessario effettuare la lettura fisica (i tempi di accesso alle pagine/blocchi di memoria secondaria e alle pagine/blocchi di memoria principale differiscono di vari ordini di grandezza);
- in caso di **scrittura**, il gestore del buffer può decidere di ritardare la scrittura fisica, qualora consentito dalle proprietà di affidabilità del sistema. In entrambi i casi, l'obiettivo è quello di ridurre i tempi di risposta alle richieste di un'applicazione.

In entrambi i casi, l'obiettivo è quello di ridurre i tempi di risposta alle richieste di un'applicazione.

Politiche di gestione del buffer

Le politiche di gestione del buffer si basano sul **principio di località dei dati** e cercano di anticipare le richieste future caricando in anticipo le pagine necessarie basandosi sul fatto che i dati acceduti di recente hanno **maggior probabilità** di essere nuovamente acceduti nel futuro prossimo.

In aggiunta, anche in tale ambito sembra valere una variante del noto **principio di Pareto** (l'80% degli effetti è prodotto dal 20% delle cause): il 20% dei dati è acceduto dall'80% delle applicazioni. La conseguenza di ciò è che la maggior parte delle volte, quando arriva una richiesta da una transazione, il dato di interesse è **già presente** nel buffer.

Direttorio e variabili di stato

Per la gestione del buffer, il buffer manager mantiene le seguenti informazioni/strutture dati:

- un目录 che descrive il contenuto corrente del buffer, indicando per ciascuna pagina caricata il file fisico e il numero di blocco corrispondenti;
- per ogni pagina del buffer, due variabili di stato (un contatore, che specifica quanti sono i programmi/transazioni che utilizzano la pagina, e un bit di stato, che specifica se la pagina è stata o meno modificata – le modifiche devono essere prima o poi riportate in memoria secondaria).

Osservazione: al crescere della dimensione del buffer cresce l'importanza di gestire in modo efficiente il directory.

Primitive per la gestione del buffer

Il buffer manager mette a disposizione delle transazioni un insieme di primitive per il caricamento/scaricamento delle pagine:

1. **fix**: viene utilizzata per richiedere l'accesso ad una pagina e restituisce al chiamante il riferimento alla pagina del buffer, in modo da consentire l'effettivo accesso ai dati. Come funziona?

- a. Si cerca la pagina fra quelle già presenti nel buffer (pagine caricate in precedenza). Se trovata, l'operazione termina immediatamente con successo. Per il principio di località, ciò avviene abbastanza spesso;
 - b. Se non è presente, si cerca nel buffer una pagina libera (valore del contatore uguale a 0). Se il bit di stato segnala che è stata modificata (e non ancora salvata), viene aggiornata in memoria secondaria (operazione asincrona di `flush`). Vengono, quindi, operate le necessarie conversioni di indirizzi per identificare la pagina da caricare nel buffer ed viene effettuata l'operazione di lettura;
 - c. Se non esistono pagine libere, due politiche alternative sono possibili:
 - i. **politica steal:** consente di sottrarre una pagina ad un'altra transazione. La pagina selezionata, detta vittima, viene scaricata in memoria di massa (tramite un'operazione asincrona di `flush`). Vengono, quindi, operate le necessarie conversioni di indirizzi e viene effettuata l'operazione di lettura.
 - ii. **politica no-steal:** non consente di sottrarre pagine alle transazioni attive. La transazione viene pertanto sospesa ed entra in una coda di transazioni gestita dal buffer manager. Quando si libera una pagina, il buffer manager procede come al punto (b).
 - d. In tutti i casi considerati, quando si effettua un accesso ad una pagina, viene incrementato il contatore relativo agli utilizzatori della pagina.
2. **setDirty:** indica al buffer manager che una pagina è stata modificata. L'effetto è la modifica del relativo bit di stato.
 3. **unfix:** indica al buffer manager che il chiamante ha terminato di utilizzare la pagina. Ha l'effetto di decrementare il contatore di utilizzo della pagina.
 4. **force:** trascrive in memoria secondaria, in modo sincrono, una pagina del buffer (e aggiorna il corrispondente bit di stato). Tale operazione è richiesta dal gestore dell'affidabilità quando risulta necessario garantire che alcuni dati non vengano persi.

La scrittura in memoria secondaria può avvenire in modo sincrono, all'interno della transazione che la richiede, attraverso la primitiva `force` (politica `force`) oppure in modo asincrono (indipendente dall'applicazione) sulla base delle decisioni del buffer manager che tengono conto della necessità di recuperare spazio e dei criteri di ottimizzazione (politica `no-force`).

Pre-fetching e pre-flushing

Esiste la possibilità di anticipare i tempi di caricamento e di scaricamento delle pagine.

Pre-fetching. Caricamento delle pagine anticipato rispetto alle richieste delle transazioni, in quei casi in cui sono note a priori le modalità di accesso alle pagine della base di dati da parte di una transazione.

Pre-flushing. Scaricamento anticipato delle pagine rispetto al momento in cui vengono scelte come vittime: scrittura anticipata di pagine rese libere dall'esecuzione di un'operazione di `unfix`, che sono state modificate nel corso del loro utilizzo (bit di stato con valore `dirty`). L'esecuzione del pre-flushing rende più efficienti le successive operazioni di `fix`.

Osservazione: una pagina utilizzata da molte applicazioni può restare a lungo nel buffer, subendo varie modifiche, e venire trascritta in memoria secondaria con una sola operazione di scrittura.

DBMS e file system

Il DBMS usa alcune (poche) funzionalità del file system (`create`, `delete`, `open` e `close`), creando, però, una propria **astrazione** dei file, che consentono di garantire **efficienza** (tramite l'uso del buffer e una gestione di basso livello delle strutture fisiche) e **transazionalità** (attraverso il gestore dell'affidabilità, che assicura le proprietà di atomicità e persistenza). La prospettiva futura è far migrare tali funzionalità del DBMS al di fuori di essi, ad esempio, incorporando all'interno del sistema operativo.

Esercizi su livelli di isolamento

Esercizio 1

Si elenchino i livelli di isolamento dello standard SQL, spiegando quali tecniche devono essere implementate.

Lo standard SQL definisce quattro livelli di isolamento per gestire la concorrenza nei database. Ogni livello specifica il grado di visibilità delle transazioni tra loro, limitando o permettendo determinati fenomeni di concorrenza:

1. **Read Uncommitted:** è il livello di isolamento più basso, dove le transazioni possono leggere dati non ancora confermati da altre transazioni. Questo può causare letture sporche. Le tecniche sono:
 - a. Lettura Diretta dei Dati: le letture non bloccano le scritture e viceversa.
 - b. Assenza di Locking Rigo: non usa lock rigidi per prevenire la lettura di dati non confermati.
2. **Read Committed:** le transazioni possono leggere solo dati che sono stati confermati da altre transazioni. Questo previene le letture sporche, ma non impedisce le letture non ripetibili o le phantom reads.
 - a. Read lock: mantiene read lock per tutta la durata della transazione per garantire letture ripetibili.
 - b. Release di Lock su Scrittura: rilascia lock su scrittura subito dopo la conferma della transazione.
3. **Repeatable Read:** garantisce che se una transazione legge un valore, successive letture dello stesso valore nella stessa transazione restituiscono sempre lo stesso valore, prevenendo le letture non ripetibili. Non impedisce però le phantom reads.
 - a. Read lock: mantiene read lock per tutta la durata della transazione per garantire letture ripetibili.
 - b. Write lock: mantiene write lock per tutta la durata della transazione.
4. **Serializable:** è il livello di isolamento più alto, garantendo che le transazioni siano serializzabili, ovvero che il risultato di una serie di transazioni concorrenti sia lo stesso che se fossero state eseguite in serie, una dopo l'altra. Impedisce letture sporche, letture non ripetibili e phantom reads.
 - a. Locking Completo: usa sia read lock che write lock e li mantiene per tutta la durata della transazione.
 - b. Timestamping: ogni transazione riceve un timestamp che regola l'ordine delle operazioni.
 - c. Verifica delle Conflittualità: Utilizzo di algoritmi di controllo delle conflittualità per assicurare che le transazioni concorrenti non violino l'isolamento serializzabile.

R	
X	Y
a	10
b	20
c	30

T ₁	T ₂
start transaction; update R set Y=80 where X='a'; abort;	start transaction; <u>val</u> ← select Y from R where X='a'; update R set Y= <u>val</u> +10 where X='a'; commit;

Per ognuno dei livelli di isolamento dello standard SQL elencati in precedenza, stabilire se lo schedule proposto è ammesso e, se questo è il caso, scrivere il contenuto della tabella R al termine dello schedule.

Questo schedule origina una violazione di tipo lettura sporca, per questo motivo, se si usa un livello di isolamento al di sotto di quello read committed (dunque solo read uncommitted), si avranno violazioni.

Esercizio 2

Si scriva il codice SQL per creare e popolare la tabella T sotto riportata, tenendo conto che l'attributo X è composto da esattamente un carattere ed è la chiave primaria, mentre l'attributo Y è un numero intero non nullo, strettamente positivo.

Table 1: T	
X	Y
a	27532
b	128
c	76

```
create table T (
    X char(1) primary key,
    Y integer not null check (Y > 0)
);
```

Si consideri il seguente schedule che coinvolge due transazioni T1 e T2, assumendo di disporre di un'opportuna tabella di supporto A.

T1	T2
<pre>start transaction; select Y into A from T where X='a'; insert into T values ('f', 300); select Y into A from T where X='a'; commit;</pre>	<pre>start transaction; update T set Y = 1000 where X='a'; ...</pre>

Per ognuno dei 4 livelli di isolamento previsti dallo standard SQL, stabilire se lo schedule proposto risulta ammissibile qualora, al posto dei puntini, venga inserita l'istruzione commit e qualora, invece, venga inserita l'istruzione rollback.

nel caso di **commit**, T1 salva in A '27532', successivamente T2 aggiorna T, ponendo al posto di '27532' il valore 1000, che viene effettivamente aggiornato in T dato che T2 viene committato. A questo punto T1 inserisce ('f', 300) e salva in A '100'. Questo rappresenta una **lettura inconsistente**: T1 legge una riga, T2 la modifica, e T1 la legge di nuovo ottenendo un valore diverso. Per risolvere la lettura inconsistente serve isolation level repeatable read o serializable.

nel caso di **rollback**, non c'e' alcuna violazione, dato che avremmo solo la transazione 1.

Esercizio 3

Si scriva il codice SQL per creare e popolare la relazione T, riportata in Table 1, tenendo conto che l'attributo X è composto da cinque caratteri ed è la chiave primaria, l'attributo Y è un numero intero non nullo, compreso tra 0 e 30, e l'attributo Z è un Booleano, che può essere settato a True se e solo se Y = 30.

Table 1: La relazione T		
X	Y	Z
aaaaa	12	False
bbbbbb	25	False
cccccc	30	True

```

1  create table T (
2    X char(5) primary key,
3    Y integer not null check (Y between 0 and 30),
4    Z boolean check ((Z = true) and (Y = 30))
5  );
6  insert into T(X, Y, Z) values
7    ('aaaaa', 12, false),
8    ('bbbbbb', 25, false),
9    ('cccccc', 30, true);

```

Per ciascuno dei 4 livelli di isolamento previsti dallo standard SQL, si stabilisca se lo schedule proposto risulta ammissibile (i) qualora, al posto dei puntini, venga inserita l'istruzione commit e (ii) qualora venga, invece, inserita l'istruzione rollback, giustificando la risposta.

T1	T2
<pre> start transaction; SELECT AVG(Y) FROM T WHERE Y > 18; UPDATE T SET Y = 26 WHERE X = 'aaaaaa'; SELECT AVG(Y) FROM T WHERE Y > 18; ... </pre>	<pre> start transaction; SELECT * FROM T WHERE X = 'aaaaaa'; INSERT INTO T (X, Y, Z) VALUES ('dddddd', 30, True); commit; </pre>

la transazione T1 valuta un **valore aggregato** AVG relativo all'insieme di tutti gli elementi che **soddisfano il dato predicato** di selezione “> 18”. Tale valore aggregato viene calcolato **due volte** e tra la prima e la seconda volta viene inserito il valore “30” (che soddisfa il predicato) da parte di un'altra transazione. Il risultato della seconda lettura sarà chiaramente diverso.

- Nel caso di **commit** siamo di fronte ad un inserimento fantasma, risolto solo dal livello serializable.
- Nel caso di **abort**,abbiamo che T2 sta leggendo un dato modificato e non ancora commitato da parte di T1, la quale subito dopo abortisce. Pertanto siamo di fronte ad una lettura sporca, risolvibile operando con isolation level read committed o superiori (repeatable read e serializable.)

Riassunto	Con commit (inserimento fantasma)	Con Rollback (lettura sporca)
read uncommitted	Non ammesso	Non ammesso
read committed	Non ammesso	Ammesso
repeatable read	Non ammesso	Ammesso
serializable	Ammesso	Ammesso

Controllo della concorrenza

In generale, un sistema di gestione di database (DBMS) deve essere in grado di supportare diverse applicazioni e gestire simultaneamente **molteplici richieste** provenienti da utenti **differenti**.

L'unità di misura del carico applicativo di un DBMS è comunemente espressa nel **numero di transazioni per secondo** (TPS). Tale valore può variare notevolmente, spaziando dalle decine o centinaia di TPS fino alle migliaia di TPS. Questa variazione dipende dal **tipo** di sistema utilizzato, che può andare da sistemi informativi bancari o finanziari a sistemi di gestione di carte di credito o sistemi di prenotazione delle grandi compagnie aeree.

Di conseguenza, l'**esecuzione** delle transazioni in modo **seriale** (cioè una dopo l'altra, in sequenza) è **impensabile**, poiché rallenterebbe drasticamente le operazioni e non sarebbe sostenibile per gestire il carico di lavoro richiesto.

È importante notare che la definizione di transazione può variare tra l'utente e il sistema stesso. Ad esempio, una singola transazione dal punto di vista del sistema potrebbe corrispondere a più transazioni percepite dall'utente.

Il **controllo della concorrenza** è eseguito dallo **scheduler**, che tiene traccia di tutte le operazioni eseguite sulla base di dati dalle transazioni e decide se **accettare** o **rifiutare** le operazioni che vengono via via richieste dalle transazioni.

Per il momento, assumiamo che l'esito (commit/abort) delle transazioni sia noto a priori. Ciò consente di rimuovere dalla schedule tutte le transazioni abortite:

$$\text{schedule} = \text{commit-proiezione}$$

Si noti che tale assunzione non consente di trattare alcune anomalie (lettura sporca).

La nozione di schedule

Uno schedule è una **sequenza di operazioni di ingresso/uscita** relative ad un dato insieme di transazioni concorrenti. Formalmente, uno schedule S1 è una sequenza del tipo:

$$S1 : r1(x) \ r2(z) \ w1(x) \ w2(z) \dots$$

dove $r1(x)$ rappresenta la lettura dell'oggetto x da parte della transazione $t1$ e $w2(z)$ rappresenta la scrittura dell'oggetto z da parte della transazione $t2$.

Le operazioni compaiono nello schedule nell'**ordine temporale di esecuzione** sulla base di dati.

La nozione di schedule seriale

Uno schedule S si dice **seriale** se per ogni transazione t , tutte le azioni di t compaiono in S in sequenza, senza essere inframmezzate da azioni di altre transazioni.

Esempio di schedule seriale S2 in cui le transazioni $t0$, $t1$, e $t2$ vengono eseguite in sequenza:

$$S2 : r0(x) \ r0(y) \ w0(x) \ r1(y) \ r1(x) \ w1(y) \ r2(x) \ r2(y) \ r2(z) \ w2(z)$$

La nozione di schedule serializzabile

Uno schedule Si si dice **serializzabile** (corretto) se produce lo stesso risultato prodotto da uno dei possibili schedule seriali Sj delle stesse transazioni.

Questione: cosa vuol dire produrre lo stesso risultato? Diverse risposte a tale questione hanno dato origine a diversi criteri di serializzabilità.

Equivalenza di vista (VSR)

L'equivalenza di vista è caratterizzata in termini della **relazione legge** e dell'insieme delle **scritture finali**.

La **relazione legge** lega coppie di operazioni di lettura e scrittura: un'operazione di lettura $r_i(x)$ legge da un'operazione di scrittura $w_j(x)$

$\text{legge}(r_i(x), w_j(x))$

se e solo se il valore di x letto dalla transazione i è il valore scritto (prodotto) dalla transazione j., ossia:

1. $w_j(x)$ precede $r_i(x)$
2. non vi è alcun $w_k(x)$ tra $w_j(x)$ e $r_i(x)$.

Un'operazione di scrittura $w_i(x)$ viene detta una **scrittura finale** se è l'ultima scrittura dell'oggetto x che appare nello schedule.

Due schedule S_i e S_j vengono detti **equivalenti rispetto alle viste** ($S_i \approx_V S_j$) se possiedono la stessa relazione legge e le stesse scritture finali.

Uno schedule viene detto **serializzabile rispetto alle viste** se è equivalente rispetto alle viste ad un generico schedule seriale. Denotiamo con **VSR** l'insieme degli schedule serializzabili rispetto alle viste.

Esempio

$S_3 : w_0(x) r_2(x) r_1(x) w_2(x) w_2(z) \text{ legge}(r_2(x), w_0(x)) \text{ e legge}(r_1(x), w_0(x))$ scritture finali: $w_2(x)$ e $w_2(z)$
 $S_4 : w_0(x) r_1(x) r_2(x) w_2(x) w_2(z) \text{ legge}(r_1(x), w_0(x)) \text{ e legge}(r_2(x), w_0(x))$ scritture finali: $w_2(x)$ e $w_2(z)$
 $S_5 : w_0(x) r_2(x) w_2(x) r_1(x) w_2(z) \text{ legge}(r_2(x), w_0(x)) \text{ e legge}(r_1(x), w_2(x))$ scritture finali: $w_2(z)$ e $w_2(x)$
 $S_6 : w_0(x) r_2(x) w_2(x) w_2(z) r_1(x) \text{ legge}(r_2(x), w_0(x)) \text{ e legge}(r_1(x), w_2(x))$ scritture finali: $w_2(x)$ e $w_2(z)$

Lo schedule S_3 è equivalente rispetto alle viste allo schedule seriale S_4 (ed è, quindi, serializzabile rispetto alle viste). S_5 non è, invece, equivalente rispetto alle viste allo schedule seriale S_4 , ma è equivalente rispetto alle viste allo schedule seriale S_6 (ed è, quindi, serializzabile rispetto alle viste).

Limiti della tecnica VSR

Il problema di determinare se due schedule dati sono equivalenti rispetto alle viste ha **complessità lineare**. Il problema di determinare se un dato schedule è equivalente ad uno schedule seriale (**qualsiasi**) è, invece, **NP-hard** (è necessario confrontare lo schedule con tutti gli schedule seriali).

Soluzione: definire una condizione di equivalenza più ristretta, che non copra tutti i casi di equivalenza tra schedule coperti dall'equivalenza di vista, ma che sia utilizzabile nella pratica (complessità inferiore).

Equivalenza rispetto ai conflitti (CSR)

L'equivalenza rispetto ai conflitti si basa sulla nozione di azioni in **conflitto**. Un'azione a_i si dice in **conflitto** con un'azione a_j (coppia ordinata di azioni in conflitto (a_i, a_j)) se

1. sono operazioni che appartengono a transazioni diverse: $i \neq j$;
2. a_i e a_j sono operazioni che operano sullo stesso oggetto;
3. almeno una delle due azioni è una scrittura.

Si distinguono conflitti del tipo **lettura-scrittura** (rw e wr) e conflitti del tipo **scrittura-scrittura** (ww).

Due schedule S_i e S_j vengono detti **equivalenti rispetto ai conflitti** ($S_i \approx_C S_j$) se i due schedule presentano le medesime operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi gli schedule.

Uno schedule viene detto serializzabile rispetto ai conflitti se è equivalente rispetto ai conflitti ad un generico schedule seriale. Denotiamo con **CSR** l'insieme degli schedule serializzabili rispetto ai conflitti.

Il grafo dei conflitti

Per stabilire se uno schedule è serializzabile rispetto ai conflitti è possibile utilizzare il **grafo dei conflitti**. I nodi del grafo sono le transazioni e vi è un arco orientato da un nodo t_i ad un nodo t_j se (e solo se) esiste almeno una coppia di azioni in conflitto a_i e a_j , con a_i che precede a_j .

Proposizione: uno schedule è in CSR se e solo se il suo **grafo dei conflitti è aciclico**. L'analisi di ciclicità di un grafo ha una complessità lineare nella dimensione del grafo.

Dimostrazione appartenenza a CSR implica acicità

Per definizione, se uno schedule S appartiene a CSR, S è equivalente rispetto ai conflitti ad uno schedule seriale S_0 . Sia t_1, t_2, \dots, t_n la sequenza delle transazioni in S_0 . Dato che tutte le coppie in conflitto compaiono nello stesso ordine in S e S_0 , nel grafo dei conflitti di S vi sono soltanto archi (t_i, t_j) , con $i < j$, e, quindi, il grafo risulta privo di cicli (un ciclo richiede la presenza di almeno un arco (t_i, t_j) , con $i \geq j$).

Dimostrazione acicità implica appartenenza a CSR

Se il grafo di S è aciclico, allora esiste un ordinamento topologico dei nodi (una numerazione dei nodi tale che il grafo contiene solo archi (i, j) con $i < j$). Lo schedule seriale le cui transazioni sono ordinate secondo l'ordinamento topologico è equivalente a S , perché per ogni conflitto (i, j) , si ha che $i < j$.

<https://fereidani.com/tools/scheduleparser>

Limiti della tecnica CSR

- Seppur più efficiente di VSR, è ancora **troppo onerosa**,
- Il grafo dei conflitti si modifica **dinamicamente** rendendo difficoltose le decisioni dello scheduler,
- Impraticabile nel caso di **basi di dati distribuite**.

Legame tra VSR e CSR

E' possibile dimostrare che la classe degli schedule CSR è **strettamente contenuta** nella classe degli schedule VSR (la condizione di serializzabilità rispetto ai conflitti è condizione sufficiente, ma non necessaria, per la serializzabilità rispetto alle viste):

- $\text{Se } S \in \text{CSR} \rightarrow S \in \text{VSR}$ tutti gli schedule in CSR appartengono anche a VSR;
- $\exists S \text{ tale che } S \in \text{CSR} \wedge S \notin \text{VSR}$ esistono degli schedule appartenenti a VSR che non appartengono a CSR.

Locking a due fasi (2PL)

Si utilizza la tecnica del **locking a due fasi**. Questa idea consiste nel proteggere le operazioni di lettura e scrittura tramite l'esecuzione di opportune **primitive** (read lock, write lock e unlock).

Lo scheduler (**lock manager**) riceve una sequenza di richieste di esecuzione di tali primitive da parte delle transazioni e ne determina la correttezza mediante l'ispezione di un'opportuna struttura dati.

Vincoli che caratterizzano le transazioni ben formate

- ogni operazione di **lettura** deve essere **preceduta** da un **read lock** e **seguita** da un **unlock** (**lock condiviso**);
- ogni operazione di **scrittura** deve essere **preceduta** da un **write lock** e **seguita** da un **unlock** (**lock esclusivo**).

Le transazioni sono in genere automaticamente ben formate (lock / unlock inseriti in modo trasparente all'applicazione).

Il funzionamento

Lo scheduler riceve **richieste di lock** dalle transazioni e **concede/rifiuta** di concedere i lock sulla base dei lock precedentemente concessi ad altre transazioni.

Quando viene concesso il lock su una certa risorsa ad una data transazione, si dice che **la risorsa è acquisita dalla transazione**. Nel momento dell'unlock, la risorsa viene **rilasciata**. Quando una richiesta di lock viene rifiutata, la transazione richiedente viene messa in **stato di attesa**. Tale attesa può terminare quando la risorsa torna disponibile.

Per tener traccia dei lock già concessi e gestire le nuove richieste, lo scheduler utilizza opportune **tabelle dei conflitti** (una per ogni risorsa).

Ogni richiesta di lock è caratterizzata da tre parametri:

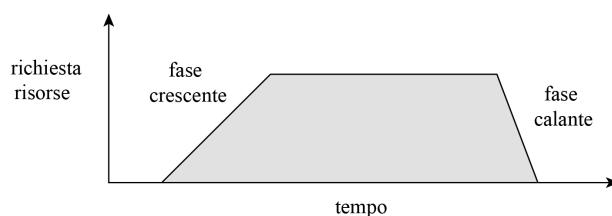
1. il **tipo** (read o write);
2. l'identificativo della **transazione** che effettua la richiesta;
3. la **risorsa** cui la richiesta si riferisce.



Nell'esecuzione di una transazione nel 2PL, si può distinguere in **due fasi**:

- nella prima si eseguono i lock alle risorse a cui si deve accedere (**fase crescente**);
- nella seconda si rilasciano progressivamente i lock (**fase calante**).

In figura viene fornita una rappresentazione grafica del comportamento richiesto dal protocollo 2PL, dove l'ascissa rappresenta il tempo e l'ordinata il numero di risorse possedute da una transazione durante la sua esecuzione.



Una volta che una transazione esegue un unlock, essa **non può più acquistare risorse**, ma può solo rilasciarle.

Le tabelle dei conflitti

La politica del lock manager è descritta dalla **tabella dei conflitti** sotto riportata, dove le righe identificano le richieste, le colonne lo stato della risorsa, il primo valore nella cella l'esito della richiesta e il secondo valore lo stato della risorsa dopo l'eventuale esecuzione della primitiva.

Richiesta	Stato risorsa		
	libero	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 / dipende	OK / libero

Se arriva una richiesta di **read lock** su una certa risorsa ci sono tre casi:

1. **la risorsa è libera**: viene concessa e passa nello stato di read locked;
2. **la risorsa è bloccata nello stato di read locked**: la richiesta può essere accolta e lo stato resta read locked;
3. **la risorsa è bloccata nello stato di write locked**: la richiesta viene sospesa e lo stato resta write locked;

Se arriva una richiesta di **write lock** su una certa risorsa ci sono due casi:

1. **la risorsa è libera**: viene concessa e passa nello stato di write locked;
2. **la risorsa è bloccata nello stato di read/write locked**: la richiesta viene sospesa e lo stato resta read locked;

Un **unlock** rilascia una risorsa solo se non ci sono altre transazioni che la stanno utilizzando in lettura; in caso contrario, la risorsa rimane bloccata in lettura. Per tener traccia del numero di transazioni che hanno bloccato in lettura una risorsa, è necessario utilizzare un contatore che viene opportunamente incrementato / decrementato.

Legame tra 2PL e CSR

Sia 2PL la classe contenente gli schedule che soddisfano le condizioni imposte dal protocollo 2PL.

- $Se S \in 2PL \rightarrow S \in CSR$ La classe 2PL è contenuta nella classe CSR;
- $\exists S \text{ tale che } S \in CSR \wedge S \notin 2PL$ Esistono schedule in CSR, ma non in 2PL.

Dimostrazione punto 1.

Assumiamo, per assurdo, che esista uno schedule S tale che $S \in 2PL \wedge S \notin CSR$.

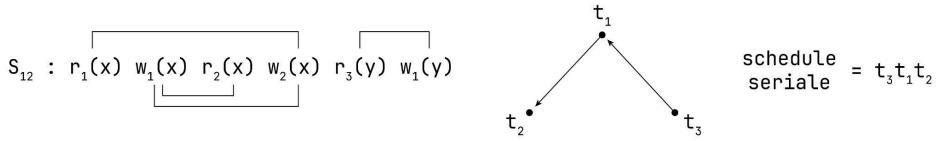
Da $S \notin 2PL$ segue che il grafo dei conflitti per S contiene un ciclo $t_1, t_2, \dots, t_n, t_1$. Se esiste un arco (conflitto) tra t_1 e t_2 , significa che esiste una risorsa x su cui le due transazioni operano in modo conflittuale. Affinché t_2 possa procedere, è necessario che t_1 rilasci il lock (unlock, inizio fase calante) su x e t_2 lo acquisisca. Analogamente, se esiste un arco (conflitto) tra t_n e t_1 , significa che esiste una risorsa y su cui le due transazioni operano in modo conflittuale. Affinché t_1 possa procedere, è necessario che t_n rilasci il lock su x e t_1 lo acquisisca. La transazione t_1 non rispetta, quindi, il protocollo 2PL.

Dimostrazione punto 2.

Prendiamo come esempio lo schedule:

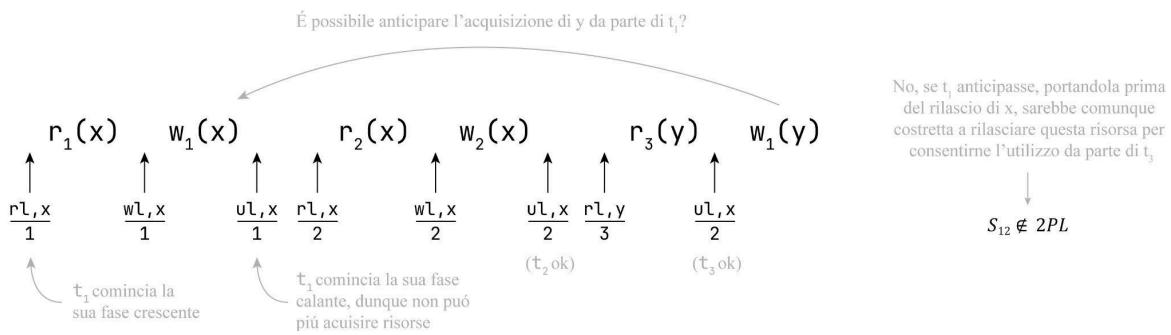
$$S_{12} : r_1(x) \ w_1(x) \ r_2(x) \ w_2(x) \ r_3(y) \ w_1(y)$$

Mostriamo che $S_{12} \in CSR$:



Dunque, lo schedule è serializzabile rispetto ai **conflitti** (è equivalente allo schedule seriale t_3, t_1, t_2).

Mostriamo che $S_{12} \notin 2PL$:



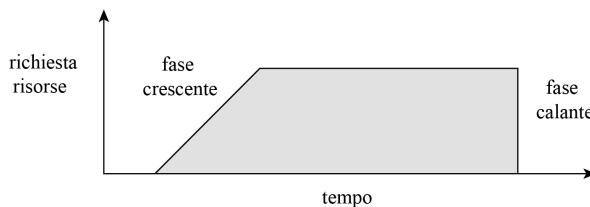
La transazione t_1 deve cedere un lock esclusivo sulla risorsa x per consentire alla transazione t_2 di accedervi, prima in lettura e poi in scrittura, e successivamente richiedere un lock esclusivo sulla risorsa y . Si noti che t_1 non può anticipare il lock su y in modo da effettuarlo prima del rilascio di x in quanto dovrebbe comunque rilasciare la risorsa y (per poi riacquisirla) per consentirne l'utilizzo da parte di t_3 .

Locking a due fasi stretto (2PL stretto)

Questa versione di 2PL viene utilizzata nei sistemi commerciali.

È un insieme di scheduler che permette di risolvere il problema della **lettura sporca** rimuovendo l'assunzione di **commit-proiezione**. Per garantire l'assenza di tale anomalia, è necessario imporre una **restrizione al protocollo 2PL**, che porta al cosiddetto 2PL stretto (locking a due fasi stretto):

I **lock** di una transazione possono essere **rilasciati solo dopo** aver correttamente effettuato le **operazioni di commit**. Questo implica che i valori modificati da una transazione vengono resi disponibili alle altre transazioni solo nel momento in cui sono resi definitivi dall'esecuzione del commit. Se la transazione termina con un **abort**, siamo certi che le altre transazioni non hanno letto i valori modificati dalla transazione che ha abortito.



Una volta che una transazione esegue un unlock, essa **non può più acquistare risorse** e deve **rilasciare tutte le sue risorse in un unico punto**.

Inserimento fantasma: lock di predicato

Per evitare gli inserimenti fantasma, è necessario che i lock possano essere definiti anche con riferimento a condizioni (o prediciati) di selezione, impedendo non solo l'accesso ai dati coinvolti, ma anche la scrittura di nuovi dati che soddisfano il predicato.

I lock di predicato sono realizzati nei sistemi relazionali con l'ausilio degli indici o, nel caso non siano disponibili, bloccando intere relazioni. Ciò significa che il sistema applica un lock sulla selezione dei dati che soddisfano il predicato, impedendo l'accesso a tali dati per la durata della transazione che detiene il lock. Se gli indici non sono disponibili per supportare il locking di predicato, il sistema può essere configurato per bloccare l'intera relazione coinvolta nel predicato, garantendo così l'integrità dei dati e prevenendo gli inserimenti fantasma.

Controllo basato sui timestamp (TS)

Ad ogni transazione si associa un **timestamp** che rappresenta il momento di inizio transazione. Successivamente si accetta uno schedule solo se riflette l'**ordinamento seriale** delle transazioni definito dai valori dei loro timestamp.

Per formalizzare questa tecnica, si inseriscono 2 parametri per ogni risorsa x:

- **WTM(x)**: il timestamp della transazione che ha eseguito l'ultima scrittura di x;
- **RTM(x)**: il timestamp più grande fra i timestamp delle transazioni che hanno letto x.

Allo scheduler arrivano richieste di accesso agli oggetti della forma $r_t(x)$ (la transazione con timestamp t chiede di leggere x) e $w_t(x)$ (la transazione con timestamp t chiede di scrivere x). Lo scheduler concede o meno di eseguire l'operazione secondo la seguente politica:

- $r_t(x)$: se $t < WTM(x)$ la transazione viene **uccisa**, altrimenti la richiesta viene accettata e $RTM(x)$ viene posto pari al massimo fra il valore corrente e t;
- $w_t(x)$: se $t < WTM(x)$ o $t < RTM(x)$ la transazione viene **uccisa**, altrimenti la richiesta viene accettata e $WTM(x)$ viene posto pari a t.

Detto in parole semplici, **una transazione non può leggere o scrivere un dato scritto da una transazione con timestamp superiore e non può scrivere un dato che è stato già letto da una transazione con timestamp superiore**.

Esempio

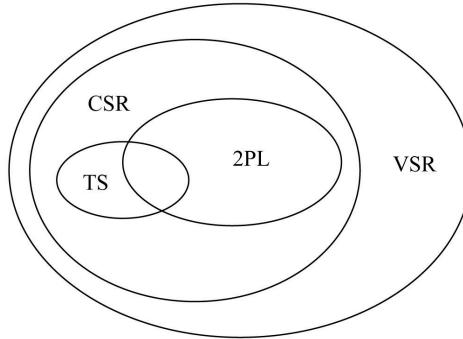
Si assuma che $RTM(x) = 7$ e $WTM(x) = 5$. Vediamo le risposte dello scheduler a fronte di una sequenza di richieste.

Richieste	Risposte	Nuovi valori
$r_6(x)$	ok	
$r_7(x)$	ok	
$r_9(x)$	ok	$RTM(x) = 9$
$w_8(x)$	no	t_8 uccisa
$w_{11}(x)$	ok	$WTM(x) = 11$
$r_{10}(x)$	no	t_{10} uccisa

Limiti del metodo TS

Il metodo TS (Timestamp Ordering) può causare l'**uccisione** di un gran numero di transazioni. Inoltre, funziona assumendo la **commit-proiezione**. Per eliminare questa ipotesi, è necessario **bufferizzare le scritture**, cioè effettuarle e trascriverle nella memoria di massa solo dopo il **commit**. Ciò comporta che le letture da parte di altre transazioni dei dati memorizzati nel buffer e in attesa di commit vengano anch'esse messe in attesa del commit della transazione scrivente (utilizzando meccanismi di attesa analoghi a quelli del 2PL stretto).

Quadro complessivo



Meccanismi per la gestione dei lock

Il lock manager svolge un ruolo cruciale nel sistema, essendo invocato da tutti i processi che desiderano accedere alla base di dati. Tali processi devono eseguire le seguenti operazioni: read lock, write lock e unlock, le quali sono caratterizzate dai seguenti parametri:

- `read lock(T, x, errorcode, timeout)`
- `write lock(T, x, errorcode, timeout)`
- `unlock(T, x)`

Dove:

- T rappresenta l'identificativo della transazione,
- x è l'elemento per il quale si richiede o si rilascia il lock,
- $errorcode$ indica un valore restituito dal lock manager (0 se la richiesta è stata soddisfatta, diverso da 0 altrimenti),
- $timeout$ è l'intervallo massimo di tempo che il chiamante è disposto ad aspettare per acquisire il lock sulla risorsa.

Le situazioni possibili sono tre:

1. **Richieste immediatamente soddisfatte:** Quando un processo richiede una risorsa e questa può essere immediatamente assegnata, il lock manager la assegna e aggiorna lo stato della risorsa. Il ritardo introdotto dal lock manager è minimo.
2. **Richieste non immediatamente soddisfatte:** Se una risorsa non può essere immediatamente assegnata, il processo viene messo in coda. L'attesa può essere lunga e il processo associato alla transazione viene sospeso. Quando la risorsa viene rilasciata, il lock manager la assegna al primo processo in coda. La probabilità di conflitto è proporzionale al numero di transazioni attive e al numero medio di risorse per transazione, e inversamente proporzionale al numero di risorse totali nella base di dati.
3. **Timeout di una richiesta non soddisfatta:** Se scatta il timeout e la richiesta non è stata soddisfatta, la transazione può eseguire un rollback seguito da un nuovo avvio. Altrimenti, può decidere di continuare richiedendo nuovamente il lock. Un fallimento nella richiesta di lock non comporta il rilascio delle risorse precedentemente acquisite.

Le **tabelle dei lock** sono frequentemente accedute e quindi solitamente risiedono nella memoria centrale per ridurre i tempi di accesso. La struttura delle tabelle prevede che ad ogni oggetto sia associato un set di due bit di stato, rappresentanti i tre possibili stati dell'oggetto, insieme a un contatore per monitorare il numero di processi in attesa.

Granularità dei lock

La scelta del livello di lock **dipende dalle risorse** a cui si intende accedere e se è necessaria la mutua esclusione. Questi possono essere:

- Intere tabelle
- Insiemi di tuple
- Singole tuple
- Campi di singole tuple

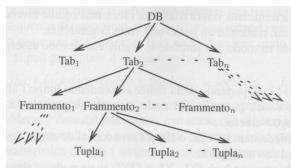
In molti sistemi è possibile specificare **diversi livelli di lock**, noti come **granularità dei lock**. La decisione sul livello di lock appropriato è presa dal progettista dell'applicazione o dall'amministratore del database.

Per scegliere il livello giusto, è necessario considerare diverse considerazioni:

- Un livello di lock **troppo elevato**, come quello delle tabelle, può ridurre il parallelismo e aumentare i conflitti.
- Un livello di lock **troppo basso** aumenta il rischio di fallimento dopo una notevole quantità di lavoro.

Il lock gerarchico

Il lock gerarchico estende il protocollo di lock tradizionale permettendo alle transazioni di definire in modo molto efficiente i lock di cui hanno bisogno, operando al livello prescelto della gerarchia (dal lock per l'intera base di dati al lock per una singola tupla).



Deadlock (stallo)

Un deadlock è una situazione in cui due o più processi o thread sono **bloccati indefinitamente** perché ciascuno attende che l'altro rilasci una risorsa che ha acquisito, impedendo così il progresso di entrambi. In altre parole, si verifica quando due o più processi sono bloccati in una condizione di stallo, dove **ciascuno detiene una risorsa necessaria dall'altro per procedere**.

Un esempio classico di deadlock coinvolge due processi: il primo acquisisce la risorsa A e attende la risorsa B, mentre il secondo acquisisce la risorsa B e attende la risorsa A. In questo caso, nessuno dei due processi può procedere, creando una situazione di stallo.

Soluzioni

I deadlock sono problemi critici nei sistemi concorrenti e possono causare il blocco dell'intero sistema se non vengono gestiti correttamente. Le tecniche per gestire i deadlock sono tre:

- **timeout** (la più usata): le transazioni attendono una risorsa per un tempo definito dopo del quale vengono negative;
- **rilevamento** (deadlock detection): eseguito periodicamente, consiste nella ricerca di cicli nei grafi;
- **prevenzione** (deadlock prevention): richiede il lock di tutte le risorse necessarie alla transazione in una sola volta.

Come scegliere le transazioni da uccidere

Nella scelta delle transazioni da interrompere (vittime), esistono due opzioni principali:

1. **Politiche interrompenti:** Questo approccio consente di risolvere i conflitti terminando la transazione che detiene la risorsa richiesta. In questo modo, la risorsa viene rilasciata e può essere concessa ad un'altra transazione. Le politiche interrompenti sono più aggressive nell'affrontare i deadlock poiché possono interrompere le transazioni in qualsiasi momento.

Si **uccidono** le transazioni che hanno svolto **meno lavoro**. Tuttavia, questo approccio può portare al problema del blocco individuale, dove una transazione, pur avendo svolto poco lavoro, viene ripetutamente interrotta perché entra in conflitto con altre transazioni che accedono frequentemente agli stessi oggetti.

Questa situazione può causare il blocco individuale, noto anche come **starvation**, dove una transazione **non riesce a procedere** a causa della **priorità sempre più bassa** rispetto alle altre transazioni.

Una possibile **soluzione** consiste nel mantenere invariato il timestamp delle transazioni abortite e riavviate. In questo modo, le transazioni **più vecchie** avranno una **priorità crescente** rispetto a quelle più recenti, consentendo loro di completare il proprio lavoro prima di essere interrotte nuovamente.

2. **Politiche non interrompenti:** In questo caso, una transazione può essere terminata solo quando effettua una nuova richiesta di risorsa. Questo approccio è meno aggressivo rispetto alle politiche interrompenti e può comportare una gestione più delicata dei deadlock, poiché le transazioni vengono terminate solo quando tentano di acquisire ulteriori risorse.

La scelta tra politiche interrompenti e non interrompenti dipende dalle esigenze del sistema, dalla tolleranza al rischio di perdita di lavoro e dal livello di priorità delle transazioni coinvolte.

Nella pratica...

La tecnica basata sulla prevenzione dei blocchi critici **non è comunemente utilizzata** nei DBMS commerciali. In genere, si preferisce adottare un approccio **reattivo**, dove si termina una transazione ogni volta che si verifica un **conflitto**. Tuttavia, poiché la probabilità di un blocco critico è molto minore rispetto alla probabilità di un conflitto, questa strategia è accettabile nella gestione dei deadlock. In media, si termina una transazione ogni due conflitti, garantendo una gestione efficace dei deadlock senza la necessità di prevenire attivamente i blocchi critici.

Gestione dell'affidabilità

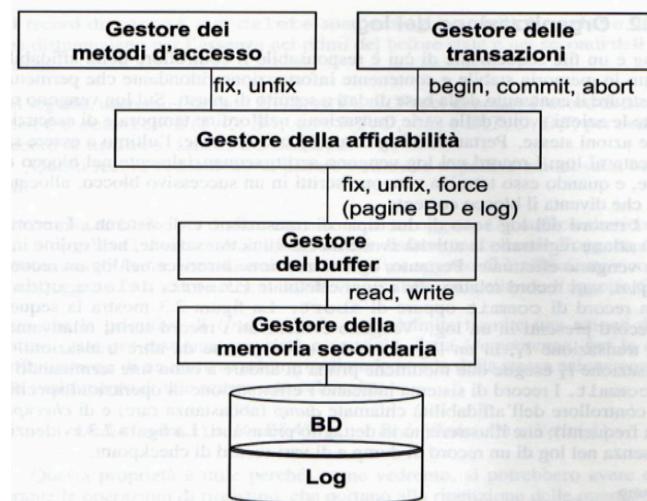
Il **controllore dell'affidabilità** si occupa di garantire due proprietà fondamentali delle transazioni: **atomicità** e **persistenza**. Questo viene realizzato attraverso la **gestione del log**, un archivio/file persistente che registra tutte le azioni svolte dal DBMS.

Ogni operazione di **scrittura** sulla base di dati viene **protetta tramite un'azione sul log**, consentendo di eseguire il **rollback** (undo) delle azioni in caso di malfunzionamenti o guasti prima del commit, oppure di **rieseguire** (redo) le azioni nel caso in cui la loro buona riuscita sia incerta e le transazioni abbiano già effettuato il commit.

Poiché il log svolge un ruolo fondamentale, è essenziale che sia **sufficientemente robusto**, in grado di **garantire l'integrità** e la **consistenza** delle transazioni anche in presenza di errori o guasti. Per esserlo, è necessario che venga memorizzato nella **memoria stabile**, ossia, una memoria resistente ai guasti (realisticamente impossibile). Tuttavia, i meccanismi di controllo dell'affidabilità vengono definiti come se la memoria stabile fosse effettivamente esente da guasti: un guasto della memoria stabile viene considerato un evento **catastrofico**.

Per ottenere questa stabilità si utilizzano **tecniche di replicazione** (si usano più dispositivi di memoria).

Architettura del gestore dell'affidabilità



I compiti del controllore dell'affidabilità

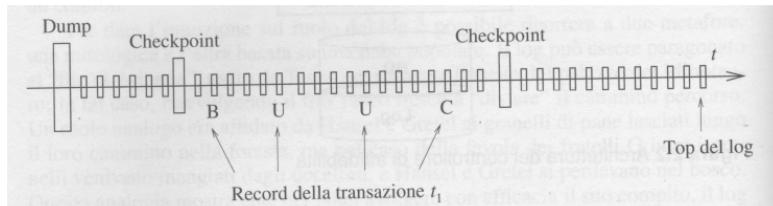
I compiti del controllore dell'affidabilità includono:

1. **Implementare** i comandi transazionali, come **begin transaction**, **commit work** e **rollback work**.
2. **Eseguire** le operazioni di **ripristino** (recovery) dopo malfunzionamenti o guasti, incluse la ripresa a caldo (hot recovery) e la ripresa a freddo (cold recovery).
3. **Gestire** le **richieste di accesso** alle pagine in lettura e scrittura, **trasferendole al gestore del buffer**, e **generando ulteriori** richieste di lettura e scrittura per garantire la robustezza e la resistenza ai guasti.

- Preparare i dati essenziali per i meccanismi di **ripristino dai guasti**, come ad esempio l'esecuzione di azioni di **checkpoint** e di **dump**, che consentono di creare **punti di ripristino** e salvare lo stato attuale del database su supporti di archiviazione esterni.

Organizzazione del file di log

Il log è un **file sequenziale** (append-only), gestito dal controllore dell'affidabilità e scritto in memoria stabile. Ogni azione eseguita dalle transazioni viene registrata sul log nell'**ordine temporale** di esecuzione. Il log è organizzato in **blocchi**, con un blocco corrente che rappresenta l'ultimo blocco allocato. I record di log vengono scritti sequenzialmente all'interno di questo blocco corrente.



I record del log sono di due tipi: record di **transazione** e record di **sistema**.

- Record di transazione:** Registrano le attività svolte dalle transazioni nell'ordine in cui vengono eseguite. Ogni transazione è registrata tramite un record di inizio (B) seguito da record di insert, delete e update (U) per le operazioni eseguite dalla transazione, e infine un record di commit (C) o di abort (A) per segnalare la conclusione della transazione con successo o insuccesso, rispettivamente.
- Record di sistema:** Registrano l'esecuzione di operazioni di sistema come dump (raramente utilizzati) e checkpoint (più frequenti). Questi record sono utilizzati per salvare lo stato del database su supporti di archiviazione esterni e creare punti di ripristino per garantire la consistenza e la robustezza del sistema.

Struttura dei record

La struttura dei record nel log è la seguente:

- Record di begin, commit e abort:** Contengono il tipo di record (B per begin, C per commit, A per abort) e l'identificativo della transazione t_i .
- Record di update:** Contengono l'identificativo della transazione t_i , l'identificativo dell'oggetto modificato O_j , e i valori BS_i (before state) e AS_i (after state) che rappresentano rispettivamente lo stato dell'oggetto O_j prima e dopo la modifica. BS_i e AS_i possono contenere copie complete delle pagine modificate.
- Record di insert e delete:** I record di insert non contengono il valore BS_i , mentre i record di delete non contengono il valore AS_i . Questi record registrano l'operazione di inserimento o cancellazione di un oggetto senza specificare lo stato precedente o successivo dell'oggetto stesso.

Le primitive undo e redo

Data una transazione T , indicheremo con $B(T)$, $C(T)$ e $A(T)$ i record di **begin**, **commit** e **abort** relativi a T , rispettivamente, e con $U(T, O, BS, AS)$, $I(T, O, AS)$ e $D(T, O, BS)$ i record di **update**, **insert** e **delete**, rispettivamente.

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

- primitive di **undo**: per disfare un'azione su un oggetto O , ricopia in O il valore BS ;
- primitive di **redo**: per rifare un'azione su un oggetto O , ricopia in O il valore AS .

Proprietà di idempotenza

Le operazioni di undo e redo godono della **proprietà di idempotenza**, il che significa che eseguire un numero arbitrario di volte l'undo o il redo di un'azione è equivalente a eseguire una sola volta quell'azione stessa. Formalmente:

$$\begin{aligned}\text{undo}(\text{undo}(A)) &= \text{undo}(A) \\ \text{redo}(\text{redo}(A)) &= \text{redo}(A)\end{aligned}$$

Questa proprietà è estremamente utile in caso di errori durante il ripristino. Se si verificano errori, è sufficiente ripetere le operazioni di undo o redo fino a quando non vengono completate con successo. Grazie alla proprietà di idempotenza, la ripetizione delle operazioni non comporta alcuna conseguenza negativa sullo stato del sistema.

L'operazione di checkpoint

L'operazione di **checkpoint** viene svolta **periodicamente**, in stretto coordinamento con il buffer manager, per registrare le transazioni attive. È una sorta di **fotografia** della situazione, mirata a vedere quali sono le transazioni attive in un certo istante. Per svolgere un'operazione di checkpoint:

1. si sospende l'accettazione di operazioni di scrittura, commit o abort, da parte di qualunque transazione;
2. si trasferiscono in memoria di massa (tramite opportune operazioni di force) tutte le pagine del buffer su cui sono state eseguite delle modifiche da parte di transazioni che hanno già effettuato il commit;
3. si scrive in modo sincrono (force) nel log un record di checkpoint che contiene gli identificatori delle transazioni attive;
4. si riprende l'accettazione delle operazioni sospese.

L'operazione di dump

L'operazione di **dump** produce una **copia completa della base di dati**, effettuata in mutua esclusione con tutte le altre transazioni quando il sistema non è operativo.

La copia viene memorizzata in **memoria stabile** (backup). Al termine del dump, viene scritto nel log un record di dump, che segnala l'avvenuta esecuzione dell'operazione in un dato istante. Il sistema riprende, quindi, il suo funzionamento normale. Convenzioni notazionali.

Useremo DUMP per denotare il record di dump e CK(T₁, T₂, ..., T_n) per denotare il record di checkpoint, dove T₁, T₂, ..., T_n sono gli identificatori delle transazioni attive all'istante del checkpoint.

Le regole

Durante il funzionamento normale delle transazioni, il gestore dell'affidabilità garantisce il rispetto delle seguenti due regole che impongono dei requisiti minimi che consentono di ripristinare la correttezza della base di dati in caso di guasti:

Write-ahead-log (WAL)

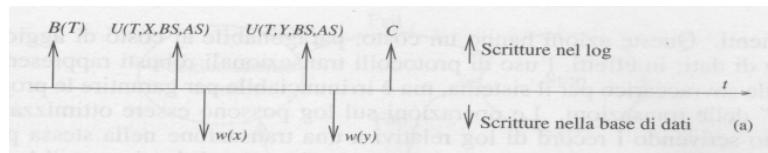
La regola del write-ahead-log (WAL) stabilisce che la parte relativa allo 'stato precedente' dei record di un log debba essere scritta nel log (tramite un'operazione di scrittura in memoria stabile) prima di eseguire l'operazione corrispondente sul database.

Commit-precedenza

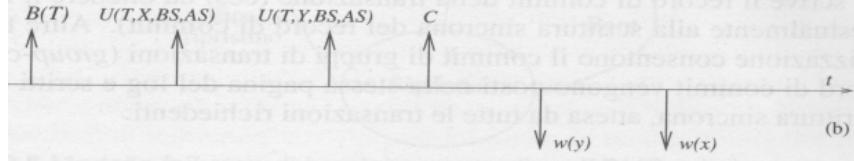
La regola del commit-precedenza impone che la parte relativa allo stato successivo dei record di un log venga scritta nel log (tramite un'operazione di scrittura in memoria stabile) prima di effettuare il commit.

Le politiche

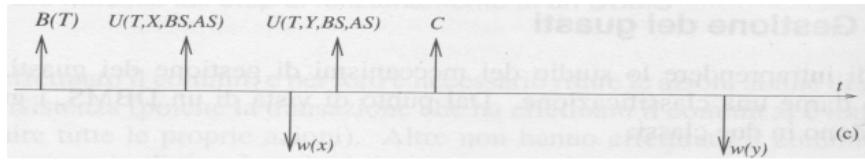
La transazione scrive inizialmente il record $B(T)$. Successivamente, esegue le operazioni di update scrivendo **prima** il record di log $U(T, O, BS, AS)$ e **poi** la pagina della base di dati, che passa dal valore BS al valore AS . Tutte queste pagine devono essere effettivamente scritte **prima del commit**, il quale comporta una scrittura sincrona (force). Al commit, tutte le pagine della base di dati modificate dalla transazione sono state scritte in memoria di massa. Tale schema non richiede operazioni di redo.



Nel seguente schema invece, la scrittura dei record di log **precede** quella delle azioni sulla base di dati, che avvengono **dopo** la decisione di commit e la conseguente scrittura sincrona del record di commit sul log. Tale schema non richiede operazioni di undo. Anche questa situazione soddisfa le due regole.



Nell'ultimo schema (più generale e comunemente usato), le scritture sulla base di dati, una volta protette dalle opportune scritture sul log, possono avvenire in un qualunque momento rispetto alla scrittura del record di commit sul log. Tale schema consente al gestore del buffer di ottimizzare le operazioni di flush relative ai suoi buffer, indipendentemente dal controllore dell'affidabilità. Tale schema può richiedere operazioni sia di undo sia di redo.



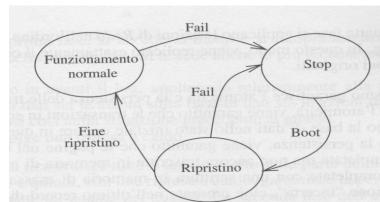
Gestione dei guasti

Distinguerai due categorie di guasti:

Guasti di sistema: Questi sono causati da difetti nel software, come il sistema operativo, o da interruzioni nel funzionamento dei dispositivi, come i black-out, che portano il sistema a uno stato inconsistente. Effetto: perdita del contenuto della memoria principale (buffer), ma conservazione del contenuto della memoria di massa (base di dati e log).

Guasti di dispositivo: Questi sono legati ai dispositivi utilizzati per gestire la memoria di massa, come il malfunzionamento delle testine di un disco rigido, che provoca la perdita del suo contenuto. Poiché il log risiede in una memoria stabile, un guasto di dispositivo può causare soltanto la perdita (parziale) del contenuto della base di dati, non del log.

Modello ideale di funzionamento di un DBMS:



Quando il sistema rileva un guasto, sia di tipo sistemico che legato al funzionamento, viene attuata una **procedura di arresto completo delle transazioni** seguita dal riavvio del sistema operativo (boot). Successivamente, viene avviata una procedura di ripristino, chiamata "**ripristino a caldo**" nel caso di guasti di sistema e "**ripristino a freddo**" nel caso di guasti di dispositivo.

Al termine delle operazioni di ripristino, il sistema torna disponibile per le transazioni. Il buffer è completamente svuotato e può riprendere a caricare pagine della base di dati e del log.

Come funzionano le procedure di ripresa?

Al verificarsi del guasto, si trovano un insieme di **transazioni potenzialmente attive** di cui non si sa se abbiano **completato** le loro azioni sulla base di dati, poiché il buffer manager ha perso tutte le informazioni utili.

Queste transazioni possono essere suddivise in due categorie in base alle informazioni presenti nel log:

- **Transazioni che hanno effettuato il commit:** serve ripetere le azioni per garantire la persistenza dei loro effetti.
- **Transazioni che non hanno effettuato il commit:** serve annullare le azioni poiché non è noto quali azioni siano state effettivamente eseguite. Di conseguenza, la base di dati deve essere riportata allo stato precedente l'esecuzione della transazione.

Esercizi schedule

Esercizio 1.

Sia dato lo schedule:

$$S_{10} : w_0(x) \ r_1(x) \ w_0(z) \ r_1(z) \ r_2(x) \ r_3(z) \ w_3(z) \ w_1(x)$$

E' possibile mostrare (algoritmo di ordinamento topologico applicato al corrispondente grafo dei conflitti) che S_{10} è equivalente rispetto ai conflitti allo schedule seriale S_{11} :

$$S_{11} : w_0(x) \ w_0(z) \ r_2(x) \ r_1(x) \ r_1(z) \ w_1(x) \ r_3(z) \ w_3(z)$$

Costruiamo un grafo in cui i nodi sono le transazioni t_0, t_1, t_2, t_3 . Ora è necessario trovare le coppie in conflitto: si parte da sinistra e si scorre verso destra nello schedule cercando i conflitti. Ad esempio $w_0(x)$ è una scrittura e quindi sarà in conflitto con tutte le altre operazioni relative ad x svolte da transazioni diverse da t_0 . La prima coppia in conflitto è

$$(w_0(x), r_1(x))$$

Quindi inseriamo un arco da t_0 a t_1 nel grafo dei conflitti. Un'altra coppia in conflitto è

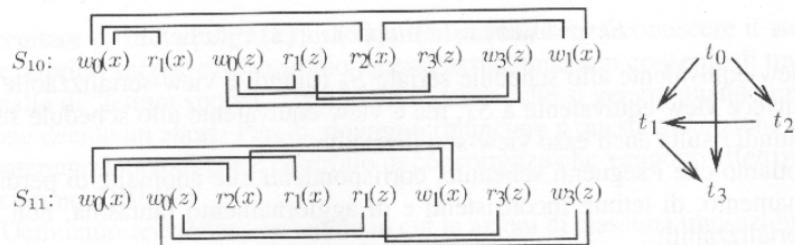
$$(w_0(x), r_2(x))$$

Quindi inseriamo un arco da t_0 a t_2 nel grafo dei conflitti. Un ultimo conflitto lo abbiamo sulla coppia

$$(w_0(x), w_1(x))$$

Che non ha alcun impatto sul grafo dato che l'arco da 0 a 1 lo abbiamo già. Ora passiamo ad $r_1(x)$, la quale essendo una **operazione di lettura** avrà **conflitti solo** con operazioni di **scrittura** svolte da transazioni **diverse da t_1** . Quindi non c'è alcun conflitto.

Continuando in questo modo e facendo lo stesso ragionamento anche per S_{11} , otteniamo la seguente situazione:



Il grafo ottenuto è **aciclico**. Dimostriamolo: per ogni nodo riporto il numero di archi uscenti:

- da t_0 escono 3 archi
- da t_1 esce un arco;
- da t_2 esce un arco;
- da t_3 non esce alcun arco;

Se il grafo è aciclico è immediato osservare che ci dev'essere almeno un nodo privo di archi uscenti. Avendo dunque ottenuto un grafo aciclico, $S_{10} \in CSR$ e quindi è uno schedule **serializzabile rispetto ai conflitti**.

Costruisco ora il mio **ordinamento topologico** a partire dall'ultimo nodo dell'ordinamento dove colloco uno dei nodi privi di archi uscenti. Una volta fatto, si elimina il nodo t_3 dal grafo assieme agli archi in esso entranti e si procede con il nuovo grafo: per ogni nodo riporto il numero di archi uscenti:

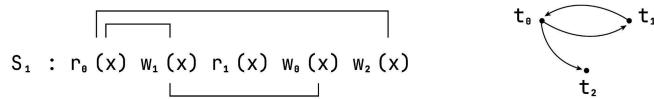
- da t_0 escono 2 archi
- da t_1 non esce alcun arco;
- da t_2 esce un arco;

Questo nuovo grafo deve sicuramente (e infatti lo è) essere aciclico. Il ragionamento procede inserendo nell'ordinamento topologico il nodo privo di archi uscenti, che ora è t_1 , che viene poi rimosso assieme ai suoi archi entranti e il procedimento ricomincia. Al termine dell'algoritmo si ottiene lo schedule seriale equivalente a S_{10} : $t_0 t_2 t_1 t_3$.

Esercizio 2.

Mostrare che lo schedule $S_1 : r_0(x) w_1(x) r_1(x) w_0(x) w_2(x)$ appartiene a VSR ma non a CSR.

Mostriamo che $S \notin CSR$ facendo il grafo dei conflitti:



Che presenta un ciclo, dunque $S \notin CSR$. Mostriamo ora che $S \in VSR$ costruendo la relazione legge e le scritture finali:

$$\begin{aligned} \text{Legge} &= \{(r_1(x), w_1(x))\} \\ \text{scritture finali} &= \{w_2(x)\} \end{aligned}$$

Verifichiamo se esiste uno schedule seriale equivalente (stessa relazione legge e stesse scritture finali) rispetto alle viste a S_1 . Sapendo che $\text{scritture finali} = \{w_2(x)\}$ segue che $t_0 < t_2$ e anche $t_1 < t_2$ e quindi t_2 dovrà necessariamente essere l'ultima transazione. Notiamo come $r_0(x)$ nello schedule non legga da nessuno (non compare nelle coppie della relazione legge), segue che t_0 deve precedere sia t_1 che t_2 . Quindi posso concludere che esiste lo schedule seriale equivalente $S : t_0 t_1 t_2$:

$$S_1 : r_0(x) w_0(x) w_1(x) r_1(x) w_2(x)$$

Dove abbiamo:

$$\begin{aligned} \text{Legge} &= \{(r_1(x), w_1(x))\} \\ \text{scritture finali} &= \{w_2(x)\} \end{aligned}$$

Questo ci permette di concludere che $S \in VSR$.

Esercizio 3. (22.01.2019)

Si stabilisca se i seguenti schedule appartengono o meno a 2PL, 2PL stretto, TS, CSR e VSR.

1. $s_1 : r_3(z), r_1(x), w_2(z), r_4(y), w_2(x), r_2(x), r_3(y), w_1(x), w_4(y);$

Provo a dimostrare $S \in VSR$.

```
legge = {(r2(x),w2(x))} 
scritture finali = {w4(y), w1(x), w4(z)}
```

Verifichiamo se esiste uno schedule seriale equivalente (stessa relazione legge e stesse scritture finali) rispetto alle viste a S_1 .

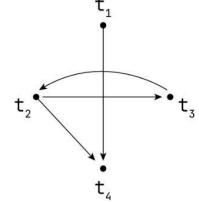
- Dal fatto che $\text{scritture finali} = \{\dots, w_1(x), \dots\}$ segue che $t_2 < t_1$ perché altrimenti ci sarebbe $w_2(x)$.
- Dal fatto che $(r_3(y), w_4(y)) \notin \text{legge}$, segue che $t_3 < t_4$. Altrimenti la coppia sarebbe nella relazione legge.
- Dal fatto che $(r_1(x), w_2(x)) \notin \text{legge}$, segue che $t_1 < t_2$. altrimenti la coppia sarebbe nella relazione legge.

Avendo trovato dei vincoli incompatibili, questo ci permette di concludere che $S \notin VSR$. Da il fatto che $CSR, 2PL, 2PL\ Stretto, TS \subseteq VSR$, segue che $S \notin CSR, 2PL, 2PL\ Stretto, TS$.

2. $s_2 : r_2(z), w_1(t), w_3(z), r_2(x), w_4(t), r_1(y), w_2(z), w_3(y), r_1(x), w_4(z), w_2(x);$

Provo a dimostrare $S \in CSR$.

$s_2 : r_2(z), w_1(t), w_3(z), r_2(x), w_4(t), r_1(y), w_2(z), w_3(y), r_1(x), w_4(z), w_2(x);$



Avendo trovato un ciclo, questo ci permette di concludere che $S \notin CSR$. Dal fatto che $2PL, 2PL\ Stretto, TS \subseteq CSR$, segue che $S \notin 2PL, 2PL\ Stretto, TS$.

Provo a dimostrare $S \in VSR$.

```
legge = {} 
scritture finali = {w2(x), w4(z), w3(y), w4(t)}
```

Verifichiamo se esiste uno schedule seriale equivalente (stessa relazione legge e stesse scritture finali) rispetto alle viste a S_1 .

- Dal fatto che $w_4(x)$ è una scrittura finale, segue che t_4 precede tutte le transazione su t , quindi $t_1 < t_4$;
- Dal fatto che $w_4(z)$ è una scrittura finale, segue che t_4 precede tutte le transazione su z , quindi $t_2 < t_4$ e $t_3 < t_4$;
- Dal fatto che $(r_2(z), w_3(z)) \notin \text{legge}$, segue che $t_2 < t_3$;
- Dal fatto che $(r_2(z), w_4(z)) \notin \text{legge}$, segue che $t_2 < t_4$;
- Dal fatto che $(r_1(x), w_2(x)) \notin \text{legge}$, segue che $t_1 < t_2$;

Quindi posso concludere che esiste lo schedule seriale equivalente $S : t_1 t_2 t_3 t_4$, ossia.

$S : w_1(t), r_1(y), r_1(x), r_2(z), r_2(x), w_2(z), w_2(x), w_3(t), w_3(y), w_4(t), w_4(z)$

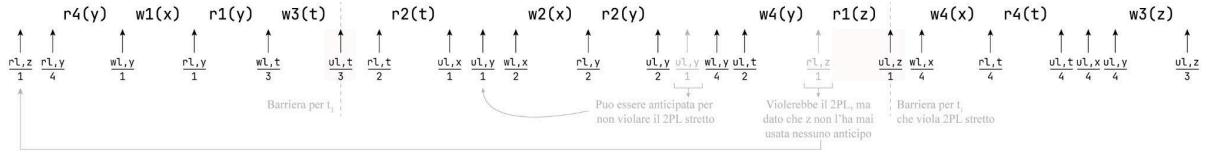
Dove abbiamo:

```
legge = { }
scritture finali = {w2(x), w4(z), w3(y), w4(t)}
```

Questo ci permette di concludere che $S \in VSR$.

3. $s_3 : r4(y), w1(x), r1(y), w3(t), r2(t), w2(x), r2(y), w4(y), r1(z), w4(x), r4(t), w3(z)$.

Provo a dimostrare $S \in 2PL$.



t_3 non può acquisire z prima di rilasciare t perché successivamente deve essere letta da t_1 . Dunque $S \notin 2PL$ e $S \notin 2PL Stretto$.

Potrebbe ancora stare in TS , CSR , VSR .

Provo a dimostrare $S \in TS$:

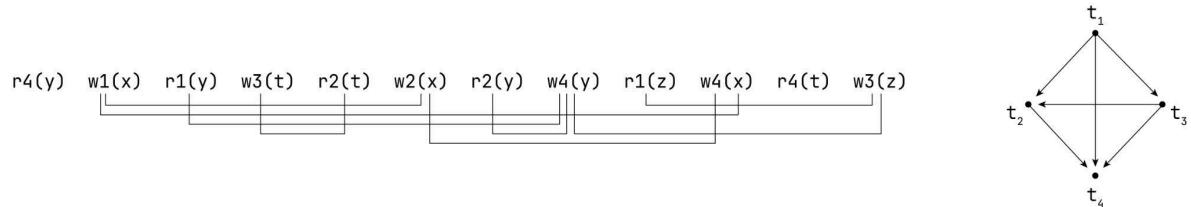
Per ogni variabile v assumiamo che $RTM(v) = WTM(v) = -1$:

```
r4(y)    ok RTM(y) = 4
w1(x)   ok WTM(x) = 1
r1(y)    ok RTM(y) rimane 4 perché 1 < 4
w3(t)   ok WTM(t) = 3
r2(t)    fallimento → WTM(t) = 3 > 2
```

Possiamo concludere che $S \notin TS$.

Provo a dimostrare $S \in CSR$:

Costruiamo il grafo dei conflitti



Dato che il grafo è aciclico, possiamo concludere che $S \in CSR$. Dato che $CSR \subseteq VSR$, da $S \in CSR$ segue che $S \in VSR$.

Esercizio 4. (3-7-2017)

Stabilire se i seguenti schedule sono o meno serializzabili rispetto al metodo del locking a due fasi, al metodo del locking a due fasi stretto e al metodo basato sui timestamp.

1. $r1(x), w1(x), r2(z), r1(y), w1(y), r2(x), w2(x), w2(z);$
2. $r1(x), r2(x), w2(x), r3(x), r4(z), w1(x), w3(y), w3(x), w1(y), w5(x), w1(z), w5(y), r5(z);$
3. $r1(x), r3(y), w1(y), w4(x), w1(t), w5(x), r2(z), r3(z), w2(z), w5(z), r4(t), r5(t).$

Primo schedule

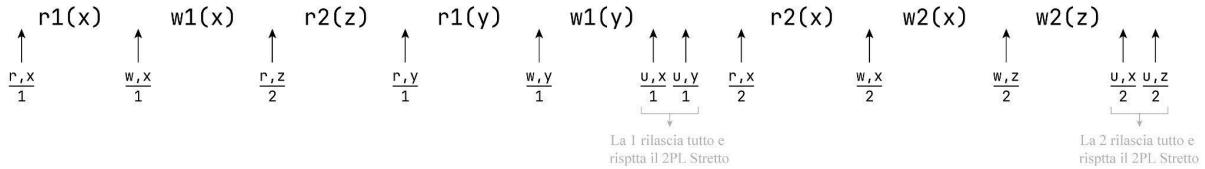
Provo a dimostrare $S \in TS$:

Per ogni variabile v assumiamo che $RTM(v) = WTM(v) = -1$:

$r1(x)$	ok	$RTM(x) = 1$
$w1(x)$	ok	$WTM(x) = 1$
$r2(z)$	ok	$RTM(z) = 2$
$r1(y)$	ok	$RTM(y) = 1$
$w1(y)$	ok	$WTM(y) = 1$
$r2(x)$	ok	$RTM(x) = 2$
$w2(x)$	ok	$WTM(x) = 2$
$w2(z)$	ok	$WTM(z) = 2$

Possiamo concludere che $S \in TS$.

Provo a dimostrare $S \in 2PL$ e $S \in 2PL$ Stretto:



Possiamo concludere che $S \in 2PL$ e $S \in 2PL$ Stretto.

Secondo schedule

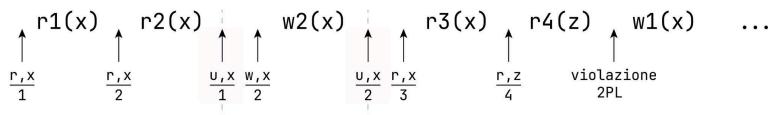
Provo a dimostrare $S \in TS$:

Per ogni variabile v assumiamo che $RTM(v) = WTM(v) = -1$:

$r1(x)$	ok	$RTM(x) = 1$
$r2(x)$	ok	$RTM(x) = 2$
$w2(x)$	ok	$WTM(x) = 2$
$r3(x)$	ok	$RTM(x) = 3$
$r4(z)$	ok	$RTM(z) = 4$
$w1(x)$	fallimento	$\rightarrow WTM(x) = 1 < 2$

Possiamo concludere che $S \notin TS$.

Provo a dimostrare $S \in 2PL$ e $S \in 2PL$ Stretto:



Possiamo concludere che $S \notin 2PL$ e $S \notin 2PL$ Stretto.

Terzo schedule

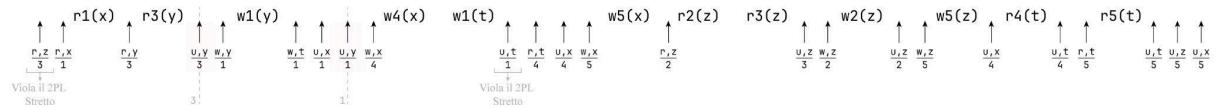
Provo a dimostrare $S \in TS$:

Per ogni variabile v assumiamo che $RTM(v) = WTM(v) = -1$:

$r1(x)$ ok $RTM(x) = 1$
 $r3(y)$ ok $RTM(y) = 3$
 $w1(y)$ fallisce $WTX(y) = 1 < 3$

Possiamo concludere che $S \notin TS$.

Provo a dimostrare $S \in 2PL$ e $S \in 2PL$ Stretto:



Possiamo concludere che $S \in 2PL$ e $S \notin 2PL$ Stretto.

Organizzazione Fisica dei Dati

I dati archiviati su disco sono organizzati in **file di record**, dove ogni record rappresenta un insieme di informazioni interpretabili come fatti relativi a entità, loro attributi e relazioni. Ad esempio, una tupla può essere memorizzata come un record, in cui il valore di un attributo è conservato in un campo del record stesso.

Esistono diverse tecniche per la memorizzazione fisica dei record di un file su un disco magnetico:

1. **Heap file** (file non ordinato): posiziona i record sul disco in modo non ordinato.
2. **Sorted file** (file ordinato): ordina i record in base al valore di un campo specifico.
3. **Hashed file** (file hash): utilizza una funzione di hash per determinare la posizione dei record sul disco.

Record di lunghezza fissa e variabile

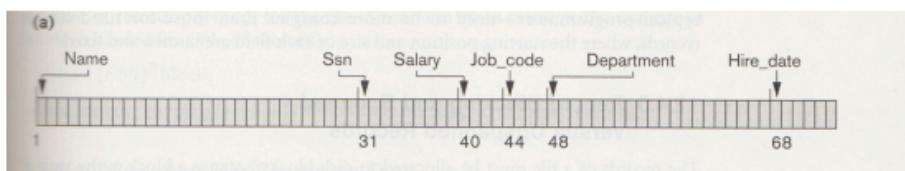
Un file è una **sequenza di record** (tabelle), di solito dello **stesso tipo**. Se i record di un file hanno tutti la stessa dimensione (in byte), il file è considerato composto da record di **lunghezza fissa**; altrimenti, si dice che è composto da record di **lunghezza variabile**.

Le ragioni per cui un file può avere record di lunghezza variabile possono essere diverse:

- uno o più campi hanno **dimensione variabile** (a.e., il campo NOME di IMPIEGATO);
- uno o più campi possono avere **valori multipli per singoli record**;
- uno o più campi sono **opzionali**;
- Il file contiene record di **più tipi** e di **dimensioni variabili**.

Esempio

Nel caso (a), il record IMPIEGATO di lunghezza fissa ha una dimensione di 71 byte e è composto da 6 campi, le cui dimensioni sono fisse. Questo consente di determinare la posizione del byte iniziale di ciascun campo rispetto alla posizione del byte iniziale del record. Per rappresentare un file di record di lunghezza variabile, è possibile utilizzare un file di record di lunghezza fissa, specificando la lunghezza massima di ogni campo. Per indicare che un determinato record manca di un valore per un campo opzionale, può essere utilizzato un valore speciale (nullo).



Nel caso (b), il record IMPIEGATO ha tre campi di lunghezza fissa (SSN, STIPENDIO e CODICE LAVORO) e due di lunghezza variabile (NOME e DIPARTIMENTO). Se un campo di lunghezza variabile non è opzionale e ogni record ha un valore per tale campo, ma non è possibile conoscere a priori la sua lunghezza esatta, possono essere utilizzati dei separatori opportuni (come ?, %, \$) per segnalare la fine dei valori dei campi di lunghezza variabile.

(b)

Name	Ssn	Salary	Job_code	Department	
Smith, John	123456789	XXXX	XXXX	Computer	
1	12	21	25	29	Separator Characters

Nel caso (c), più generale, in cui sono presenti campi opzionali e/o di lunghezza variabile, vengono utilizzati caratteri separatori distinti. Questi separatori servono a separare il nome di un campo dal suo valore, segnalare la fine di un campo e la fine di un record.

(c)

Name = Smith, John	Ssn = 123456789	DEPARTMENT = Computer	X	
<hr/>				
= Separates field name from field value				
█ Separates fields				
☒ Terminates record				

Organizzazione dei record in blocchi

Un **blocco** è un **insieme di record**, utilizzato per trasferire i record da memoria secondaria a memoria principale.

Generalmente la **dimensione del blocco** è superiore a quella del **singolo record** e si ha dunque che un blocco può contenere più record. Supponiamo che la dimensione del blocco sia pari a B byte. Assumendo che un dato file sia composto da record di dimensione fissa pari a R byte, con $B \geq R$, **in ogni blocco possono essere inseriti bfr record** (blocking factor).

Esempio: se $B = 120$ e $R = 15$, allora $bfr = 8$

$$bfr = \left\lfloor \frac{B}{R} \right\rfloor$$

In generale, B può non essere divisibile per R . Se imponiamo che ogni record sia interamente contenuto in uno e un solo blocco, ogni blocco conterrà dello spazio inutilizzato pari a:

$$B - (bfr \cdot R) \text{ byte}$$

Esempio: se $B = 120$ e $R = 25$, allora $bfr = 4$ e $B - (bfr \cdot R) = 20$ (più del 15% dello spazio viene sprecato). Tuttavia le cose possono andare ancora peggio:

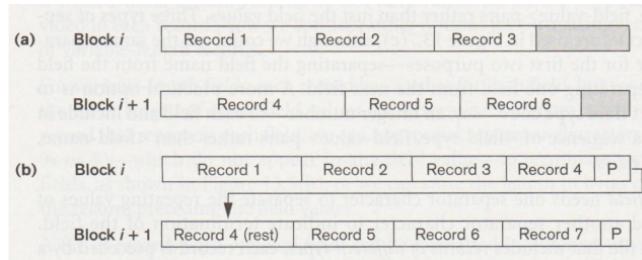
- Se la dimensione dei record è comparabile a quella del blocco (esempio, $B = 120$ e $R = 70$), il vincolo proposto risulta inaccettabile dato che in un blocco si riesce ad inserire solo un record sprecando 50 byte.
- Se la dimensione del record è maggiore di quella del blocco, ossia la condizione $B \geq R$ non risulta più soddisfatta (esempio, $B = 120$ e $R = 125$), diventa impossibile memorizzare i record.

Record spanned e unspanned

Ci sono due soluzioni:

- consentire la memorizzazione di un record in più blocchi, usando i puntatori per collegarli (**spanned record**);
- la suddivisione di un record in più blocchi sia vietata (**unspanned record**);

Un esempio di record spanned e unspanned é il seguente (organizzazione unspanned (a) e spanned (b) dei record)



Dimensionamento dei file

Se conosciamo la dimensione del blocco (B), la dimensione del record (R), e il numero di record contenuti nel file (r), possiamo determinare il **numero n_b di blocchi necessari per memorizzare il file**. Assumiamo un'organizzazione dei record di tipo unspanned, quindi ogni record occupa interamente uno o più blocchi senza essere suddiviso su più blocchi. Il numero di blocchi necessari puo essere calcolato utilizzando la seguente formula:

$$n_b = \left\lceil \frac{r}{bfr} \right\rceil$$

Dove:

- n_b è il numero di blocchi necessari,
- r è il numero di record del file,
- bfr è il blocking factor (il numero massimo di record contenuti in un blocco).

Allocazione dei blocchi di un file su disco

Esistono diverse tecniche consolidate per l'allocazione dei blocchi di un file su disco:

- **Allocazione contigua:** I blocchi del file sono allocati in blocchi consecutivi sul disco. Vantaggio: lettura rapida dell'intero file con la tecnica del doppio buffering. Svantaggio: espansione difficoltosa.
- **Allocazione con collegamenti:** Ogni blocco del file contiene un puntatore al blocco successivo. Vantaggio: espansione estremamente semplice. Svantaggio: scansione/lettura dell'intero file dispendiosa/lenta.
- **Combinazione delle due precedenti soluzioni:** Vengono definiti e allocati cluster (detti anche segmenti o extent) di blocchi consecutivi su disco, collegati fra loro tramite puntatori.
- **Allocazione indicizzata:** Uno o più blocchi di indici contengono i puntatori agli effettivi blocchi del file. Questa è la soluzione che viene implementata.

File header e ricerca dei record

Un **file header** (o descrittore del file) contiene le informazioni necessarie per determinare gli indirizzi sul disco dei blocchi del file.

La ricerca di un record su disco comporta la **copia di uno o più blocchi nel buffer** della memoria principale. Successivamente, specifici programmi cercano all'interno dei buffer il record desiderato, utilizzando le informazioni contenute nell'intestazione del file.

Quando l'indirizzo del blocco che contiene il record desiderato non è noto, i programmi di ricerca devono eseguire una ricerca **lineare attraverso** i blocchi del file: ogni blocco del file viene copiato nel buffer e la ricerca prosegue fino a quando non viene individuato il record desiderato o sono stati controllati tutti i record del file. Tale ricerca risulta **estremamente dispendiosa** per file di notevole lunghezza.

Operazioni

Le operazioni **record-at-a-time** sono operazioni applicate ad un singolo record, mentre le operazioni **set-at-a-time** sono operazioni applicate ad un intero file.

Operazioni record-at-a-time

Le operazioni record-at-a-time sono operazioni applicate ad un singolo record:

- **Find (o Locate)**: cerca il primo record che soddisfa una data condizione (record corrente) e trasferisce il blocco che contiene tale record in un buffer della memoria primaria.
- **Read (o Get)**: copia il record corrente dal buffer in una variabile del programma o in un'area di lavoro del programma, e fa avanzare il puntatore al record corrente al successivo record nel file.
- **FindNext**: individua il record successivo che soddisfa la condizione di ricerca e trasferisce il blocco che lo contiene in un buffer della memoria primaria.
- **Delete**: cancella il record corrente e, in alcuni casi, aggiorna il file sul disco.
- **Modify**: modifica i valori di alcuni campi del record corrente e, in alcuni casi, aggiorna il file sul disco.
- **Insert**: inserisce un nuovo record nel file individuando il blocco in cui il record deve essere inserito, trasferendo tale blocco in un buffer della memoria primaria, inserendo il nuovo record nel blocco contenuto nel buffer e, in tali casi, ricopiando il contenuto del blocco nel buffer sul disco.

Operazioni set-at-a-time

Le operazioni set-at-a-time sono operazioni applicate ad un intero file:

- **FindAll**: individua tutti i record di un file che soddisfano la condizione di ricerca.
- **FindOrdered**: recupera tutti i record presenti nel file in un ben preciso ordine.
- **Reorganize**: esegue un processo di riorganizzazione dei record del file, ad esempio, riordinando i record sulla base del valore che assumono su uno specifico campo.

Altre operazioni sono necessarie per preparare un file per l'accesso (**Open**) o per la chiusura (**Close**).

Organizzazione dei file e metodi di accesso

Esistono diverse tecniche per la memorizzazione fisica dei record di un file su un disco magnetico:

1. **Heap file** (file non ordinato): posiziona i record sul disco in modo non ordinato.
2. **Sorted file** (file ordinato): ordina i record in base al valore di un campo specifico.
3. **Hashed file** (file hash): utilizza una funzione di hash per determinare la posizione dei record sul disco.

I **metodi di accesso** variano a seconda dell'organizzazione dei file usata. Ad esempio, un metodo di accesso basato su indice non può ovviamente essere applicato ad un file che non è organizzato attraverso indici.

File di record non ordinati (heap file)

Nel caso più semplice di organizzazione, i record sono posizionati nel file **nell'ordine in cui vengono inseriti**: ogni nuovo record viene aggiunto alla fine del file.

Inserimenti

L'**inserimento** di un nuovo record può essere effettuato in modo **molto efficiente**:

- L'ultimo blocco del file (su disco) viene copiato in un buffer.
- Il nuovo record viene aggiunto nel blocco contenuto nel buffer.
- Il blocco aggiornato viene riscritto su disco.

L'indirizzo dell'ultimo blocco del file viene mantenuto nel file header.

Ricerche

La **ricerca** di un record in un heap file richiede una **ricerca lineare** su file blocco per blocco, una procedura computazionalmente **molto costosa**.

Se solo un record soddisfa la condizione di ricerca, mediamente sarà necessario controllare almeno il 50% dei blocchi del file su disco. Per un file di n_b blocchi, il programma dovrà controllare $n_b / 2$ blocchi nel caso medio; se nessun blocco soddisfa la condizione di ricerca, il programma dovrà controllare tutti i n_b blocchi.

Cancellazioni

Per **cancellare** un record, il programma deve effettuare le seguenti operazioni:

1. cercare il record;
2. copiare il blocco che lo contiene in un buffer;
3. cancellare il record dal buffer;
4. riscrivere il blocco sul disco.

In generale, ogni cancellazione **crea dello spazio inutilizzato** (wasted space) nel blocco su disco. .

Un'altra tecnica per cancellare record utilizza un **extra byte o bit** (deletion marker) associato ad ogni record. Il record è cancellato (logicamente) assegnando al deletion marker un certo valore. Un valore diverso dal deletion marker indica che il record è valido. Tale tecnica richiede, però, una **riorganizzazione periodica** del file per recuperare dello spazio di memoria.

Un altro problema

Per leggere tutti i record secondo l'**ordine** definito dai valori di uno o più campi, viene creata una copia ordinata (sorted) del file. L'**ordinamento** del file è computazionalmente **oneroso** per file di grandi dimensioni:

1. i record contenuti in ciascun blocco vengono ordinati;
2. coppie di blocchi ordinati vengono unite per creare gruppi (run) di record ordinati (di dimensione pari a 2 blocchi);
3. run di due blocchi vengono a loro volta uniti per formare run di 4 blocchi; tale operazione viene ripetuta fino a quando si ottiene il run finale che coincide con il file ordinato.

File di record ordinati (file ordinati)

È possibile disporre fisicamente i record di un file su disco **in base ai valori** di uno (o più) **dei loro campi**. Questo processo, chiamato "ordinamento dei campi", consente di organizzare i dati in modo che siano **ordinati** in modo crescente o decrescente rispetto ai valori di uno specifico campo o di più campi.

I file ordinati **raramente trovano applicazione pratica** nel contesto delle basi di dati, a meno che non sia presente una struttura di indicizzazione, come un indice primario, che consenta di stabilire un percorso ottimale per un accesso efficiente ai dati.

	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
		⋮				
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
		⋮				
	Akers, Jan					
Block 3	Alexander, Ed					
	Alfred, Bob					
		⋮				
	Allen, Sam					

Vantaggi

I file ordinati presentano evidenti vantaggi:

- La **lettura ordinata** rispetto ad un campo ordinante risulta **molto efficiente** (non serve ordinare).
- La **ricerca del record successivo** al record corrente è **efficiente** perché non richiede accessi aggiuntivi a blocchi (il record successivo e il record corrente si trovano nello stesso blocco, a meno che il record corrente non sia l'ultimo record del blocco).
- Una condizione di **ricerca basata sul valore del campo** (chiave) ordinante può essere verificata in modo **efficiente** attraverso l'uso della ricerca binaria, anziché la ricerca lineare.

File ordinati e ricerca binaria

Nella ricerca binaria, per un file con n_b blocchi, numerati da 1 a n_b , dove i record sono ordinati crescentemente basandosi sul valore di un campo chiave, la ricerca binaria richiede solitamente $\log_2(n_b)$ accessi ai blocchi. Questo è indipendente dalla presenza o meno del record, a confronto con i $n_b / 2$ accessi necessari in media nel caso di ricerca lineare quando il record è presente, e n_b accessi quando il record non è presente.

```

l := 1; u := nb
while u ≥ l do
    begin i := (l + u) DIV 2
    read block i of the file into the buffer
    if k < ordering key field value of the first record in the block
    then u := i - 1
    else if k > ordering key field value of the last record in the block
    then l := i + 1
    else if the record with ordering key field value = k is in the buffer
        then goto found
        else goto notfound
    end
goto notfound

```

Ricerca

L'**efficienza** delle ricerche basate sugli operatori di confronto ($>$, $<$, \geq , e \leq) in un campo chiave ordinato deriva dal fatto che i record che soddisfano tali condizioni sono fisicamente contigui all'interno del file.

Ad esempio, considerando una condizione di ricerca basata sul campo NAME con l'operatore $<$ e il valore 'F', il sistema selezionerà tutti i record dal principio del file fino al primo record il cui campo NAME inizia con 'F'. Questa contiguità permette di utilizzare metodi di ricerca rapidi e limitati a specifiche sezioni del file, ottimizzando significativamente il tempo di accesso rispetto a un file non ordinato.

Inserimenti

L'inserimento di un record è **estremamente inefficiente** dato che comporta diversi passaggi:

1. È necessario **individuare la posizione appropriata** in cui il nuovo record deve essere inserito nel file esistente.
2. Occorre **creare lo spazio necessario** per l'inserimento del nuovo record.

La creazione di spazio spesso implica lo **spostamento dei record esistenti** per fare spazio al nuovo record. In media, questo richiede lo spostamento della metà dei record del file, ovvero leggere e riscrivere metà dei blocchi del file.

Esistono diverse soluzioni per affrontare questo problema:

- a. Mantenere dello **spazio libero in ciascun blocco** per gli eventuali inserimenti di nuovi record.
- b. Creare un **file temporaneo non ordinato** (file di overflow" o "transaction file"), dove i nuovi record vengono inseriti alla fine del file di overflow. Periodicamente, il file di overflow viene unito al file principale attraverso un'operazione di **riorganizzazione**.

Modifica

La modalità di esecuzione per modificare il valore di un campo in un record dipende da due fattori principali:

1. La condizione di ricerca del record da modificare.
2. Il campo specifico che si intende modificare.

Se la condizione di ricerca impiega il campo chiave ordinato, è possibile individuare il record tramite una ricerca binaria. Altrimenti, sarà necessaria una ricerca lineare.

Ricerca dei file con funzioni di hashing

Un'altra forma di organizzazione primaria dei file si basa sulle **tecniche di hashing**, che consentono un accesso estremamente rapido ai record sotto determinate condizioni di ricerca (conosciute come hash o file diretti). La condizione di ricerca solitamente è un'uguaglianza su un singolo campo, noto come campo hash del file, che spesso coincide con la chiave del file (hash key).

Il funzionamento delle tecniche di hashing (hashing esterno) prevede l'utilizzo di una **funzione di hash** che, applicata al **valore del campo hash** di un record, **fornisce l'indirizzo del blocco** su disco in cui il record è memorizzato.

Le tecniche di hashing possono anche essere utilizzate per gestire file di record temporanei, di dimensioni ridotte, all'interno di un programma (hashing interno).

Internal Hashing

Per i **file interni**, la tecnica di hashing viene implementata tramite **array di record**. Il funzionamento del hashing interno prevede l'uso di un array con un indice che varia da 0 a $m - 1$, dove m rappresenta il numero totale di slot, e il cui indirizzo corrisponde all'indice dell'array.

Si utilizza una **funzione di hash** che trasforma il valore del campo hash in un numero intero compreso tra 0 e $m - 1$. Ad esempio, se il campo hash è di tipo intero, una funzione di hash comune può essere

$$h(k) \equiv k \bmod m$$

dove k è il valore del campo hash e MOD indica l'operazione modulo, restituendo l'indirizzo del record come il resto della divisione intera k/m , che è un numero intero compreso tra 0 e $m - 1$.

Esempio

Si consideri una tabella IMPIEGATO con campi NAME, SSN, JOB e SALARY.

	NAME	SSN	JOB	SALARY
0				
1				
2				
3				
•				
•				
M - 2				
M - 1				

La funzione di hash $h(SSN) = SSN \text{ MOD } m$ si applica al numero di previdenza sociale (SSN), dove il risultato di questa funzione corrisponde all'indice dell'array in cui il record sarà memorizzato.

Le collisioni

La maggior parte delle funzioni di hashing **non può garantire** che **valori distinti** del campo hash **corrispondano a indirizzi distinti**, poiché l'insieme dei possibili valori del campo hash è spesso molto più grande dello spazio degli indirizzi disponibile.

Quando l'indirizzo calcolato dalla funzione di hashing per un nuovo record coincide con quello già assegnato a un altro record, si verifica una **collisione**. In questo caso, è necessario trovare un'altra posizione dove inserire il nuovo record. Questo processo di ricerca di una nuova posizione è chiamato risoluzione delle collisioni (**collision resolution**). Ci sono diverse tecniche:

- **Open addressing**: partendo dalla posizione (occupata) determinata dalla funzione di hashing, controlla le posizioni successive finché non trova una posizione libera oppure ritorna alla posizione iniziale;
- **Multiple hashing**: si utilizza una seconda funzione di hash se l'applicazione della prima ha prodotto una collisione. Se si verifica una seconda collisione, o si utilizza il metodo di open addressing o si applica una terza funzione di hash, e così via.
- **Chaining**: si considerano diverse locazioni di overflow, estendendo l'array con un certo numero di posizioni. Inoltre, esso aggiunge un campo puntatore ad ogni posizione dell'array. Ogni collisione è risolta posizionando il nuovo record in una locazione di overflow non utilizzata e settando il puntatore dell'ultima posizione occupata incontrata all'indirizzo della locazione di overflow (vedi la figura alla pagina successiva).

L'obiettivo principale di una funzione di hashing è **distribuire uniformemente i record nello spazio degli indirizzi** al fine di minimizzare le collisioni e utilizzare in modo efficiente lo spazio disponibile, evitando di lasciare troppe locazioni inutilizzate. Simulazioni e studi hanno dimostrato che una soluzione ottimale consiste nel mantenere la tabella di hash completa al 70-90%, in modo da mantenere **basso il numero di collisioni** e allo stesso tempo **non sprecare troppo spazio**.

Ad esempio, se dobbiamo memorizzare r record nella tabella, è consigliabile utilizzare uno spazio degli indirizzi che comprenda m locazioni in modo che il rapporto r/m sia compreso tra 0.7 e 0.9.

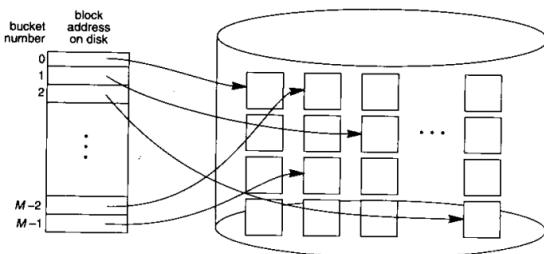
Va notato che alcune funzioni di hash richiedono che il numero di locazioni, m , sia una potenza di 2.

External hashing

La tecnica di hashing per i **file su disco** è nota come hashing esterno.

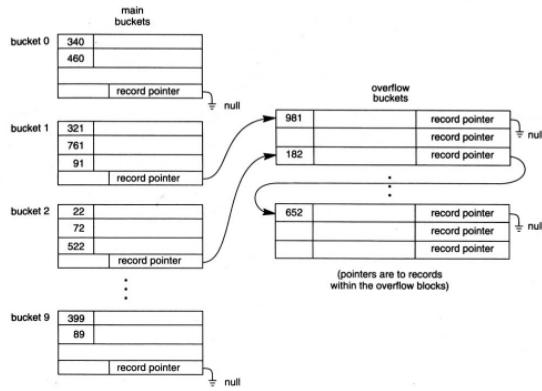
Considerando le caratteristiche della memorizzazione su disco, lo spazio degli indirizzi è composto da **celle** (o bucket), ciascuna delle quali può **contenere più blocchi**. Una cella può essere formata sia da un singolo blocco che da un insieme di blocchi contigui.

Nel file header è presente una tabella che converte il numero della cella nel corrispondente indirizzo del blocco sul disco, come illustrato nella figura



Il **problema delle collisioni** è meno grave rispetto all'hashing interno, poiché **molteplici record** possono essere contenuti **nella stessa cella**. Tuttavia, i problemi sorgono quando una cella è **piena** e un nuovo record deve essere inserito al suo interno.

In tal caso, si adotta una **variante del metodo di chaining** già discusso, dove **ogni cella contiene un puntatore ad una lista di record di overflow** specifica per quella cella, come mostrato nella figura.



Un **limite** rilevante delle tecniche di hashing è che lo spazio allocato per un file è fisso:

- Se alloca uno **spazio molto grande** al file si spreca spazio inutilmente;
- Se alloca uno **spazio molto piccolo** si rischia di dover essere costretti a costruire molte catene di overflow.

Come scegliere la grandezza dei file

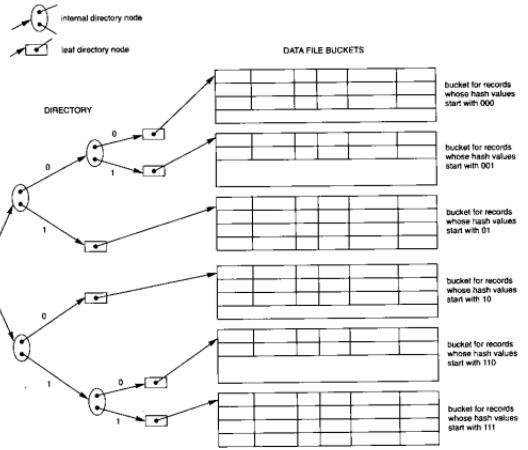
La soluzione è quella di avere un modo per **espandere dinamicamente la dimensione del file**. Questo comporta la necessità di avere delle funzioni di hash che supportino questa possibilità. Esistono (almeno) tre tecniche di hashing diverse che permettono di **espandere dinamicamente** i file:

Dynamic hashing

Si parte con una sola cella allocata per il file. Quando tale cella è piena, si effettua lo **splitting**, ossia si divide la cella. I record contenuti devono quindi essere distribuiti tra le due nuove celle. Per farlo si guarda il primo bit (bit più significativo) dei loro valori di hash.

- Se è 0 allora i record vengono memorizzati nella prima cella;
- Se è 1 allora i record vengono memorizzati nella seconda cella;

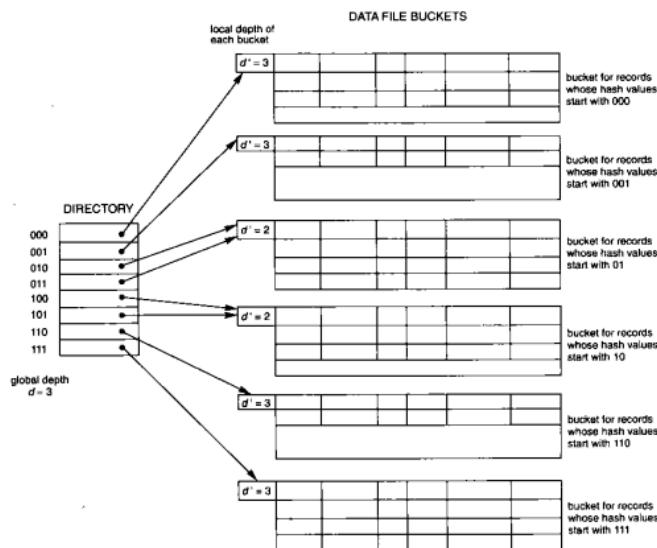
Si procede poi con l'inserimento nelle due nuove celle: quando una delle due è piena, si riesegue lo split, basandosi però, sul secondo bit dei loro valori di hash. In questo modo si crea una **struttura ad albero binario** (non necessariamente bilanciato), chiamata **directory o index**:



La **directory** è mantenuta nella **memoria principale** fino a quando non diventa troppo grande. Se supera le dimensioni di un singolo blocco, viene distribuita su due o più livelli.

Extendible hashing

Utilizza la stessa idea del dynamic hashing ma invece di usare un albero, sfrutta un **vettore** con un numero di celle che è una potenza di due. L'esponente d è detto **global depth** della directory. Per accedere a questa struttura ausiliaria si guarda il valore intero corrispondente ai primi d bit del valore di hash. Ad ogni cella si alloca una **local depth** d' , generalmente minore di d , che dice qual è il numero di bit su cui si basa la cella. La global depth d può aumentare o diminuire, raddoppiando o dimezzando il numero di elementi dell'array/directory. Il raddoppio si rende necessario se in una cella la cui local depth d' è uguale alla global depth d si verifica un overflow ($d = d'$). Il dimezzamento è possibile quando, dopo aver effettuato alcune operazioni di cancellazione, si ha che $d > d'$ per tutte le celle.



$d = \text{global depth} = 3$; $d' = \text{local depth} = \text{numero di bit sui quali si basa il contenuto di una cella } d' = 2 \text{ o } d' = 3$

Linear hashing

L'approccio del linear hashing consente a un file hash di adattare dinamicamente il numero di celle senza richiedere una directory esterna. Inizialmente, supponiamo che il file contenga un numero di celle m , numerate da 0 a $m-1$, e utilizzi una **funzione di hash** (funzione di hash iniziale h_0):

$$h_0(k) = k \bmod m$$

Le **collisioni**, che provocano overflow, sono gestite in modo tradizionale (chaining), **assegnando a ciascuna cella una catena di overflow**. Tuttavia, quando una nuova collisione porta alla situazione in cui **ogni cella del file ha almeno un record di overflow**, si attiva una **fase di espansione**.

Durante questa fase, la **prima cella** del file (cella 0) viene **divisa in due celle**:

- la cella originale 0;
- una nuova cella m alla fine del file;

I record originalmente contenuti nella cella 0 vengono **distribuiti** tra le due celle in base alla **nuova funzione di hash**:

$$h_1(k) = k \bmod 2m.$$

Che semplicemente fa la divisione di k per $2m$. Siccome k / m dava 0, il resto di $k / 2m$ può essere 0 o m . In questo modo si ridistribuiscono tutti i record della cella 0 tra la cella 0 e la m . Quando un record dev'essere inserito, viene applicata h_0 :

Se h_0 dà come risultato:

- 0: viene applicato il ragionamento con h_1 . Il valore di ritorno decide se assegnare il record alla cella 0 o alla m .
- 1: viene riapplicato il ragionamento con h_1 . Il valore di ritorno decide se assegnare il record alla cella 1 o alla $m+1$.
- ...
- n: viene riapplicato il ragionamento con h_1 . Il valore di ritorno decide se assegnare il record alla cella n o alla $m+n$.

Quando tutte le celle originali sono state divise in due ($n = m$), la funzione di hash h_1 va applicata direttamente a tutti i record del file. Ci si può dunque **dimenticare** di h_0 e ad ogni nuova collisione che causa un overflow viene ripetuto il processo descritto in precedenza, utilizzando questa volta la funzione di base h_1 ed una nuova funzione di hashing:

$$h_2(k) = k \bmod 4m$$

Dunque, quando un record dev'essere inserito, viene applicata h_1 :

Se h_1 dà come risultato:

- 0: viene applicato il ragionamento con h_2 . Il valore di ritorno decide se assegnare il record alla cella 0 o alla m .
- 1: viene riapplicato il ragionamento con h_2 . Il valore di ritorno decide se assegnare il record alla cella 1 o alla $m+1$.
- ...
- n: viene riapplicato il ragionamento con h_2 . Il valore di ritorno decide se assegnare il record alla cella n o alla $m+n$.

In generale, viene usata una sequenza di funzioni di hashing del tipo:

$$h_j(k) = k \bmod (2^j m)$$

con $j = 0, 1, 2, \dots$. Una nuova funzione di hashing h_{j+1} (e la ri-inizializzazione a 0 di n) si rende necessaria quando tutte le celle $0, 1, \dots, (2^j m) - 1$ sono state spartite.

```

if n = 0
    then m := hj(K) # m is the hash value of record with hash key K
else begin
    m := hj (K)
    if m < n then m := hj+1(K)
    end
search the bucket whose hash value is m (and its overflow, if any)

```

Le celle che sono state spezzate in due possono essere **ricominate** qualora il carico del file (**file load factor**) cada al di sotto di una certa soglia. Si definisce file load factor il seguente valore

$$l = r / (bfr \cdot n)$$

dove

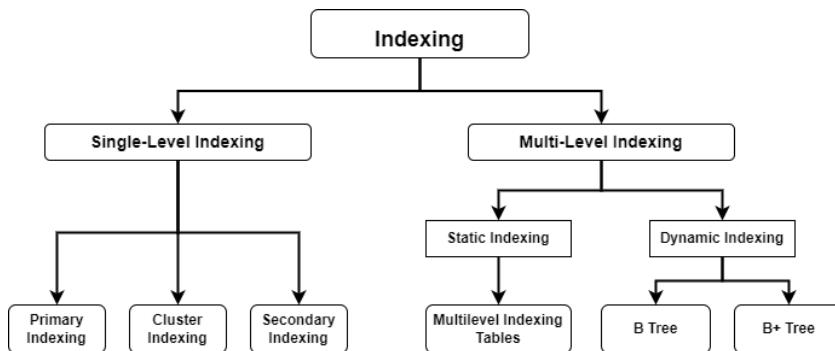
- r è il numero corrente di record presenti nel file,
- bfr è il numero massimo di record che possono essere contenuti in una cella,
- n è il numero corrente di celle del file.

Il file load factor può essere usato per mantenere ottimale il numero di celle del file, eseguendo divisioni o combinazioni di celle ogni qualvolta il fattore di carico oltrepassa opportune soglie prefissate (indicativamente, combinazioni se $l < 0.7$ e divisioni se $l > 0.9$).

Ricerca dei file con strutture ad indice per file

Gli **indici** sono strumenti fondamentali nell'accesso ai dati, progettati per ottimizzare il recupero dei record in base a determinate condizioni di ricerca. Esistono **diversi tipi di indici**, noti anche come percorsi di accesso secondario, che offrono modalità alternative per individuare rapidamente i record in base a campi specifici (campi di indicizzazione), senza influenzare direttamente la disposizione fisica dei record sul disco.

Una ricerca senza l'utilizzo di indici, richiede $\log_2 b_i$ accessi a blocco per un indice con b_i blocchi.

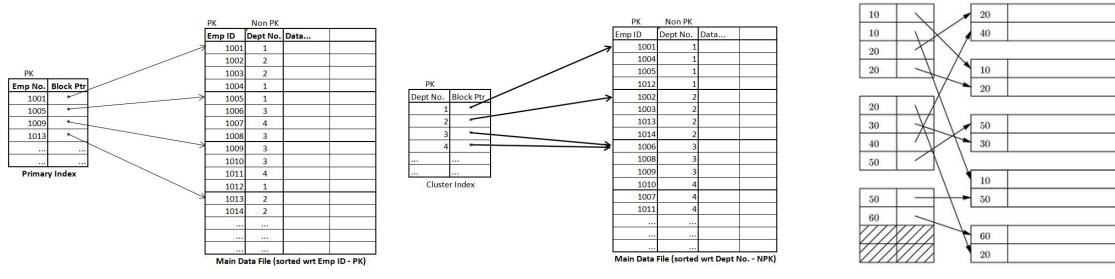


Indici di singolo livello

La struttura di un indice ordinato è simile all'indice (ordinato) di un **libro di testo**. Ogni parola è associata a una lista dei numeri di pagina in cui compare nel testo. In alternativa, per trovare una parola nel testo, è necessario scorrere il testo parola per parola, una tecnica chiamata ricerca lineare.

Un indice ordinato è **definito su un singolo campo** di un file, chiamato campo di indicizzazione. Ogni indice memorizza per ogni valore del campo una **lista di puntatori ai blocchi** su disco che contengono i **record con quel valore di campo**.

Gli indici sono **ordinati** in modo tale da consentire una **ricerca binaria**. Il file di indice è molto **più piccolo rispetto al file di dati**, il che rende la ricerca binaria **molto efficiente**. Questo approccio consente di individuare rapidamente i record desiderati nel file di dati senza doverli cercare uno per uno, migliorando notevolmente le prestazioni complessive del sistema di gestione dei dati.



Indice primario

Indici di clustering

Indici secondari

1. Indici primari

Un **indice primario** è un indice **non denso** (include una entry per ogni blocco) definito rispetto al **campo chiave** di un file di record **fisicamente ordinato rispetto a tale campo** (ordering key field).

Esercizio esame

Si consideri un file contenente $r = 30000$ record di dimensione prefissata pari a $R = 100$ byte, memorizzati in blocchi di dimensione pari a $B = 1024$ byte in modo unspanned. La dimensione del campo chiave primaria V sia $V = 9$ byte; la dimensione del puntatore a blocco P sia $P = 6$ byte. La dimensione di ogni index entry $R_i = 9 + 6 = 15$ byte.

ricerca binaria sul file di dati (senza indice)	ricerca binaria sul file indice primario
$bfr = \lfloor B/R \rfloor = \lfloor 1024/100 \rfloor = 10$ record per blocco $n_b = \lceil r/bfr \rceil = \lceil 30000/10 \rceil = 3000$ blocchi $\log_2 nb = \log_2 3000 = 12$ accessi a blocco	$bfr = \lfloor B/R \rfloor = \lfloor 1024/100 \rfloor = 10$ record per blocco $bfr_i = \lfloor B/R_i \rfloor = \lfloor 1024/15 \rfloor = 68$ index entry per blocco $n_b = \lceil r/bfr \rceil = 3000$ entry nell'indice $nb_i = \lceil n_b/bfr_i \rceil = \lceil 3000/68 \rceil = 45$ blocchi $\lceil \log_2 nb_i \rceil = \lceil \log_2 45 \rceil = 6$ accessi a blocco

Dunque, per cercare un record utilizzando l'indice occorrono 6 accessi a blocchi dell'indice più 1 accesso al blocco del file contenente il record, per un totale di $6 + 1 = 7$ accessi, contro i 12 accessi richiesti da una ricerca binaria sul file dei dati (senza uso dell'indice).

Problematiche degli indici primari

Il problema principale degli indici primari è che gestire **l'inserimento** e la **cancellazione di record** comporta la necessità di **modificare anche le voci di indice corrispondenti**. Per affrontare questo problema, esistono diverse strategie:

1. **File di overflow non ordinato:** file separato per gestire i record che non possono essere inseriti nel file principale a causa della mancanza di spazio o per altri motivi.
2. **Lista di record di overflow:** aggiungere una lista di record di overflow ad ogni blocco del file di dati. Quando un blocco diventa pieno e non può ospitare ulteriori record, i nuovi record vengono inseriti nella lista di overflow associata a quel blocco. In questo modo, non è necessario modificare l'indice primario, ma si aggiunge complessità alla gestione dei record di overflow all'interno dei blocchi.

2. Indici di clustering

L'**indice di clustering** è un indice primario in cui il campo rispetto il quale si ordina **non è la chiave** primaria (più record nel file possono assumere lo stesso valore rispetto all'ordering field). In questo caso, il campo rispetto al quale si ordina è detto **clustering field**.

Problematiche degli indici di clustering

L' inserimento e la cancellazione di record creano dei problemi anche in questo caso, in quanto i record di dati sono fisicamente ordinati.

Per ridurre il problema nel caso di inserimenti, si riserva un intero blocco per ogni valore del clustering field; tutti i record con quel valore sono posizionati nel relativo blocco ed eventualmente in blocchi addizionali ad esso collegati.

3. Indici secondari

Un **indice secondario** è un indice **denso** (include una entry per ogni record) in cui il file dei dati **non è ordinato rispetto al campo di indicizzazione**. Il vantaggio è che è possibile associare più indici ad un certo file. Il vantaggio è che è possibile associare più indici ad un certo file.

Indici primari vs indici secondari

- Un indice secondario richiede molto più spazio e un tempo di ricerca maggiore rispetto a un indice primario a causa dell'elevato numero di voci (indice denso).
- la riduzione del tempo di ricerca di un record basato sul valore assunto nel campo di indicizzazione è molto significativa. In questi casi, disponendo solo dell'indice primario, occorre eseguire una scansione lineare.

Esercizio esame

Si consideri un file contenente $r = 30000$ record di dimensione prefissata pari a $R = 100$ byte, memorizzati in blocchi di dimensione pari a $B = 1024$ byte in modo unspanned. La dimensione del campo chiave primaria V sia $V = 9$ byte; la dimensione del puntatore a blocco P sia $P = 6$ byte. La dimensione di ogni index entry $R_i = 9 + 6 = 15$ byte.

ricerca binaria sul file indice primario	ricerca binaria sul file indice secondario
$bfr = \lfloor B/R \rfloor = \lfloor 1024/100 \rfloor = 10$ record per blocco $bfr_i = \lfloor B/R_i \rfloor = \lfloor 1024/15 \rfloor = 68$ index entry per blocco $n_b = \lceil r/bfr \rceil = 3000$ entry nell'indice $nb_i = \lceil n_b/bfr_i \rceil = \lceil 3000/68 \rceil = 45$ blocchi $\lceil \log_2 nb_i \rceil = \lceil \log_2 45 \rceil = 6$ accessi a blocco	$bfr_i = \lfloor B/R_i \rfloor = \lfloor 1024/15 \rfloor = 68$ index entry per blocco $nb_i = \lceil r/bfr_i \rceil = \lceil 30000/68 \rceil = 442$ blocchi $\lceil \log_2 nb_i \rceil = \lceil \log_2 442 \rceil = 9$ accessi a blocco.

Per cercare un record utilizzando l'indice secondario, occorre 1 accesso addizionale a un blocco del file, per un totale di $9 + 1 = 10$ accessi, contro i 1500 accessi necessari in media per una ricerca lineare (poco più dei 7 accessi richiesti dalle ricerche che sfruttano l'indice primario).

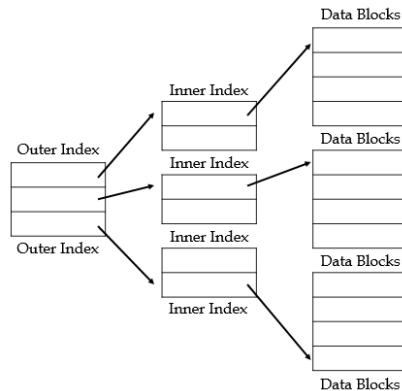
La differenza nei calcoli tra i due indici sta sul numero di entry dell'indice r_i :

- l'indice primario (non denso) $r_i = n_b$ numero di blocchi necessari per memorizzare il file,
- l'indice secondario (denso) $r_i = r$ numero di record del file.

Indici multilivello

1. Indici multilivello statici (alberi)

Una ricerca binaria sul file indice il numero di accessi medi al blocco è $\log_2 b_i$ dove b_i è il numero di blocchi. L'idea dell'**indice multilivello** è di **ridurre** ad ogni **iterazione** la porzione di indice che **resta da analizzare** di un **fattore maggiore di due**.



Utilizzando questi indici infatti, ad ogni iterazione si divide il numero di elementi da analizzare di un valore pari a quello del blocking factor dell'indice (detto **fan-out** fo), che è **molto maggiore** rispetto a 2.

Esercizio esame

Si consideri un file contenente $r = 30000$ record di dimensione prefissata pari a $R = 100$ byte, memorizzati in blocchi di dimensione pari a $B = 1024$ byte in modo unspanned. La dimensione del campo chiave primaria V sia $V = 9$ byte; la dimensione del puntatore a blocco P sia $P = 6$ byte. La dimensione di ogni index entry $R_i = 9 + 6 = 15$ byte.

Ricerca basata su indice multilivello statico ottenuto a partire da un indice primario costruito sul campo chiave primaria V	Ricerca basata su indice multilivello statico ottenuto a partire da un indice secondario costruito sul campo chiave primaria V
$bfr = \lfloor B/R \rfloor = \lfloor 1024/100 \rfloor = 10$ record per blocco $bfr_i = \lfloor B/R_i \rfloor = \lfloor 1024/15 \rfloor = 68$ index entry per blocco $n_b = \lceil r/bfr \rceil = 3000$ entry nell'indice $nb_i = \lceil n_b/bfr_i \rceil = \lceil 3000/68 \rceil = 45$ blocchi	$bfr_i = \lfloor B/R_i \rfloor = \lfloor 1024/15 \rfloor = 68$ index entry per blocco $nb_i = \lceil r/bfr_i \rceil = \lceil 30000/68 \rceil = 442$ blocchi
Dopo aver calcolato il numero di blocchi dell'indice primario nb_i trovo il numero di blocchi per ogni livello: <ul style="list-style-type: none"> • $b_1 = nb_i = 45$ blocchi • $b_2 = \lceil b_1 / fo \rceil = \lceil 45 / 68 \rceil = 1$ blocco $n_{accessi} = n_{livelli} + 1 = 3$	Dopo aver calcolato il numero di blocchi dell'indice secondario nb_i trovo il numero di blocchi per ogni livello: <ul style="list-style-type: none"> • $b_1 = nb_i = 442$ blocchi • $b_2 = \lceil b_1 / fo \rceil = \lceil 442 / 68 \rceil = 7$ blocchi • $b_3 = \lceil b_2 / fo \rceil = \lceil 7 / 68 \rceil = 1$ blocco $n_{accessi} = n_{livelli} + 1 = 4$

2. Indici multilivello dinamici (B-alberi)

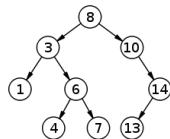
Gli indici multilivello dinamici (B-alberi e B+-alberi) sono casi speciali di strutture ad albero. Un albero è formato da **nodi**, dove, ognuno di essi ad eccezione di un nodo speciale detto **radice**, ha un **nodo padre** e zero, uno o più **nodi figli**. Se un nodo non ha nessun nodo figlio è detto **nodo foglia** (leaf node). Ogni nodo che non sia un nodo foglia è detto **nodo interno** (internal node). Un sottoalbero di un nodo comprende quel nodo e tutti i suoi discendenti (nodi figli, figli dei nodi figli, etc.)

BST: Binary Search Tree

Un Albero di ricerca binaria (BST) è un albero binario radicato, i cui nodi memorizzano una chiave e ciascuno di essi ha due **sottoalberi distinti**. L'albero dovrebbe soddisfare la proprietà BST:

$$\forall x \in T \wedge \forall y \in T_{x.left} \Rightarrow y.key < x.key \text{ e } \forall x \in T \wedge \forall y \in T_{x.right} \Rightarrow y.key > x.key$$

Tale proprietà afferma che **la chiave di ogni nodo** deve essere **maggiore** di tutte le chiavi memorizzate nel **sottoalbero di sinistra** e **non maggiore** di tutte le chiavi nel **sottoalbero di destra**.



B-alberi

Un B-tree è una struttura dati che permette la **rapida localizzazione dei file**, riducendo il numero di volte che un utente necessita per accedere alla memoria (secondaria) in cui il dato è salvato. Essi **derivano dai BST**, in quanto ogni chiave appartenente al sottoalbero sinistro di un nodo è di valore inferiore rispetto a ogni chiave appartenente ai sottoalberi alla sua destra. Inoltre, la loro struttura ne **garantisce il bilanciamento**: per ogni nodo, le altezze dei sottoalberi destro e sinistro differiscono al massimo di una unità.

Proprietà dei BTree

1. Ogni nodo interno del B Tree o ha la forma:

$$\langle P_1, \langle K_1, Pr_1 \rangle, P_2, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_q \rangle;$$

dove

- $q \leq p$;
- P_i è un **tree pointer** (puntatore ad un altro nodo del B-albero);
- K_i è il **search value**;
- Pr_i è un **data pointer** (puntatore ad un record con valore del campo chiave di ricerca uguale a K_i o al blocco che contiene tale record);

2. Per ogni nodo, tutti i valori sono ordinati:

$$K_1 < K_2 < \dots < K_{q-1};$$

3. Ogni nodo ha al più p tree pointer;
4. Per tutti i valori X della chiave di ricerca appartenenti al sottoalbero puntato da P_i , si ha che:
 - $K_{i-1} < X < K_i$ per $1 < i < q$;
 - $X < K_i$ per $i = 1$;
 - $K_{i-1} < X$ per $i = q$;
5. ogni nodo, esclusa la radice, ha almeno $\lceil p/2 \rceil$ tree pointer (la radice ha almeno due tree pointer, a meno che non sia l'unico nodo dell'albero);
6. un nodo con q tree pointer, con $q \leq p$, ha $q - 1$ campi chiave di ricerca (e $q - 1$ data pointer);
7. tutti i nodi foglia sono posti allo stesso livello (i nodi foglia hanno la stessa struttura dei nodi interni, ad eccezione del fatto che tutti i loro tree pointer P_i sono nulli).

Altezza e altezza massima

L'altezza **aumenta solo** quando viene fatto lo **split della radice**. Un BTree T di grado t con n chiavi ha **altezza**:

$$h \leq \log_t \frac{n+1}{2} \text{ (ossia } h \in O(\log_t \frac{n+1}{2})\text{)}.$$

Operazioni sui Btree

Dato che **la struttura dati risiede in memoria secondaria**, per poter lavorare con i nodi bisogna **spostarli in memoria principale**. Una volta fatto, tali nodi devono essere **ricaricati** in memoria secondaria. Per questa ragione ci sono **due tipi di operazioni** (bisogna sempre tenere **separati** i due costi, anche quando sono uguali):

- | | |
|--|-----------------------------------|
| - R/W dalla memoria secondaria | → costo $O(h) = (\log_t n)$. |
| - CPU (avvengono in memoria principale) | → costo $O(t h) = (t \log_t n)$. |

Obiettivo delle procedure è quello di cercare sempre di **limitare** il numero di operazioni R/W.

Operazioni di ricerca

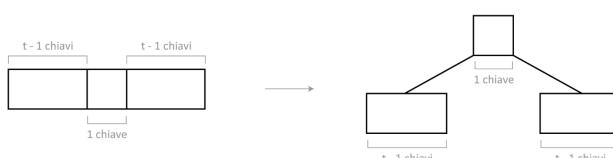
Dato un BTree T di grado t e dato k , si vuole cercare k in T : partendo dalla radice:

1. Scandisce il vettore (nodo). Se trova:
 - la chiave k : la procedura **termina** ritornando il nodo e la posizione dov'è contenuto k .
 - l maggiore k : **scende nel figlio sinistro** di l e **ripete** il ragionamento dal punto 1.
 - un valore **nil**: la procedura **termina** senza successo.

Il **numero di operazioni** di lettura/scrittura da disco dipende **solo** dall'**altezza** dell'albero. La ricerca della chiave minima e la ricerca della chiave massima hanno costi diversi in termini di operazioni di CPU e letture/scritture. Nel **caso migliore** il costo è $O(1)$ per le operazioni di CPU e $O(1)$ per le operazioni di R/W.

Operazioni di inserimento

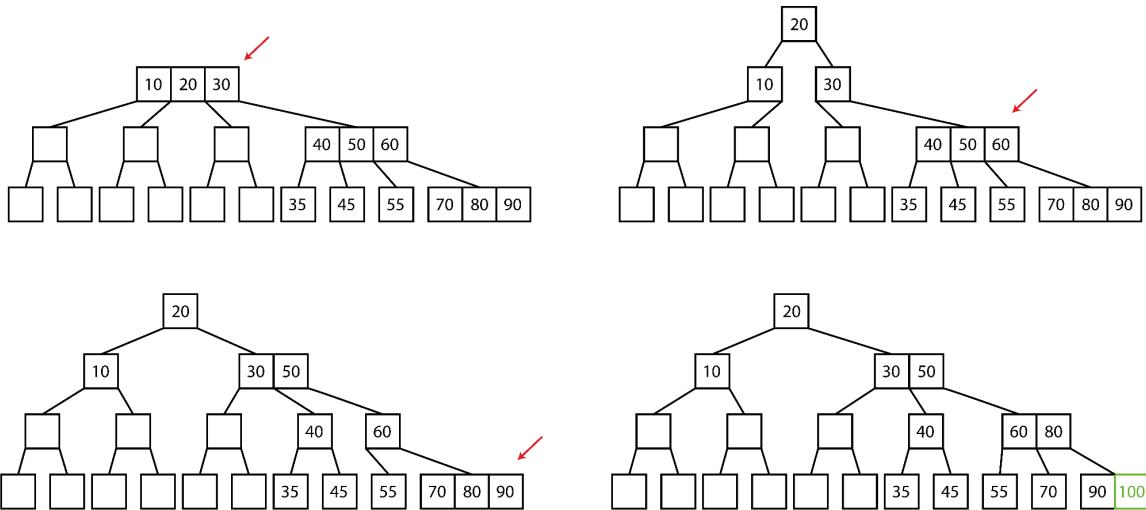
Per inserire k in T BTree di grado t , k viene inserito in una foglia che **esiste già** ma se tale foglia è **piena** (e quindi $x.n = 2t - 1$) allora si applica un'operazione di **splitting** che dato un nodo ne **genera due nuovi**:



Questa operazione viene fatta **ogni volta** che si incontra un nodo pieno **durante la discesa** (preliminare all'inserimento) dalla radice verso le foglie: Partendo dalla radice, si controlla se tale nodo è **pieno**, se lo è si **richiama la procedura di split**, se non lo è si **scende** al livello successivo e ci si **ripone la domanda**.

Esempio di inserimento

Inserire la chiave 100 nel seguente Btree:



L'inserimento di k in T ha un costo asintotico in termini di operazioni di CPU (ma non in termini di operazioni R/W) che può variare in base a k .

Operazioni di cancellazione

(non viene chiesta allo scritto) Se x si trova in una foglia...

1. con almeno t chiavi:
 - a. elimina x .
2. con esattamente $t-1$ chiavi e ha un fratello con almeno t chiavi:
 - a. porta su il valore più vino ad x in tale fratello,
 - b. abbassa un valore dal padre,
 - c. elimina x .

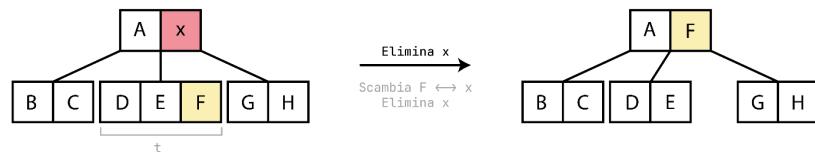


3. con esattamente $t-1$ chiavi e non ha fratelli con almeno t chiavi:
 - a. fonde x insieme a un dei suoi fratelli usando una chiave del padre come nesso,
 - b. elimina x .



Se x si trova in un nodo interno...

1. dove il figlio sinistro/destro di x ha almeno t chiavi:
 - a. scambia l'elemento maggiore/minore del figlio sinistro/destro di x al posto di x ,
 - b. elimina x .



2. dove nessun figlio di x ha almeno t chiavi:
 - a. elimina x ,
 - b. usa il minore (o il maggiore) tra le chiavi dei figli di x per chiudere il buco,
 - c. fa il merge tra i 2 figli portando giù quello che aveva sostituito x .



Il costo di queste operazioni è $O(t \log_t n)$ per operazioni di CPU e $O(\log_t n)$ per operazioni di R/W.

Esercizio esame

Si consideri un file contenente $r = 100000000$ record di dimensione prefissata pari a $R = 500$ byte, memorizzati in blocchi di dimensione pari a $B = 4096$ byte in modo unspanned. La dimensione del campo chiave primaria V sia $V = 14$ byte; la dimensione del puntatore a blocco P sia $P = 6$ byte.

Trovare il numero di accessi per una ricerca su **B-Albero** con campo di ricerca il campo chiave primaria V, puntatore ai dati di dimensione pari a $P_r = 7$ byte e puntatore ausiliario di 6 byte, assumendo che ciascun nodo sia pieno al 70%.

Determino l'ordine dei nodi risolvendo la disequazione:

$$p \cdot P + (p - 1)(V + P_r) \leq B \rightarrow 6p + (p - 1)(14 + 7) \leq 4096 \rightarrow 27p \leq 4117 \rightarrow p \leq 152$$

Trovo il numero di puntatori e il numero di valori assumendo che ciascun nodo sia pieno al 70%:

$$p \cdot 70\% = 152 \cdot 0.7 \approx 106 \text{ puntatori contenuti da ogni nodo e } 106 - 1 \text{ valori.}$$

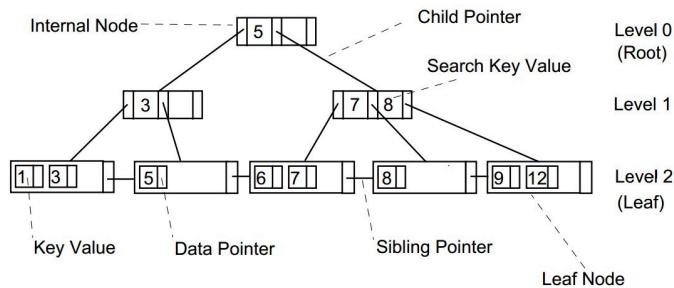
	Nodi $nodi_i = puntatori_{i-1}$	Valori $valori_i = nodi_i \cdot valori_{radice}$	Puntatori $puntatori_i = nodi_i \cdot puntatori_{radice}$
Radice	1	105	106
Livello 1	106	$106 \cdot 105 = 11130$	$106 \cdot 106 = 11236$
Livello 2	11236	$11236 \cdot 105 = 1179780$	$11234 \cdot 106 = 1191016$
Livello 3	1191016	$1191016 \cdot 105 = 125056680 \rightarrow$ maggiore rispetto a $r = 100000000$ record	
$n \text{ accessi} = n \text{ livelli} + 1 = 5 \text{ accessi}$			

B+ alberi

Un B+ albero è un B albero in cui i **data pointer** sono memorizzati solo nei **nodi foglia** dell'albero. La struttura dei nodi foglia differisce, quindi, da quella dei nodi interni. Se il campo di ricerca

- è un campo chiave: i nodi foglia hanno per ogni valore del campo di ricerca una entry e un puntatore ad un record.
- non è un campo chiave: i puntatori indirizzano un blocco che contiene i puntatori ai record del file di dati, rendendo così necessario un passo aggiuntivo per l'accesso ai dati.

I **nodi foglia** di un B+ albero sono generalmente messi fra loro in **relazione** (ciò viene sfruttato nel caso di range query). Essi corrispondono al primo livello di un indice. I nodi interni corrispondono agli altri livelli di un indice. Alcuni valori del campo di ricerca presenti nei nodi foglia sono ripetuti nei nodi interni per guidare la ricerca.



Proprietà dei BTree

1. Ogni nodo interno del B Tree o ha la forma:

$$\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle,$$

dove

- $q \leq p$;
- P_i è un **tree pointer** (puntatore ad un altro nodo del B-albero);
- K_i è il **search value**;

2. Per ogni nodo, tutti i valori sono ordinati:

$$K_1 < K_2 < \dots < K_{q-1};$$

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

4. Per tutti i valori X della chiave di ricerca appartenenti al sottoalbero puntato da P_i , si ha che:

- $K_{i-1} < X < K_i$ per $1 < i < q$;
- $X < K_i$ per $i = 1$;
- $K_{i-1} < X$ per $i = q$;

5. ogni nodo, esclusa la radice, ha almeno $[p/2]$ tree pointer (la radice ha almeno due tree pointer se è un nodo interno);

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

7. tutti i nodi foglia sono posti allo stesso livello;

La struttura dei nodi foglia

La struttura dei nodi foglia (di ordine p_{leaf}) è diversa:

1. ogni nodo foglia è della forma:

$$<K_1, Pr_1>, <K_2, Pr_2>, \dots, <K_q, Pr_q> P_{next},$$

dove

- $q \leq p_{leaf}$;
- Pr_i è un data pointer e P_{next} è un tree pointer punta al successivo nodo foglia del B+-albero;

2. per ogni nodo, si ha che

$$K_1 < K_2 < \dots < K_q$$

3. ogni Pr_i è un data pointer che punta al record con valore del campo di ricerca uguale a K_i o ad un blocco contenente tale record (o ad un blocco di puntatori ai record con valore del campo di ricerca uguale a K_i , se il campo di ricerca non è una chiave);

4. ogni nodo foglia ha almeno $[p_{leaf}/2]$ valori;

5. tutti i nodi foglia sono dello stesso livello;

Esercizio esame

Si consideri un file contenente $r = 100000000$ record di dimensione prefissata pari a $R = 500$ byte, memorizzati in blocchi di dimensione pari a $B = 4096$ byte in modo unspanned. La dimensione del campo chiave primaria V sia $V = 14$ byte; la dimensione del puntatore a blocco P sia $P = 6$ byte.

Trovare il numero di accessi per una ricerca su **B+-Albero** con campo di ricerca il campo chiave primaria V, puntatore ai dati di dimensione pari a $P_r = 7$ byte e puntatore ausiliario di 6 byte, assumendo che ciascun nodo sia pieno al 70%.

Determino l'ordine p dei nodi interni risolvendo la disequazione: $p \cdot P + (p - 1) \cdot V \leq B$

$$p \cdot P + (p - 1) \cdot V \leq B \rightarrow 6p + (p - 1) \cdot 14 \leq 4096 \rightarrow 20p \leq 4110 \rightarrow p \leq 205$$

Determino l'ordine p dei nodi foglia risolvendo la disequazione: $p \cdot (V + P_r) + P \leq B$

$$p \cdot (V + P_r) + P \leq B \rightarrow p \cdot (14 + 7) + 6 \leq 4096 \rightarrow 21p \leq 4090 \rightarrow p \leq 194$$

Trovo il numero di puntatori e il numero di valori assumendo che ciascun nodo sia pieno al 70%:

- nodi interni: $p_{nodi\ interni} \cdot 70\% = 205 \cdot 0.7 \approx 144$ puntatori nei nodi interni e 143 entry,
- nodi foglia: $p_{nodi\ foglia} \cdot 70\% = 194 \cdot 0.7 \approx 136$ puntatori nei nodi interni e 136 entry.

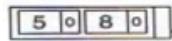
	Nodi $nodi_i = puntatori_{i-1}$	Valori $valori_i = nodi_i \cdot valori_{radice}$	Puntatori $puntatori_i = nodi_i \cdot puntatori_{radice}$
Radice	1	143	144
Livello 1	144	$144 \cdot 143 = 20592$	$144 \cdot 144 = 20736$
Livello 2	20736	$20736 \cdot 143 = 1179780$	$20736 \cdot 144 = 2985984$
Livello 3	2985984	$2985984 \cdot 136 = 406093824 \rightarrow$ maggiore rispetto a $r = 100000000$ record	
$n\ accessi = n\ livelli + 1 = 5\ accessi$			

Algoritmo di inserimento

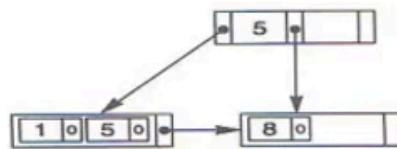
A differenza dell'algoritmo usato nei B alberi, qui si utilizza una procedura ***two pass***, ossia si tentano gli inserimenti scorrendo verso le foglie, ma si ignorano i nodi pieni. se si rende necessario, dopo l'inserimento, allora si effettuerà uno split.

inserire 8, 5, 1, 7, 3, 12, 9, 6 con $p = 3$ e $p_{leaf} = 2$

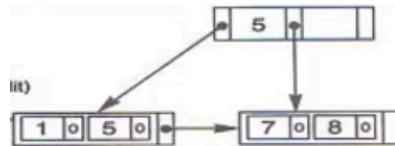
L'inserimento di 8 e 5 avviene senza problemi nel nodo radice, al momento anche foglia:



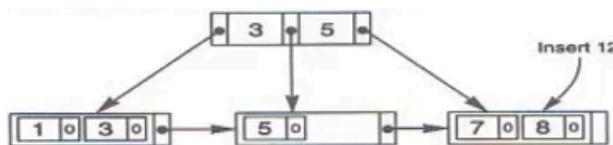
Per inserire 1, copio 5 e 8 in S, inserisco 1 in posizione corretta (prima del 8, ottenendo il nodo 1, 5, 8). Dopodichè sposto i primi $\lceil(p_{leaf} + 1)/2\rceil = \lceil(2 + 1)/2\rceil = 2$ valori (1, 5) nel primo figlio, i valori rimanenti (8) li metto nel secondo figlio e prendo il valore più grande tra quelli contenuti nel primo figlio e lo copio nel genitore. In questo caso il genitore non c'è quindi viene aggiunto in un nuovo nodo:



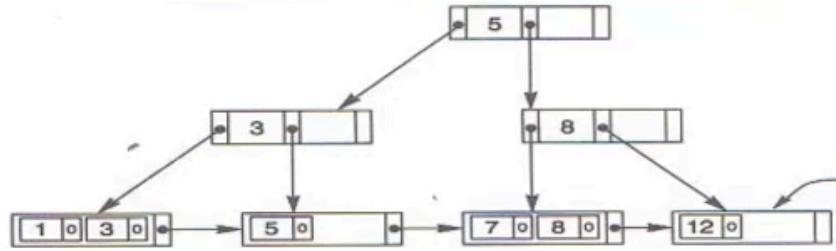
Ora, per inserire 7, analizzo la radice: $7 > 5$ dunque scendo a destra, in un nodo foglia che ha spazio, quindi aggiungo semplicemente il 7 al nodo foglia.



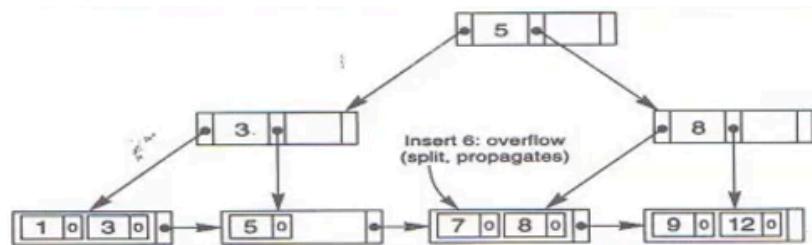
Per inserire 3, analizzo la radice: $3 < 5$, dunque scendo a sinistra. Qui trovo un nodo pieno, copio 1, 5 in S, inserisco 3 nella posizione corretta (1, 3, 5), sposto i primi $\lceil(p_{leaf} + 1)/2\rceil = \lceil(2 + 1)/2\rceil = 2$ valori (1, 3) nel primo figlio, i restanti valori (5) nel secondo figlio, prendo il valore più grande di quelli di sinistra (3) e lo copio nel genitore. La radice non è piena quindi colloco il tre nella posizione corretta:



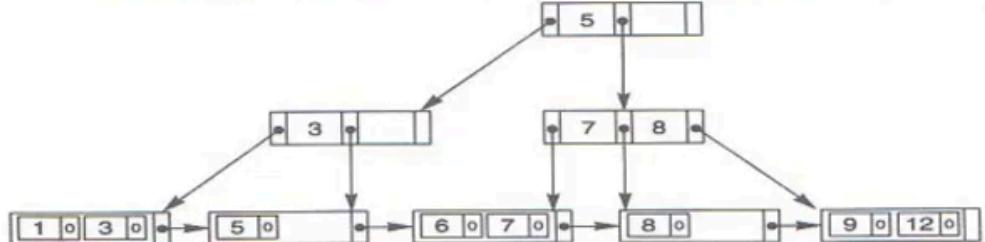
Per inserire 12, analizzo la radice: $12 > 5$: scendo nel suo figlio destro. Qui trovo un nodo pieno, copio 7, 8 in S, inserisco 12 nella posizione corretta (7, 8, 12), sposto i primi $\lceil(p_{leaf} + 1)/2\rceil = \lceil(2 + 1)/2\rceil = 2$ valori (7, 8) nel primo figlio, i restanti valori (12) nel secondo figlio, prendo il valore più grande di quelli di sinistra (8) e lo copio nel genitore. La radice è piena quindi ripeto il ragionamento (**occhio perché lo split in unodo interno richiede il pavimento e non il soffitto**): copio 3, 5 in S, inserisco 8 nella posizione corretta (3, 5, 8), sposto i primi $\lceil(p_{leaf} + 1)/2\rceil = \lceil(2 + 1)/2\rceil = 1$ valori (3) nel primo figlio, i restanti valori (5, 8) nel secondo figlio, prendo il valore più piccolo di quelli di destra (5) e lo copio nel genitore, che non esiste, quindi creo una nuova radice.



Per inserire 9 è semplice: si scorre nell'albero e lo si posiziona nella foglia che ha un buco libero:



Per inserire 6, analizzo la radice: $6 > 5$: scendo nel suo figlio destro e , dato che non è un nodo foglia, ripeto il ragionamento: $6 < 8$: dunque scendo a sinistra. Qui trovo un nodo pieno, copio 7, 8 in S, inserisco 6 nella posizione corretta (6, 7, 8), sposto i primi 2 valori (6, 7) nel primo figlio, i restanti valori (8) nel secondo figlio, prendo il valore più grande di quelli di sinistra (7) e lo copio nel genitore, che fortunatamente ha un buco libero:



B+ alberi vs B alberi

I nodi foglia di un B+-albero sono generalmente messi fra loro in relazione. Questo permette di gestire con maggior efficacia le **range query**: immaginiamo ad esempio, di aver come campo chiave di ricerca il numero di matricola degli studenti e di voler trovare tutti gli studenti che hanno matricola compresa tra un certo valore ed un altro. Per rispondere a questa richiesta, nel caso dei B+ alberi si può fare una point query sul primo valore richiesto e scorrere poi le foglie fino all'estremo di destra.

Per implementare questa range query nel caso di un B albero, sarebbe stato necessario salire e scendere ripetutamente lungo i cammini dell'albero, dato che diversi valori non si trovano nelle foglie, ma bensì nei nodi interni (così non vera nei B+ alberi).