



Università degli Studi di Verona
Dipartimento di Informatica
A.A. 2019-2020

RIASSUNTO DEL CORSO DI “BASI DI DATI”

Creato da: *Davide Zampieri*

Indice degli argomenti

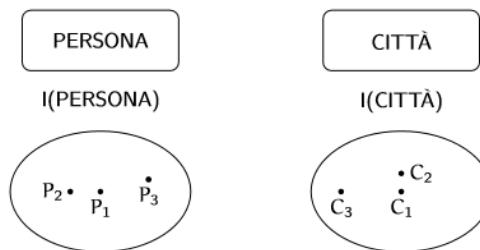
Costrutti del modello E-R	2
Modello relazionale.....	3
Algebra relazionale.....	6
SQL - Structured Query Language	8
DBMS e transazioni	12
Ottimizzazione delle interrogazioni.....	23
Interazione con le applicazioni e cenni ad altre tecnologie	27
XML e XML Schema	30

PRIMO SEMESTRE

❖ Costrutti del modello E-R

Entità.

Le entità rappresentano *classi di oggetti* che hanno *proprietà comuni*, *esistenza autonoma* e *identificazione univoca*. Le *istanze* sono oggetti appartenenti alla classe rappresentata dall'entità stessa.



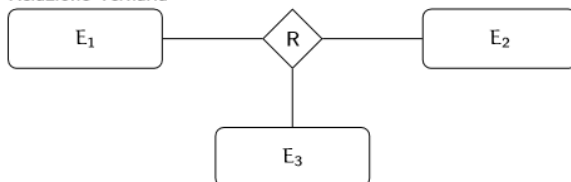
Relazione.

Le relazioni rappresentano *legami logici* tra due o più entità. Le *istanze* sono ennuple formate dalle istanze delle entità coinvolte nella relazione.

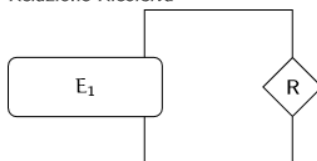
• Relazione Binaria



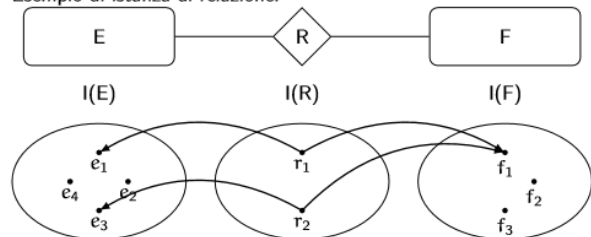
• Relazione Ternaria



• Relazione Ricorsiva

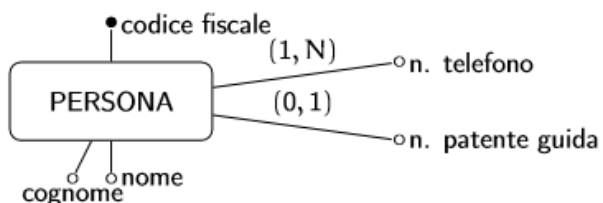


Esempio di istanza di relazione:



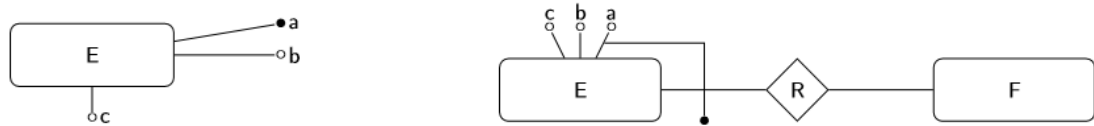
Attributo.

Gli attributi descrivono le *proprietà elementari* di entità o relazioni. Un attributo associa a ciascuna *istanza* di entità (o relazione) un valore appartenente ad un dominio (insieme di valori ammissibili). Gli attributi possono anche essere *opzionali* (0,1) o *multi-valore* (1,N). Inoltre, si possono raggruppare attributi con affinità di significato per formare un attributo *composto*.



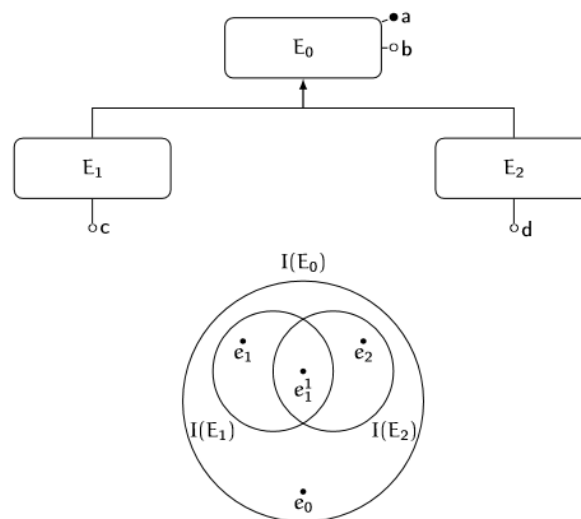
Identificatore.

L'identificatore rappresenta l'insieme di attributi e/o relazioni che permettono di *identificare univocamente* le istanze di un'entità. Si parla di *identificatore interno* quando esso contiene solo attributi. Si parla invece di *identificatore esterno* quando contiene anche relazioni; ciò è possibile solamente nel caso di cardinalità (1,1).



Generalizzazione.

Rappresenta un *legame logico* (simile ad una ereditarietà tra classi) che coinvolge un'entità *padre* e una o più entità *figlie*. Le istanze dei figli ereditano tutte le proprietà del padre e sono anche istanze dell'entità padre.



Una generalizzazione si dice:

- *Totale*, se ogni occorrenza dell'entità padre è anche occorrenza di almeno un'entità figlia (e₀ va tolta).
- *Parziale*, se non è totale (e₀ va lasciata).
- *Esclusiva*, se ogni occorrenza di un'entità è al più occorrenza di un'entità figlia (e₁₁ va tolta).
- *Sovrapposta*, se non è esclusiva (e₁₁ va lasciata).

❖ Modello relazionale

Tuple e relazioni.

Una *tupla* su un insieme di attributi X è una funzione t che associa a ciascun attributo A appartenente a X un valore del dominio $dom(A)$. Quindi, una *relazione* su X è un insieme di tuple su X .

Vincoli intra-relazionali.

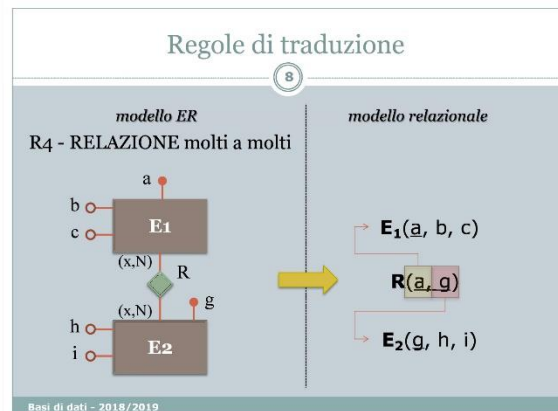
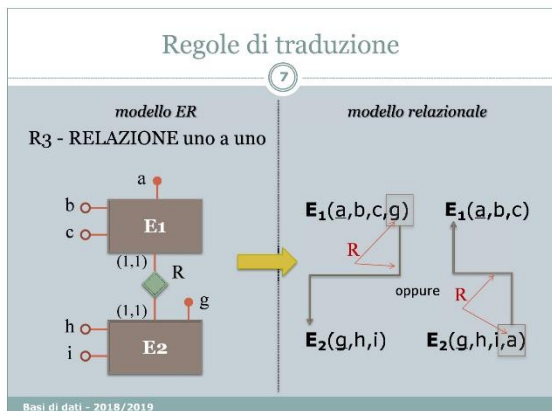
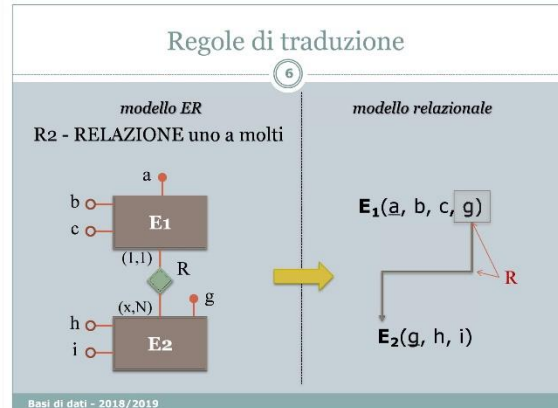
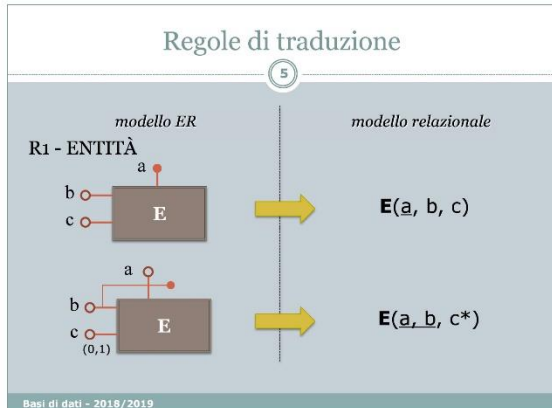
Una sottocategoria importante di tali vincoli sono i *vincoli di chiave*:

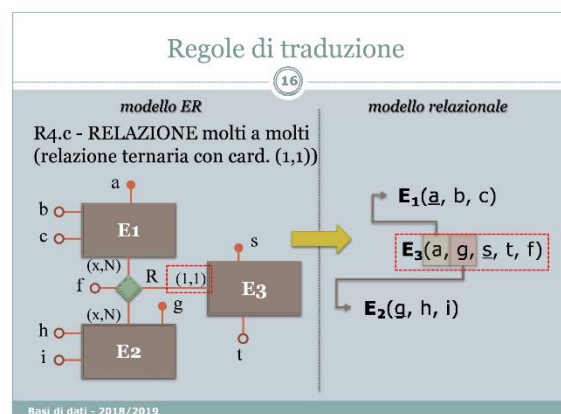
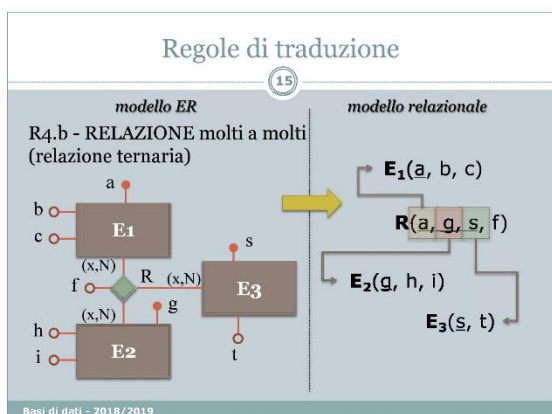
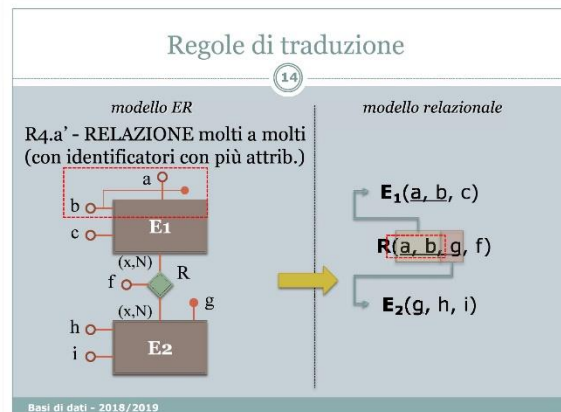
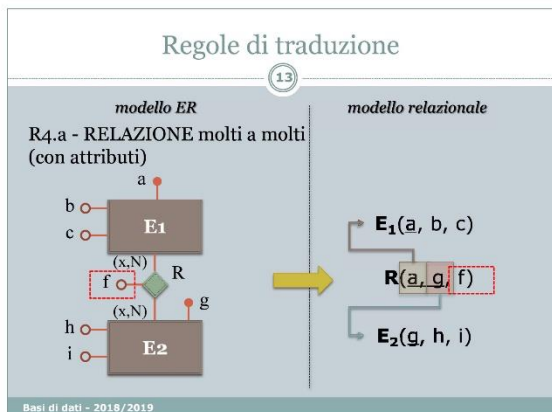
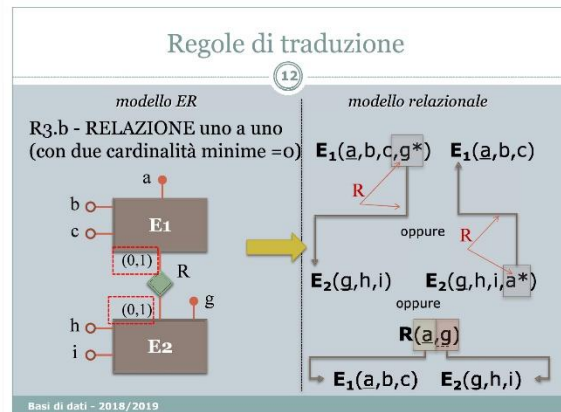
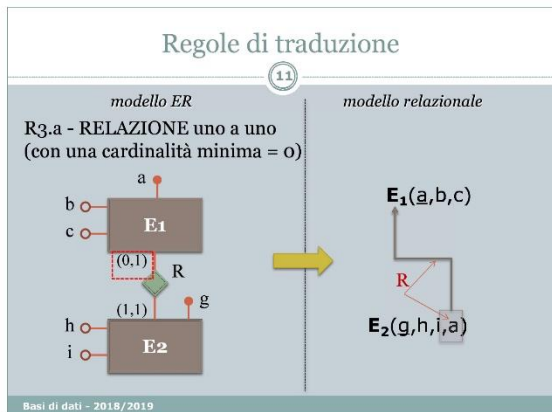
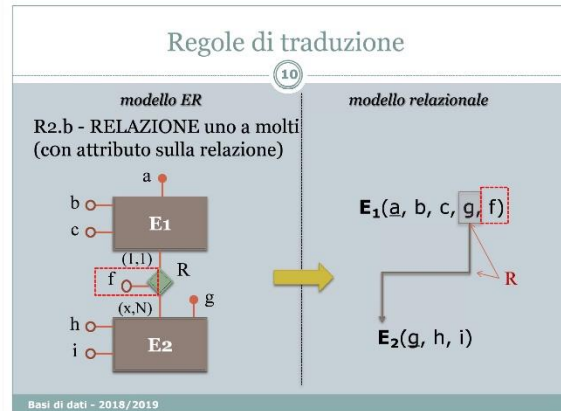
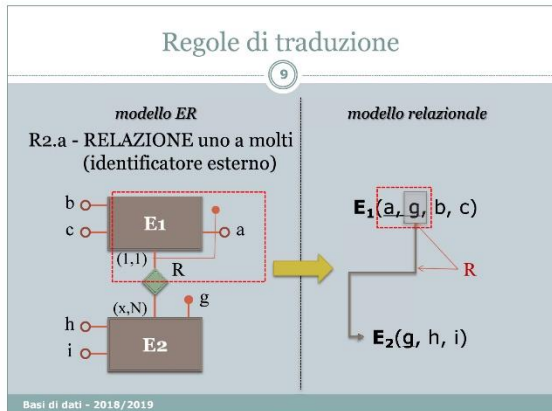
- **SUPERCHIAVE:**
Data una Relazione di schema $R(x)$, un insieme di attributi k , con $k \subseteq x$ è SUPERCHIAVE per $R(x)$ se per ogni istanza r di $R(x)$ vale la seguente condizione:
$$\forall t, t' \in r \quad t \neq t' \Rightarrow t[k] \neq t'[k]$$
- **CHIAVE oppure CHIAVE CANDIDATA:**
Data una Relazione di schema $R(x)$, un insieme di attributi k , con $k \subseteq x$ è CHIAVE CANDIDATA se è una SUPERCHIAVE per $R(x)$ e vale la seguente condizione:
$$\nexists k' \subseteq k : k' \text{ è SUPERCHIAVE per } R(x)$$
- **CHIAVE PRIMARIA:**
Data una Relazione di schema $R(x)$, un insieme di attributi k , con $k \subseteq x$ è CHIAVE PRIMARIA se è la CHIAVE CANDIDATA scelta per identificare univocamente le tuple delle istanze di $R(x)$

Per chiave candidata si intende un campo singolo o composto che soddisfa i requisiti di una chiave primaria. A dispetto di una chiave primaria, vi possono essere più chiavi candidate che competono per essere chiavi primarie.

Vincoli inter-relazionali.

Una sottocategoria importante di tali vincoli sono i *vincoli di integrità referenziale* (o vincoli sulle chiavi esportate):





❖ Algebra relazionale

Operatori insiemistici.

Si applicano solo a relazioni con lo stesso schema. Date due relazioni r_1 e r_2 di schema $R_1(X)$ e $R_2(X)$ si definiscono i seguenti operatori:

BASE

- **Unione:** $r_1 \cup r_2 = \{t \mid t \in r_1 \vee t \in r_2\}$
- **Differenza:** $r_1 - r_2 = \{t \mid t \in r_1 \wedge t \notin r_2\}$

DERIVATI

- **Intersezione :** $r_1 \cap r_2 = r_1 - (r_1 - r_2)$

Operatori specifici.

Si applicano a singole relazioni. Data una relazione r di schema $R(X)$ con $X = \{A_1, \dots, A_n\}$ si definiscono i seguenti operatori:

BASE

- **Ridenominazione:**
$$\rho_{A_1 A_2 \dots A_k \rightarrow B_1 B_2 \dots B_k}(r) = \{t \mid \exists t' \in r: \forall i \in \{1, \dots, k\}: t[B_i] = t'[A_i]\}$$
- **Selezione:**
$$\sigma_F(r) = \{t \mid t \in r \wedge F(t)\}$$
- **Proiezione**
$$\Pi_Y(r) = \{t \mid \exists t' \in r: \forall A_i \in Y: t[A_i] = t'[A_i]\}$$

In particolare:

- La *ridenominazione* consente di modificare lo schema di una relazione, infatti si passa da uno schema $X = \{A_1, \dots, A_n\}$ ad uno schema $Y = \{B_1, \dots, B_n\}$ con $|Y| = |X|$.
- La *selezione* consente di estrarre da una relazione le tuple che rendono vera la condizione F , ovvero una combinazione (attraverso \wedge, \vee, \neg) di $A \theta B$ e $A \theta c$ dove $\theta \in \{=, \neq, >, <, \geq, \leq\}$, $A, B \in X$, $c \in \text{dom}(A)$.
- La *proiezione* consente di eliminare alcuni attributi dalle tuple di una relazione, infatti si passa da uno schema $X = \{A_1, \dots, A_n\}$ ad uno schema $Y = \{A_1, \dots, A_m\}$ con $m \leq n$ e $Y \subseteq X$.

Infine, due osservazioni sulle cardinalità:

- La cardinalità della *selezione* va da 0 (se nessuna tupla rende vera F) a $|r|$ (se tutte le tuple rendono vera F).
- La cardinalità della *proiezione* va da 1 a $|r|$ (se Y è una superchiave).

Operatori di giunzione.

Si applicano a coppie di relazioni (con schema X_1 e X_2) allo scopo di unirle in un'unica relazione (con schema $X_1 \cup X_2$).

BASE

- **Join naturale** (su r_1 di schema X_1 e r_2 di schema X_2):
$$r_1 \bowtie r_2 = \{t \mid \exists t_1 \in r_1: \exists t_2 \in r_2: t_1 = t[X_1] \wedge t_2 = t[X_2]\}$$

DERIVATI

- **θ-Join** (su r_1 di schema X_1 e r_2 di schema X_2 con $X_1 \cap X_2 = \emptyset$)
$$r_1 \bowtie_F r_2 = \sigma_F(r_1 \bowtie r_2)$$

In particolare:

- Nel *join naturale* la condizione di join è implicita e dipende dallo schema delle relazioni coinvolte, infatti, due tuple costituiscono una coppia generata dal join naturale solo se presentano gli stessi valori negli attributi comuni tra le due relazioni.
- Nel θ -*join* la condizione di join è esplicitata come parametro F ; inoltre, un θ -join si dice *equi-join* se F è una congiunzione di uguaglianze tra attributi di r_1 e r_2 .

Infine, alcune osservazioni sulla cardinalità e sulle proprietà del join naturale:

- La *cardinalità* del join naturale va da 0 a $|r_1| \cdot |r_2|$ (se $X_1 \cap X_2 = \emptyset$ perché si producono tutte le possibili combinazioni), oppure va da 0 a $|r_1|$ (se $X_1 \cap X_2$ è superchiave per r_2), oppure è esattamente uguale a $|r_1|$ (se $X_1 \cap X_2$ è superchiave per r_2 ed è soggetto ad un vincolo di integrità referenziale).
- Il join naturale si dice *completo* se ogni tupla di r_1 e r_2 contribuisce a generare una tupla del risultato del join naturale (se una tupla non contribuisce si dice *dangling tuple*).
- Se il join naturale è completo, la sua *cardinalità* va da $\max(|r_1|, |r_2|)$ a $|r_1| \cdot |r_2|$.
- Il join naturale è *commutativo* e *associativo*.
- Se le due relazioni coinvolte nel join naturale hanno lo *stesso schema* ($X_1 = X_2$), allora esso si comporta come l'*intersezione*.
- Se le due relazioni coinvolte nel join naturale hanno *schemi disgiunti* ($X_1 \cap X_2 = \emptyset$), allora esso si comporta come il *prodotto cartesiano*.




Valori nulli.

Introducendo nell'algebra relazionale i valori nulli, alcune operazioni devono essere raffinate. In particolare, nella *selezione*:

- Se $t[A]$ o $t[B]$ sono *NULL*, allora $t[A] \theta t[B]$ è falso.
- Se $t[A]$ è *NULL*, allora $t[A] \theta c$ è falso.
- Si aggiungono le condizioni atomiche *A IS NULL* e *A IS NOT NULL*.

Nel *join naturale*, invece, la condizione di uguaglianza sugli attributi comuni è falsa se almeno uno di tali attributi comuni è *NULL*.

Inoltre, si possono definire i *join esterni* che consentono di ottenere nel risultato del join anche le *dangling tuples* di una o entrambe le relazioni coinvolte (estendendole con valori nulli):

- LEFT JOIN r_1  LEFT r_2
- RIGHT JOIN r_1  RIGHT r_2
- FULL JOIN r_1  FULL r_2

Ottimizzazione di espressioni.

L'*ottimizzazione* permette di generare un'espressione equivalente a quella di input ma di costo inferiore, ovvero esegue trasformazioni di equivalenza per ridurre la dimensione dei risultati intermedi. Prima di vedere le trasformazioni di equivalenza, definiamo le seguenti relazioni di *equivalenza tra espressioni algebriche*:

- **Equivalenza dipendente dallo schema:** dato uno schema R
 $E_1 \equiv_R E_2$ se $E_1(r) = E_2(r)$ per ogni istanza r di schema R
- **Equivalenza assoluta:** è indipendente dallo schema
 $E_1 \equiv E_2$ se $E_1 \equiv_R E_2$ per ogni schema R compatibile con E_1 e E_2

Trasformazioni di equivalenza.

Trasformazioni di equivalenza

12

Sia E un'espressione di schema X , si definiscono le seguenti trasformazioni di equivalenza:

- **Atomizzazione delle selezioni**

$$\sigma_{F_1 \wedge F_2}(E) \equiv \sigma_{F_1}(\sigma_{F_2}(E))$$
 È propedeutica ad altre trasformazioni. Non ottimizza se non è seguita da altre trasformazioni.
- **Idempotenza delle proiezioni**

$$\Pi_Y(E) \equiv \Pi_Y(\Pi_{YZ}(E)) \text{ dove } Z \subseteq X$$
 È propedeutica ad altre trasformazioni. Non ottimizza se non è seguita da altre trasformazioni.

Basi di dati - 2017/2018

Trasformazioni di equivalenza

13

Siano E_1 e E_2 espressioni di schema X_1 e X_2 , si definiscono le seguenti trasformazioni di equivalenza:

- **Anticipazione delle selezioni rispetto al join:**

$$\sigma_F(E_1 \bowtie E_2) \equiv E_1 \bowtie \sigma_F(E_2)$$
 Applicabile solo se F si riferisce solo ad attributi di E_2 .
- **Anticipazione della proiezione rispetto al join:**

$$\Pi_{X_1 Y}(E_1 \bowtie E_2) \equiv_R E_1 \bowtie \Pi_Y(E_2)$$
 Applicabile solo se $Y \subseteq X_2$ e $(X_2 - Y) \cap X_1 = \emptyset$

Basi di dati - 2017/2018

Trasformazioni di equivalenza

14

Combinando l'anticipazione della proiezione con l'idempotenza delle proiezioni otteniamo:

$$\Pi_Y(E_1 \bowtie_F E_2) \equiv \Pi_Y(\Pi_{Y_1}(E_1) \bowtie_F \Pi_{Y_2}(E_2))$$

$$\Pi_Y(E_1 \bowtie E_2) \equiv \Pi_Y(\Pi_{Y_1}(E_1) \bowtie \Pi_{Y_2}(E_2))$$

dove:

- $Y_1 = (X_1 \cap Y) \cup J_1$
- $Y_2 = (X_2 \cap Y) \cup J_2$
- $J_1/2$ sono gli attributi di $E_1/2$ coinvolti nel join (vale a dire presenti in F per il theta-join, mentre in caso di join naturale $J_1=J_2=X_1 \cap X_2$)

Basi di dati - 2017/2018

Ulteriori Trasformazioni di equivalenza

15

Siano E_1 e E_2 espressioni di schema X_1 e X_2 , si definiscono le seguenti trasformazioni di equivalenza:

- **Inglobamento di una selezione in un prodotto cartesiano (attenzione questa regola si applica solo dopo aver verificato che non sia possibile anticipare selezioni rispetto al join):**

$$\sigma_F(E_1 \bowtie E_2) \equiv E_1 \bowtie_F E_2$$
 dove $X_1 \cap X_2 = \emptyset$.

Basi di dati - 2017/2018

Ulteriori Trasformazioni di equivalenza

16

- Applicazione delle proprietà commutativa e associativa di: unione, prodotto cartesiano, intersezione.
- Applicazione della proprietà distributiva:

$$\sigma_F(E_1 \cup E_2) \equiv \sigma_F(E_1) \cup \sigma_F(E_2)$$

$$\sigma_F(E_1 - E_2) \equiv \sigma_F(E_1) - \sigma_F(E_2)$$

$$\Pi_Y(E_1 \cup E_2) \equiv \Pi_Y(E_1) \cup \Pi_Y(E_2)$$

$$E_1 \bowtie (E_2 \cup E_3) \equiv (E_1 \bowtie E_2) \cup (E_1 \bowtie E_3)$$

Basi di dati - 2017/2018

Ulteriori Trasformazioni di equivalenza

17

- Applicazione di altre trasformazioni:

$$\sigma_{F_1 \vee F_2}(E) \equiv \sigma_{F_1}(E) \cup \sigma_{F_2}(E)$$

$$\sigma_{F_1 \wedge F_2}(E) \equiv \sigma_{F_1}(E) \cap \sigma_{F_2}(E) \equiv \sigma_{F_1}(E) \bowtie \sigma_{F_2}(E)$$

$$\sigma_{F_1 \wedge \neg F_2}(E) \equiv \sigma_{F_1}(E) - \sigma_{F_2}(E)$$

Basi di dati - 2017/2018

❖ SQL - Structured Query Language

Introduzione.

È un linguaggio di interrogazione *dichiarativo* (precisa le proprietà che deve avere il risultato) che si basa sul *calcolo relazionale*.

Vista.

La vista è una *relazione derivata*, in quanto si specifica l'espressione che genera il suo contenuto che dipende quindi dalle relazioni che compaiono nell'espressione. Una vista si dice *virtuale* se viene calcolata ogni volta che serve. Una vista si dice *materializzata* se viene calcolata e memorizzata esplicitamente nella base di dati.

Forma base di una query.

```
SELECT <ListaAttributi>
      FROM <ListaTabelle>
      [WHERE <ListaCondizioni>]
```

Lo *schema* risultante di una query è costituito dagli attributi indicati in <ListaAttributi>. Il suo *contenuto* è costituito invece dalle tuple ottenute proiettando su <ListaAttributi> le tuple di <ListaTabelle> che soddisfano le eventuali condizioni indicate in <ListaCondizioni>.

Clausola **SELECT**.

<ListaAttributi> è una lista di espressioni con la seguente sintassi

```
< [DISTINCT] <espr> [[AS] <alias>] {, <espr> [[AS] <alias>]} | * >
```

dove:

- *DISTINCT* serve per eliminare i duplicati nella relazione risultato (se ho una superchiave non serve).
- <espr> è un'espressione che coinvolge attributi.
- <alias> è il nome assegnato all'attributo che conterrà il risultato dell'espressione.
- * indica che prendo tutti gli attributi.

Clausola **FROM**.

<ListaTabelle> è una lista di tabelle con la seguente sintassi

```
<tabella> [[AS] <alias>] {, <tabella> [[AS] <alias>]}
```

dove:

- Se ci sono più tabelle, si genera il *prodotto cartesiano* (non viene eseguito alcun join naturale).
- Se ci sono attributi con lo stesso nome in tabelle diverse, questi si denotano con <NomeTabella>.<NomeAttributo> (non c'è dipendenza dallo schema).

Clausola **WHERE**.

<ListaCondizioni> è un'espressione booleana ottenuta combinando condizioni semplici <cond> con i connettivi logici *AND*, *OR* e *NOT*. In particolare <cond> ha la seguente sintassi

```
< <espr> θ <espr> | <espr> θ <const> >
```

dove:

- $\theta \in \{=, <, >, \leq, \geq\}$.
- <const> è un valore compatibile con il tipo di <espr>.

Variabili tupla.

Le *variabili tupla* (o alias di tabella) vengono usate per risolvere l'ambiguità sui nomi degli attributi e per gestire il riferimento a più esemplari della stessa tabella.

Clausola **ORDER BY**.

Permette di ordinare le tuple della relazione risultato ed ha la seguente sintassi

```
ORDER BY <attr> [ASC | DESC] {, <attr> [ASC | DESC]}
```

dove:

- *ASC* e *DESC* indicano se l'ordinamento va fatto in modo crescente o decrescente (il default è *ASC*).
- L'ordine di comparizione degli <attr> determina l'ordine delle operazioni di ordinamento.

Interrogazioni nidificate.

Si ottengono quando nella clausola *WHERE* compare un predicato complesso, vale a dire un predicato che contiene un'altra interrogazione SQL (tipicamente mono-attributo). La sintassi è la seguente

WHERE <espr> θ' (SELECT ... FROM ... WHERE ...)

dove θ' è un *predicato complesso* avente una di queste forme:

- A θ *ANY*, che è soddisfatto dalla tupla t se esiste almeno un valore v contenuto nel risultato della query nidificata che verifica la condizione $t[A] \theta v$.
- A θ *ALL*, che è soddisfatto dalla tupla t se per ogni valore v contenuto nel risultato della query nidificata è verificata la condizione $t[A] \theta v$.

Si ricorda inoltre che \neq *ANY* si può scrivere *IN* e \neq *ALL* si può scrivere *NOT IN*.

Classificazione delle interrogazioni nidificate.

Un'interrogazione interna si dice *indipendente* (rispetto all'interrogazione esterna che la contiene) se viene valutata una sola volta, in quanto non dipende dalla tupla corrente dell'interrogazione esterna. L'indipendenza consiste nel fatto che non ci sono variabili tupla condivise tra l'interrogazione interna ed esterna. Un'interrogazione interna si dice invece *dipendente* (rispetto all'interrogazione esterna che la contiene) se viene valutata ogni volta, in quanto condivide almeno una variabile tupla con l'interrogazione esterna (passaggio di binding).

Clausola EXISTS.

È una clausola utilizzabile nei predicati complessi avente la seguente sintassi:

WHERE <espr> AND EXISTS(q)

Questa clausola ritorna *true* se q produce almeno una tupla. Inoltre, è efficace se viene applicata con passaggio di binding, vale a dire se q è un'interrogazione *dipendente* dall'interrogazione esterna.

Operatori aggregati.

Si specificano nella clausola *SELECT* e agiscono su uno o più attributi delle tuple risultato. Esistono 5 operatori aggregati standard, la cui sintassi è la seguente

COUNT (< * | [DISTINCT | ALL] <ListaAttributi> >)
<SUM | MAX | MIN | AVG> ([DISTINCT | ALL] <espr>)

dove:

- *COUNT(*)* restituisce il numero di tuple contenute nel risultato dell'interrogazione SQL.
- *ALL* considera le combinazioni dei valori degli attributi contenute nelle tuple risultato e restituisce il numero di combinazioni che non contengono valori nulli.
- *DISTINCT* è come *ALL* ma restituisce il numero di combinazioni distinte (eliminando i duplicati).

Si ricorda inoltre che se la clausola *SELECT* contiene operatori aggregati, non può contenere attributi singoli o espressioni su attributi.

Clausola GROUP BY.

Permette l'applicazione di un operatore aggregato distintamente a sottoinsiemi di tuple (gruppi) della relazione risultato di una interrogazione SQL. La suddivisione viene eseguita *raggruppando* insieme tutte le tuple che presentano gli stessi valori per l'insieme di attributi assegnato attraverso la clausola. La sua sintassi è la seguente

GROUP BY <attr> {, <attr>}

Si ricorda inoltre che <ListaAttributi> della clausola *SELECT* può contenere solo operatori aggregati applicati a espressioni oppure attributi indicati nella clausola *GROUP BY*.

Clausola *HAVING*.

Permette di applicare una condizione per selezionare parte dei gruppi, in quanto non è possibile usare operatori aggregati nella clausola *WHERE*. La sintassi è la seguente

HAVING <condizione_sel_gruppi>

dove <condizione_sel_gruppi> è un'espressione booleana in cui le formule atomiche sono del tipo:

- *OperatoreAggregato(<espr>) θ <const>*.
- *OperatoreAggregato(<espr1>) θ OperatoreAggregato(<espr2>)*.

Interrogazioni insiemistiche.

Consentono di eseguire operazioni insiemistiche sul risultato di due o più interrogazioni SQL. Sono disponibili le operazioni di *unione*, *intersezione* e *differenza* con la seguente sintassi

q {<UNION | INTERSECT | EXCEPT> [all] q}

dove:

- Se viene precisata la parola chiave *all*, allora non vengono eliminati i duplicati.
- Le relazioni risultato coinvolte devono avere lo stesso numero di attributi e attributi di tipo compatibile.

Viste.

La creazione di *viste* in SQL è usata per salvare interrogazioni complesse. La sintassi è la seguente

CREATE VIEW <nome_vista> [(<ListaAttributi>)] AS q

Si ricorda inoltre che:

- Non è possibile definire viste ricorsive.
- È possibile definire una vista utilizzando altre viste, ma evitando dipendenze circolari.
- Le viste non sono in generale aggiornabili.

SECONDO SEMESTRE

❖ DBMS e transazioni

Transazione.

Un DBMS (sistema per la gestione di basi di dati) basato sul modello relazionale è nella maggior parte dei casi anche un sistema *transazionale*, ovvero fornisce un meccanismo per la definizione ed esecuzione di transazioni. Una *transazione* è un'unità di lavoro svolto da un programma applicativo per la quale si vogliono garantire alcune proprietà. La principale caratteristica è che essa non ammette esecuzioni parziali, ovvero o va a buon fine causando effetti sulla base di dati oppure abortisce e non ha nessun effetto sulla base di dati.

Una transazione è *ben formata* se:

- Inizia con un `begin transaction`.
- Termina con un `end transaction`.
- La sua esecuzione comporta il raggiungimento di un `commit work` (se va a buon fine) o di un `rollback work` (se non ha effetto).

Proprietà delle transazioni (ACID properties).

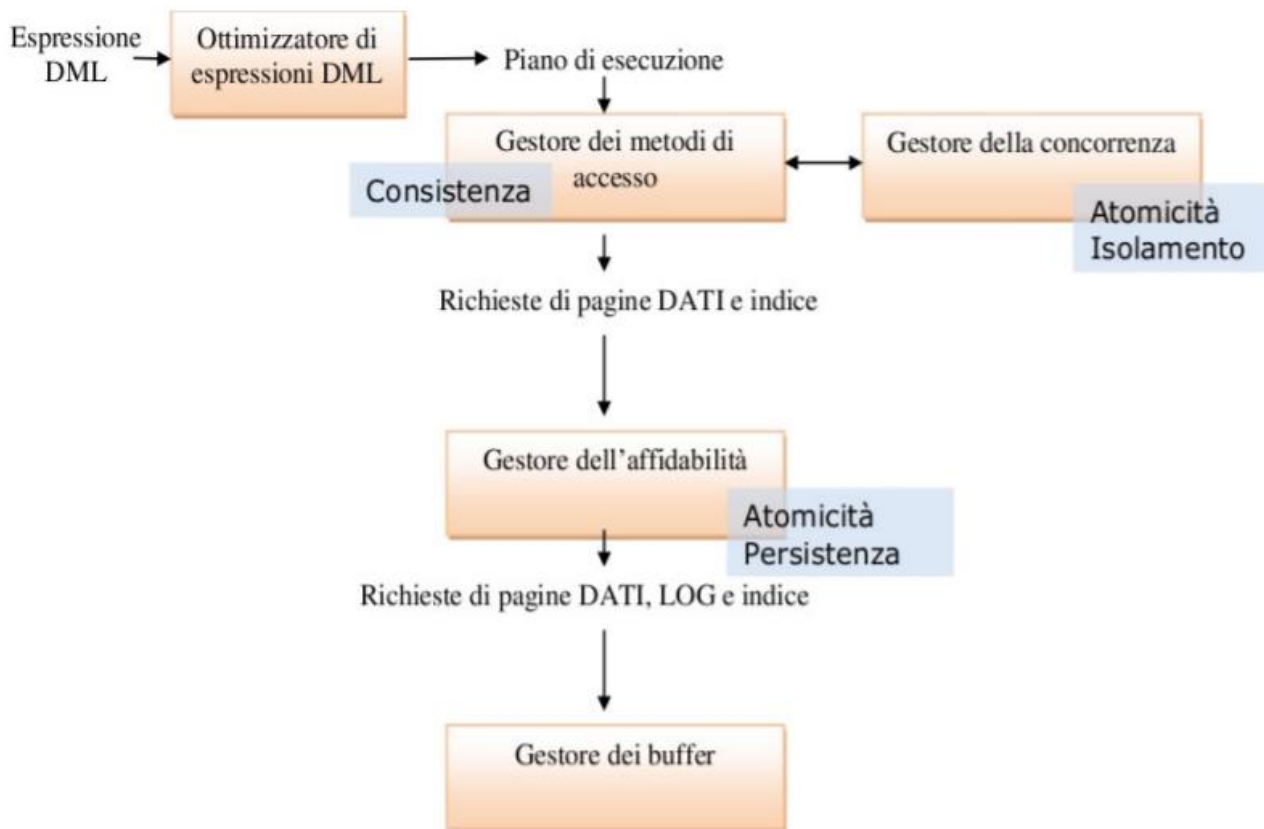
Un DBMS che gestisce transazioni dovrebbe garantire, per ogni transazione che esegue, le seguenti proprietà:

- **Atomicità (Atomicity)** – una transazione è una unità di esecuzione indivisibile (o viene eseguita completamente oppure non viene eseguita affatto); questo implica che, se una transazione viene interrotta prima del commit, il lavoro fin lì eseguito deve essere disfatto ripristinando la situazione in cui si trovava la base di dati prima dell'inizio della transazione e che, se una transazione viene interrotta all'esecuzione del commit (commit eseguito con successo), il sistema deve assicurare che la transazione abbia effetto sulla base di dati.
- **Consistenza (Consistency)** – l'esecuzione di una transazione non deve violare i vincoli di integrità; questo implica che, al verificarsi della violazione di un vincolo, il sistema può o reagire alla violazione abortendo l'ultima operazione e restituendo all'applicazione una segnalazione d'errore (verifica immediata) oppure abortire la transazione senza possibilità da parte dell'applicazione di reagire alla violazione (verifica differita).
- **Isolamento (Isolation)** – l'esecuzione di una transazione deve essere indipendente dalla contemporanea esecuzione di altre transazioni; questo implica che il rollback di una transazione non deve creare rollback a catena di altre transazioni che si trovano in una situazione di esecuzione concorrente (regolata dal sistema con appositi meccanismi di controllo dell'accesso alle risorse).
- **Persistenza (Durability)** – l'effetto di una transazione che ha eseguito il commit non deve andare perso; questo implica che il sistema deve essere in grado, in caso di guasto, di garantire gli effetti delle transazioni che al momento del guasto avevano già eseguito un commit.

Architettura di un DBMS.

L'architettura di un DBMS è composta da diversi moduli che rappresentano le operazioni svolte durante l'esecuzione delle transazioni. Tra i *moduli principali* troviamo:

- *Gestore dei metodi d'accesso*, che garantisce la consistenza.
- *Gestore della concorrenza*, che garantisce atomicità e isolamento.
- *Gestore dell'affidabilità*, che garantisce atomicità e persistenza.

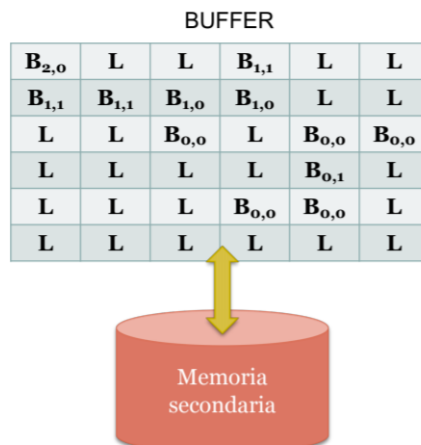


DBMS e memoria secondaria.

Le basi di dati gestite da un DBMS risiedono in *memoria secondaria* perché sono *grandi e persistenti*. Nella memoria secondaria i dati sono organizzati in blocchi (o pagine) e le uniche operazioni possibili sono la lettura e la scrittura di un intero blocco. I costi di tali operazioni sono di *ordini di grandezza maggiori* del costo per accedere ai dati in memoria centrale.

Buffer.

L'interazione tra memoria secondaria e memoria centrale avviene attraverso il *trasferimento di pagine* della memoria secondaria in una zona appositamente dedicata della memoria centrale (condivisa dalle applicazioni) detta *buffer*. Quando uno stesso dato viene utilizzato più volte in tempi ravvicinati, il buffer evita l'accesso alla memoria secondaria ed è quindi strategico per ottenere buone prestazioni nell'accesso ai dati.



$B_{i,j}$ indica che nella pagina del buffer è caricato il blocco B_i , inoltre "i" indica che il blocco è attualmente utilizzato da i transazioni mentre "j" è 1 se il blocco è stato modificato e 0 altrimenti. L indica pagina libera.

Gestore dei buffer.

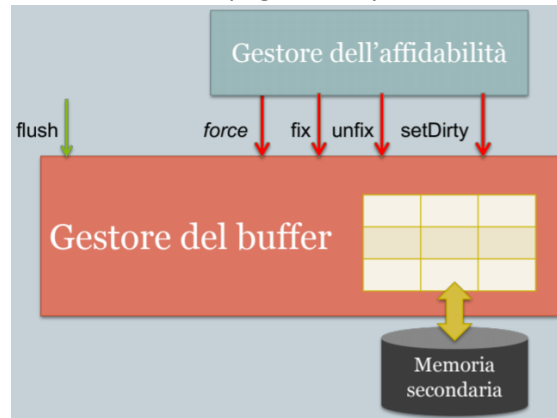
Il gestore dei buffer si occupa del caricamento/salvataggio delle pagine in memoria secondaria con l'obiettivo di aumentare la velocità di accesso ai dati. Tale comportamento del gestore dei buffer si basa sul *principio di località*, il quale prevede l'implementazione delle seguenti politiche (a seconda che si richieda una lettura o una scrittura):

- *Richiesta di lettura di un blocco* – se il blocco è presente in una pagina del buffer allora si restituisce un puntatore a quella pagina (non si esegue alcuna lettura su memoria secondaria), altrimenti si cerca una pagina libera in cui caricare il blocco e si restituisce il puntatore a tale pagina.
- *Richiesta di scrittura di un blocco* precedentemente caricato in una pagina del buffer – il gestore dei buffer può decidere di differire la scrittura su memoria secondaria in un altro momento.

Gestione delle pagine del buffer.

Per gestire le pagine del buffer si usano le seguenti *primitive*:

- *fix*, che viene usata dalle transazioni per richiedere l'accesso ad un blocco e restituisce al chiamante un puntatore alla pagina contenente il blocco richiesto.
- *unfix*, che viene usata dalle transazioni per indicare che la transazione ha terminato di usare il blocco (l'effetto è il decremento del contatore i che indica l'uso della pagina).
- *setDirty*, che viene usata dalle transazioni per indicare al gestore dei buffer che il blocco della pagina è stato modificato (l'effetto è la modifica del bit di stato j a 1).
- *force*, che viene usata dal gestore dell'affidabilità per salvare in memoria secondaria in modo sincrono il blocco contenuto nella pagina (l'effetto è il salvataggio in memoria secondaria del blocco e la modifica del bit di stato j a 0).
- *flush*, che viene usata dal gestore dei buffer per salvare blocchi sulla memoria secondaria in modo asincrono (l'effetto è la liberazione delle pagine "dirty" e la modifica del bit di stato j a 0).



Gestore dell'affidabilità.

È il modulo responsabile di ciò che riguarda l'esecuzione delle istruzioni per la gestione delle transazioni e la realizzazione delle operazioni necessarie al *ripristino* della base di dati dopo eventuali malfunzionamenti. Per il suo funzionamento, il gestore dell'affidabilità deve disporre di un dispositivo di *memoria stabile* (ovvero una memoria resistente ai guasti) in cui viene memorizzato il *file di log*, il quale registra in modo sequenziale le operazioni eseguite dalle transazioni sulla base di dati. I *record di transazione* memorizzati nel file di log riguardano:

- *Begin* della transazione T , indicato con il record $B(T)$.
- *Commit* della transazione T , indicato con il record $C(T)$.
- *Abort* della transazione T , indicato con il record $A(T)$.
- *Insert*, *Delete* e *Update* eseguiti dalla transazione T sull'oggetto O , indicati dai record $I(T, O, AS)$, $D(T, O, BS)$ e $U(T, O, BS, AS)$ dove AS indica After State e BS indica Before State.

Gestione del file di log.

I record di transazione salvati nel log consentono di eseguire, in caso di ripristino, le seguenti operazioni:

- *UNDO*, per cui se si vuole disfare un'azione su un oggetto O è sufficiente ricopiare in O il valore BS (l'Insert/Delete viene disfatto cancellando/inserendo O); questa operazione implica che i record di log debbano essere scritti prima dell'esecuzione delle corrispondenti operazioni sulla base di dati (WAL, Write Ahead Log).
- *REDO*, per cui se si vuole rifare un'azione su un oggetto O è sufficiente ricopiare in O il valore AS (l'Insert/Delete viene rifatto inserendo/cancellando O); questa operazione implica che i record di log debbano essere scritti prima dell'esecuzione del commit della transazione (*Commit-Precedenza*).

I record memorizzati nel file di log possono derivare da operazioni di:

- *Dump* (copia completa) della base di dati, indicato con il record DUMP.
- *Check Point* periodico, indicato con il record CK(T₁, . . . , T_n) dove T₁,...,T_n sono le transazioni che erano attive all'esecuzione del Check Point e che vengono quindi sospese fino alla scrittura sul log del record CK.

Una transazione T sceglie in *modo atomico* l'esito del commit nel momento in cui scrive nel file di log, in modo sincrono (primitiva force), il suo record di commit C(T).

Operazioni di ripristino.

In base al tipo di guasto, possiamo individuare due diverse operazioni di ripristino:

- *Ripresa a caldo*, da attuare dopo un *guasto di sistema* (ossia la perdita del contenuto della *memoria centrale*).
- *Ripresa a freddo*, da attuare dopo un *guasto di dispositivo* (ossia la perdita di una parte o di tutto il contenuto della base di dati in *memoria secondaria*).

Passi della ripresa a caldo.

1. Si accede all'ultimo blocco del log e si ripercorre all'indietro il log fino al *più recente record CK*.
2. Si decidono le transazioni da rifare/disfare iniziando l'*insieme UNDO* con le transazioni attive al momento del CK e l'*insieme REDO* con l'insieme vuoto.
3. Si ripercorre *in avanti* il log; per ogni record B(T) incontrato si aggiunge T all'insieme UNDO e per ogni record C(T) incontrato si sposta T dall'insieme UNDO all'insieme REDO.
4. Si ripercorre *all'indietro* il log, disfacendo le operazioni eseguite dalle transazioni nell'insieme UNDO risalendo fino alla prima azione della transazione più vecchia.
5. Si *rifanno* le operazioni delle transazioni nell'insieme REDO.

Passi della ripresa a freddo.

1. Si accede al *Dump* della base di dati e si copia selettivamente la parte deteriorata della base di dati.
2. Si accede al *log* risalendo al record di Dump.
3. Si ripercorre *in avanti* il log rieseguendo tutte le operazioni relative alla parte deteriorata (comprese le azioni di commit e abort).
4. Si applica una *ripresa a caldo*.

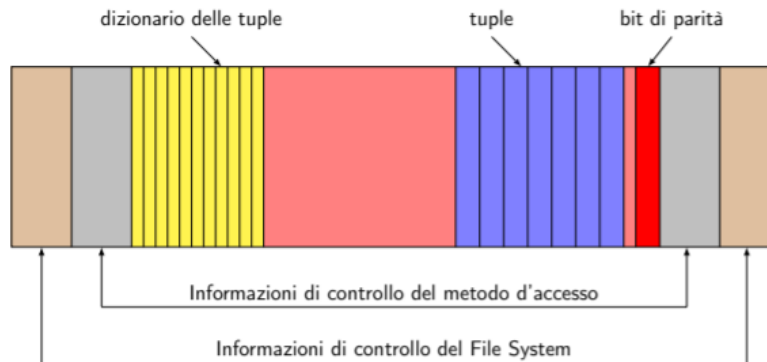
Gestore dei metodi di accesso.

È il modulo del DBMS che esegue il piano di esecuzione prodotto dall'ottimizzatore e produce sequenze di accessi ai blocchi della base di dati presenti in memoria secondaria. In particolare, i *metodi di accesso* sono i moduli software che implementano gli algoritmi di manipolazione dei dati (ad es. scansione sequenziale, accesso via indice, ordinamento, varie implementazioni del join). Ogni metodo di accesso ai dati conosce l'*organizzazione delle tuple* (o dei record di un indice) nei blocchi salvati in memoria secondaria e l'*organizzazione fisica interna dei blocchi* che contengono le tuple (o i record di un indice).

Organizzazione di un blocco dati.

In un blocco dati sono presenti:

- *Informazioni utili*, ovvero le tuple della tabella.
- *Informazioni di controllo*, come ad esempio il dizionario delle tuple, il bit di parità e altre informazioni del file system o della specifica struttura fisica.



Struttura del dizionario.

Se le tuple hanno *lunghezza fissa*, il dizionario non è necessario e quindi si deve solo memorizzare la dimensione delle tuple e l'offset del punto iniziale. Invece, se le tuple hanno *lunghezza variabile*, il dizionario deve memorizzare l'offset di ogni tupla presente nel blocco e di ogni attributo di ogni tupla. La lunghezza massima di una tupla corrisponde alla dimensione massima dell'area disponibile su un blocco. Tuttavia, si può anche gestire il caso di tuple memorizzate su più blocchi.

Operazioni sul blocco dati.

Per l'*inserimento* di una tupla, se esiste spazio contiguo sufficiente è molto semplice; se invece non esiste spazio contiguo ma esiste spazio sufficiente, bisogna riorganizzare lo spazio per poter eseguire un inserimento semplice; infine, se non esiste spazio sufficiente, l'operazione viene rifiutata. La *cancellazione* è sempre possibile, anche senza riorganizzare.

File sequenziale.

Il *file sequenziale* è una struttura fisica di rappresentazione dei dati la cui caratteristica fondamentale è che le tuple sono ordinate secondo una chiave di ordinamento. Per l'*inserimento* di una tupla bisogna:

- Individuare il blocco che contiene la tupla che precede, nell'ordine della chiave, la tupla da inserire.
- Inserire la nuova tupla nel blocco individuato ed eventualmente, se l'operazione non dovesse andare a buon fine, aggiungere un nuovo blocco alla struttura (overflow page) per inserirvi la nuova tupla.
- Aggiustare la catena dei puntatori.

Per la *cancellazione* di una tupla bisogna invece:

- Individuare il blocco che contiene la tupla da cancellare.
- Cancellare la tupla dal blocco individuato.
- Aggiustare la catena dei puntatori.

Indici.

Gli *indici* sono strutture di accesso ai dati ausiliarie utilizzate per aumentare le prestazioni degli accessi alle tuple memorizzate nelle strutture fisiche. Tali strutture velocizzano l'accesso casuale via *chiave di ricerca*, ossia l'insieme di attributi utilizzati dall'indice nella ricerca. Ci sono due tipi di indici su file sequenziali:

- *Indice primario*, in cui la chiave di ordinamento del file sequenziale coincide con la chiave di ricerca dell'indice.
- *Indice secondario*, in cui la chiave di ordinamento e la chiave di ricerca sono diverse.

Indice primario.

Usa una chiave di ricerca che coincide con la chiave di ordinamento del file sequenziale. Ogni *record* dell'indice primario contiene una coppia $\langle v_i, p_i \rangle$ dove v_i è il valore della chiave di ricerca e p_i è il puntatore al primo record nel file sequenziale con chiave v_i . Esistono due varianti dell'indice primario:

- *Indice denso*, in cui per ogni occorrenza della chiave presente nel file esiste un corrispondente record nell'indice.
- *Indice sparso*, in cui solo per alcune occorrenze delle chiavi presenti nel file esiste un corrispondente record nell'indice (tipicamente una per blocco).

Indice secondario.

Usa una chiave di ricerca che non coincide con la chiave di ordinamento del file sequenziale. Ogni *record* dell'indice secondario contiene una coppia $\langle v_i, p_i \rangle$ dove v_i è il valore della chiave di ricerca e p_i è il puntatore al bucket di puntatori che individuano nel file sequenziale tutte le tuple con valore di chiave v_i . Gli indici secondari sono *sempre densi*.

INDICE PRIMARIO

14

Operazioni

- Ricerca di una tupla con chiave di ricerca K .
 - DENSO ($\Rightarrow K$ è presente nell'indice)
 - » Scansione sequenziale dell'indice per trovare il record (K, p_k)
 - » Accesso al file attraverso il puntatore p_k

Costo: 1 accesso indice + 1 accesso blocco dati

Basi di dati 2019/2020

INDICE PRIMARIO

15

Esempio ricerca dei conti della filiale B

Indice denso

Filiale	Conto	Cliente	Saldo
A	102	Rossi	1000
B	110	Rossi	3020
B	198	Bianchi	500
E	17	Neri	345
M	102	Verdi	1200
E	113	Bianchi	200
H	53	Neri	120
M	78	Verdi	3400

Indice sparso

Basi di dati 2019/2020

INDICE PRIMARIO

16

Operazioni

- Ricerca di una tupla con chiave di ricerca K .
 - SPARSO ($\Rightarrow K$ potrebbe non essere presente nell'indice)
 - » Scansione sequenziale dell'indice fino al record $(K', p_{k'})$ dove K' è il valore più grande che sia minore o uguale a K
 - » Accesso al file attraverso il puntatore $p_{k'}$ e scansione del file (blocco corrente) per trovare le tuple con chiave K .

Costo: 1 accesso indice + 1 accesso blocco dati

Basi di dati 2019/2020

INDICE PRIMARIO

17

Esempio ricerca dei conti della filiale B

Indice denso

Filiale	Conto	Cliente	Saldo
A	102	Rossi	1000
B	110	Rossi	3020
B	198	Bianchi	500
E	17	Neri	345
E	102	Verdi	1200
E	113	Bianchi	200
H	53	Neri	120
M	78	Verdi	3400

Indice sparso

Basi di dati 2019/2020

INDICE SECONDARIO

22

Operazioni

- Ricerca di una tupla con chiave di ricerca K .
 - » Scansione sequenziale dell'indice per trovare il record (K, p_k)
 - » Accesso al bucket B di puntatori attraverso il puntatore p_k
 - » Accesso al file attraverso i puntatori del bucket B.
- Inserimento e cancellazione: come indice primario denso con in più l'aggiornamento dei bucket.

Costo: 1 accesso indice + 1 accesso al bucket + n accessi pagine dati

Basi di dati 2019/2020

INDICE SECONDARIO

23

Esempio di ricerca dei conti di Verdi

buckets

Filiale	Conto	Cliente	Saldo
A	102	Rossi	1000
B	110	Rossi	3020
B	198	Bianchi	500
E	17	Neri	345
E	102	Verdi	1200
E	113	Bianchi	200
H	53	Neri	120
M	78	Verdi	3400

Basi di dati 2019/2020

Rappresentazione degli indici.

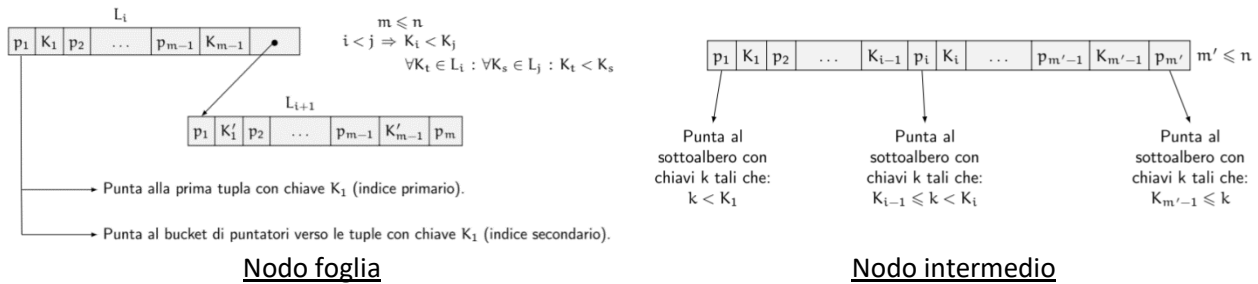
Quando l'indice aumenta di dimensioni, non può risiedere sempre in memoria centrale. Di conseguenza deve essere gestito in *memoria secondaria*, ad esempio tramite un *file sequenziale ordinato*. Tuttavia, le prestazioni di accesso a tale struttura fisica tendono a degradare. Per superare questo problema si introducono per gli indici strutture fisiche diverse come le *strutture ad albero* e le *strutture ad accesso calcolato*.

B⁺-tree.

È una struttura ad albero in cui ogni *nodo* corrisponde ad una *pagina della memoria secondaria*. I legami tra nodi diventano *puntatori a pagina* ed ogni nodo ha un *numero elevato di figli* (pochi livelli e molti nodi foglia). L'albero è *bilanciato*, cioè la lunghezza dei percorsi che collegano la radice ai nodi foglia è costante. Inoltre, inserimenti e cancellazioni non alterano le prestazioni dell'accesso ai dati, infatti l'albero si mantiene bilanciato.

Struttura di un B⁺-tree (con fan-out = n).

Ogni nodo foglia può contenere fino a $n - 1$ valori della chiave di ricerca e fino a n puntatori. Inoltre, i nodi foglia sono ordinati. Ogni nodo intermedio, invece, può contenere fino a n puntatori a nodo.



Vincoli di riempimento.

Ogni nodo foglia contiene un numero di valori chiave *#chiavi* vincolato come segue:

$$\left\lceil \frac{(n-1)}{2} \right\rceil \leq \#chiavi \leq (n-1)$$

Ogni nodo intermedio contiene un numero di puntatori *#puntatori* vincolato come segue (per la radice non vale il minimo):

$$\left\lceil \frac{n}{2} \right\rceil \leq \#puntatori \leq n$$

Operazioni con il B⁺-tree (tempo logaritmico).

Per la *ricerca* dei record con chiave K bisogna:

1. Cercare nel nodo intermedio (o nella radice) il più piccolo valore di chiave maggiore di K ; se tale valore esiste (supponiamo sia K_i) seguire il puntatore p_i , altrimenti seguire il puntatore p_m .
2. Se il nodo raggiunto è un nodo foglia basta cercare il valore K e seguire il corrispondente puntatore verso le tuple, altrimenti si riprende dal passo 1.

Per l'*inserimento* di un nuovo valore della chiave K bisogna:

1. Cercare il nodo foglia NF in cui va inserito il valore K .
2. Se K non è già presente in NF , inserire K e il relativo puntatore in NF rispettando l'ordine.

Per la *cancellazione* di un valore della chiave K bisogna:

1. Cercare il nodo foglia NF in cui va cancellato il valore K .
2. Cancellare K e il relativo puntatore da NF .

Se a causa di un'operazione di inserimento su NF verrebbe violato il vincolo di riempimento massimo di NF , allora bisognerebbe eseguire uno split di NF . Inoltre, se dopo un'operazione di cancellazione su NF viene violato il vincolo di riempimento minimo di NF , allora bisogna eseguire un merge di NF .

Profondità dell'albero.

Il costo di una ricerca nell'indice, in termini di numeri di accessi alla memoria secondaria, risulta pari al numero di nodi acceduti nella ricerca. Tale numero in una struttura ad albero è pari alla *profondità dell'albero*, che nel B⁺-tree è indipendente dal percorso ed è funzione del fan-out n e del numero di valori chiave presenti nell'albero:

$$prof_{B^+-tree} \leq 1 + \log_{\lfloor \frac{n}{2} \rfloor} \left(\frac{\#valoriChiave}{\lfloor \frac{(n-1)}{2} \rfloor} \right)$$

Strutture ad accesso calcolato (hashing).

Si basano su una *funzione di hash* (del tipo $h: K \rightarrow B$ dove K è il dominio delle chiavi e B è il dominio degli indirizzi) che mappa i valori della chiave di ricerca sugli indirizzi di memorizzazione delle tuple nelle pagine dati della memoria secondaria. Un *uso pratico* della funzione di hash negli indici prevede che:

- Si stimi il numero di valori chiave che saranno contenuti nella tabella.
- Si allochi un numero di bucket di puntatori B uguale al numero stimato.
- Si definisca una *funzione di folding* $f: K \rightarrow Z^+$ che trasforma i valori chiave in numeri interi positivi.
- Si definisca una *funzione di hashing* $h: Z^+ \rightarrow B$.

Infine, una caratteristica di una buona funzione di hashing deve essere quella di distribuire in modo *uniforme* e *casuale* i valori della chiave nei bucket, in quanto sarebbe poi pesante cambiare la funzione di hashing dopo che la struttura d'accesso è stata riempita (l'indice andrebbe ricostruito da capo).

Operazioni con l'hashing (tempo costante).

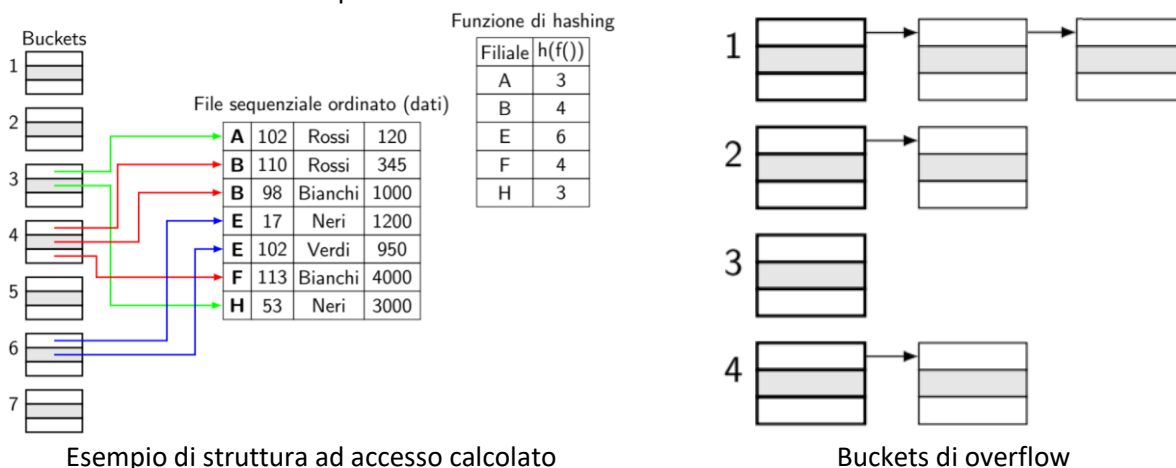
Nell'operazione di *ricerca*, dato un valore di chiave K bisogna trovare la corrispondente tupla. I passi da fare sono i seguenti:

1. Calcolare $b = h(f(K))$, che costa zero.
2. Accedere al bucket b , che costa 1 accesso a pagina.
3. Accedere alle n tuple attraverso i puntatori del bucket, che costa m accessi a pagina (con $m \leq n$).

Le operazioni di *inserimento* e *cancellazione* sono di complessità simile alla ricerca.

Collisioni.

La struttura ad accesso calcolato funziona se i buckets conservano un basso coefficiente di riempimento. Infatti, il problema delle strutture ad accesso calcolato è la gestione delle collisioni. Una *collisione* si verifica quando, dati due valori di chiave K_1 e K_2 con $K_1 \neq K_2$, risulta che $h(f(K_1)) = h(f(K_2))$. Un numero eccessivo di collisioni porta alla *saturazione* del bucket corrispondente. Per gestire tale situazione si prevede la possibilità di allocare *buckets di overflow* collegati al bucket di base, a discapito però delle prestazioni in ricerca (individuato il bucket di base potrebbe essere infatti necessario accedere anche ai buckets di overflow).



Concorrenza.

L'esecuzione concorrente di transazioni senza controllo può generare *anomalie*. È quindi necessario introdurre dei meccanismi di controllo nell'esecuzione delle transazioni per evitarle. Prima di vedere alcuni esempi delle anomalie più tipiche, introduciamo la seguente *notazione*:

- t_i è una transazione.
- $r_i(x)$ è un'operazione di lettura eseguita dalla transazione t_i sulla risorsa x .
- $w_i(x)$ è un'operazione di scrittura eseguita dalla transazione t_i sulla risorsa x .

t1: bot r1(x); x=x+1; w1(x); commit; eot
t2: bot r2(x); x=x+1; w2(x); commit; eot

Operazioni eseguite da t1	Segmento di memoria centrale di t1	Memoria secondaria [buffer]	Segmento di memoria centrale di t2	Operazioni eseguite da t2
bot	Stato iniziale	x=2	Stato iniziale	
r1(x)	2	[2]		
x = x + 1	3	[2]		bot
	3	[2]	2	r2(x)
	3	[2]	3	x = x + 1
	3	[3]	3	w2(x)
	3	3	3	commit
	3	3		eot
w1(x)	3	[3]		
commit	3	3		
eot		3		

x alla fine vale 3
invece di 4

Perdita di aggiornamento

t1: bot r1(x); r1'(x); eot
t2: bot r2(x); x = x+1; w2(x); commit; eot

Operazioni eseguite da t1	Segmento di memoria centrale di t1	Memoria secondaria [buffer]	Segmento di memoria centrale di t2	Operazioni eseguite da t2
bot	Stato iniziale	x=2	Stato iniziale	
r1(x)	2	[2]		
	2	[2]		bot
	2	[2]	2	r2(x)
	2	[2]	3	x = x + 1
	2	[3]	3	w2(x)
	2	3	3	commit
	2	3		eot
r1'(x)	3	[3]		
eot		3		

Letture inconsistenti

Letture inconsistenti

t1: bot r1(x); eot
t2: bot r2(x); x = x+1; w2(x); ... abort;

Operazioni eseguite da t1	Segmento di memoria centrale di t1	Memoria secondaria [buffer]	Segmento di memoria centrale di t2	Operazioni eseguite da t2
bot	Stato iniziale	x=2	Stato iniziale	
		2		bot
		2	2	r2(x)
		[2]	3	x = x + 1
		[2]	3	w2(x)
r1(x)	3	[3]		
eot		2		abort

Letture sporca

Letture sporca

Risorse: x, y, z
Vincoli: x+y+z = 100
t1: bot r1(y); r1(x); r1(z); s = x+y+z; eot
t2: bot r2(y); y = y + 10; r2(z); z = z - 10; w2(y); w2(z); commit; eot

Operazioni eseguite da t1	Segmento di memoria centrale di t1	Memoria secondaria [buffer]	Segmento di memoria centrale di t2	Operazioni eseguite da t2
bot	Stato iniziale	(x,y,z)=20,20,60	Stato iniziale	
r1(y)	-20,-	20,[20],60		
	-20,-	20,[20],60		bot
	-20,-	20,[20],60	-20,-	r2(y)
	-20,-	20,[20],60	-30,-	y = y + 10
	-20,-	20,[20],[60]	-30,60	r2(z)
	-20,-	20,[20],[60]	-30,50	z = z - 10
	-20,-	20,[30],[60]	-30,50	w2(y)
	-20,-	20,[30],[50]	-30,50	w2(z)
	-20,-	20,30,50	-30,50	commit
	-20,-	20,30,50		eot
r1(x)	20,20,-	[20],30,50		
r1(z)	20,20,50	[20],30,[50]		
s=x-y-z	20,20,50	[20],30,[50]		
eot		20,30,50		

S=90
VIOLAZIONE VINCOLO

Aggiornamento fantasma

Schedule.

Uno *schedule* rappresenta una sequenza di operazioni di lettura e scrittura eseguite su risorse della base di dati da diverse transazioni concorrenti. Esso rappresenta quindi una possibile esecuzione concorrente di diverse transazioni. Uno *schedule* può essere:

- *Seriale*, ossia uno *schedule* dove le operazioni di ogni transazione compaiono in sequenza senza essere inframmezzate da operazioni di altre transazioni.
- *Serializzabile*, ossia uno *schedule* "equivalente" ad uno *schedule* seriale.

Equivalenza tra schedule.

Per evitare le anomalie è stato stabilito che si devono accettare solo gli *schedule* serializzabili, ovvero quelli che hanno gli stessi effetti di uno *schedule* seriale. L'idea è che si vogliono considerare equivalenti due *schedule* che producono gli stessi effetti sulla base di dati. Inoltre, secondo l'*ipotesi di commit-proiezione*, si suppone che le transazioni abbiano esito noto e quindi si tolgono dagli *schedule* tutte le operazioni delle transazioni che non vanno a buon fine. Possiamo individuare due tecniche di gestione della concorrenza che richiedono questa ipotesi: una basata sulla *view-equivalenza* e una basata sulla *conflict-equivalenza*.

Definizioni preliminari.

Dato uno schedule S si dice che un'operazione di lettura $r_i(x)$, che compare in S , *legge da* un'operazione di scrittura $w_j(x)$, che compare in S , se $w_j(x)$ precede $r_i(x)$ in S e non vi è alcuna operazione $w_k(x)$ tra le due. Inoltre, dato uno schedule S si dice che un'operazione di scrittura $w_i(x)$, che compare in S , è una *scrittura finale* se è l'ultima operazione di scrittura della risorsa x in S .

S1: $r_1(x) r_2(x) w_2(x) w_1(x)$ $\text{LEGGE_DA}(S1) = \emptyset$
S2: $r_1(x) w_1(x) r_2(x) w_2(x)$ $\text{LEGGE_DA}(S2) = \{(r_2(x), w_1(x))\}$

Relazione "legge da"

S1: $r_1(x) r_2(x) w_2(x) w_3(y) w_1(x)$
 $\text{SCRITTURE_FINALI}(S1) = \{w_3(y), w_1(x)\}$
S2: $r_1(x) w_1(x) r_2(x) w_2(x) w_3(y)$
 $\text{SCRITTURE_FINALI}(S2) = \{w_3(y), w_2(x)\}$

Scritture finali

View-equivalenza.

Due schedule S_1 e S_2 sono *view-equivalenti* se possiedono le stesse relazioni "legge da" e le stesse scritture finali. L'algoritmo per il test di view-equivalenza tra due schedule è di *complessità lineare*.

Test VSR.

Uno schedule S è *view-serializzabile* (VSR) se esiste uno schedule seriale S' tale che S' è view-equivalente a S . L'algoritmo per il test di view-serializzabilità di uno schedule è di *complessità esponenziale*, in quanto è necessario generare tutti i possibili schedule seriali S' considerando tutte le possibili permutazioni delle transazioni che compaiono in S (senza però cambiare l'ordine delle operazioni di una transazione). In conclusione, la VSR non è applicabile nei sistemi reali perché richiede algoritmi di complessità troppo elevata e l'ipotesi di commit-proiezione.

Conflict-equivalenza.

Dato uno schedule S , si dice che una coppia di operazioni (a_i, a_j) , dove a_i e a_j compaiono nello schedule S , rappresentano un *conflitto* se:

- Operano su transazioni diverse ($i \neq j$).
- Operano sulla stessa risorsa.
- Almeno una di esse è un'operazione di scrittura.
- a_i compare in S prima di a_j .

Quindi, due schedule S_1 e S_2 sono *conflict-equivalenti* se possiedono le stesse operazioni e gli stessi conflitti.

Test CSR.

Uno schedule S è *conflict-serializzabile* (CSR) se esiste uno schedule seriale S' tale che S' è conflict-equivalente a S . L'algoritmo per il test di conflict-serializzabilità di uno schedule è di *complessità lineare*. I passi di tale algoritmo sono:

1. Si costruisce il grafo $G(N, A)$ dove $N = \{t_1, \dots, t_n\}$ con t_1, \dots, t_n transazioni di S e $(t_i, t_j) \in A$ se esiste almeno un conflitto (a_i, a_j) in S .
2. Se il grafo costruito è aciclico (ovvero se contiene solo archi (t_i, t_j) con $i < j$), allora S è *conflict-serializzabile*.

Osservazioni.

La conflict-serializzabilità è condizione sufficiente ma non necessaria per la view-serializzabilità, vale a dire che $CSR \subset VSR$. Infatti, se supponiamo che S_1 e S_2 sono *conflict-equivalenti* allora i due schedule avranno:

- *Stesse scritture finali*, perché se così non fosse ci sarebbero almeno due scritture sulla stessa risorsa in ordine diverso e poiché due scritture sono in conflitto i due schedule non sarebbero conflict-equivalenti.
- *Stessa relazione "legge da"*, perché se così non fosse ci sarebbero scritture in ordine diverso o coppie lettura/scrittura in ordine diverso che violerebbero la conflict-equivalenza.

Locking a due fasi.

È il metodo applicato nei sistemi reali per la gestione dell'esecuzione concorrente di transazioni, in quanto non richiede di conoscere in anticipo l'esito delle transazioni. Gli aspetti che caratterizzano il locking a due fasi sono tre: il meccanismo di base per la *gestione dei lock*; la *politica di concessione* dei lock sulle risorse; la regola che garantisce la *serializzabilità*.

Meccanismo di base.

Si basa sull'introduzione di alcune *primitive di lock* che consentono alle transazioni di bloccare (lock) le risorse sulle quali vogliono agire con operazioni di lettura e scrittura:

- $r_lock_k(x)$ – richiesta di un lock *condiviso* da parte di t_k sulla risorsa x per eseguire una lettura.
- $w_lock_k(x)$ – richiesta di un lock *esclusivo* da parte di t_k sulla risorsa x per eseguire una scrittura.
- $unlock_k(x)$ – richiesta da parte della transazione t_k di liberare la risorsa x da un precedente lock.

Per il corretto uso delle primitive da parte delle transazioni, ogni lettura deve essere preceduta da un r_lock e seguita da un $unlock$ (sono ammessi più r_lock contemporanei sulla stessa risorsa); inoltre, ogni scrittura deve essere preceduta da un w_lock e seguita da un $unlock$ (non sono ammessi più w_lock contemporanei sulla stessa risorsa). Se una transazione segue queste due regole, essa si dice *ben formata* rispetto al locking.

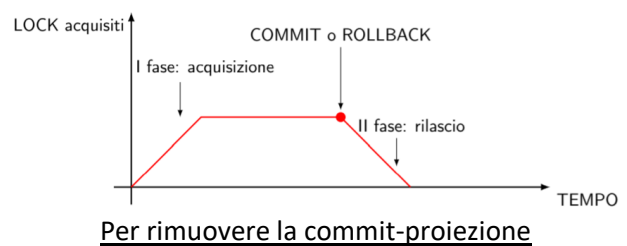
Politica di concessione dei lock.

Il *gestore dei lock* (o gestore della concorrenza) mantiene per ogni risorsa due informazioni: lo stato (libero, r_lock , w_lock) e le transazioni in r_lock (ossia le transazioni che bloccano in lettura la risorsa).

Stato di x	LIBERO		R_LOCK		W_LOCK	
Richiesta						
$r_lock_k(x)$	Esito OK	Operazioni $s(x) = r_lock$ $c(x) = \{k\}$	Esito OK	Operazioni $c(x) = c(x) \cup \{k\}$	Esito Attesa	Operaz. -
$w_lock_k(x)$	Esito OK	Operazioni $s(x) = w_lock$	if $ c(x) =1$ and $k \in c(x)$ then		Esito Attesa	Operaz. -
			Esito OK	Operazioni $s(x) = w_lock$		
			else Attesa	-		
$unlock_k(x)$	Esito Errore	Operazioni -	Esito OK	Operazioni $c(x) = c(x) - \{k\}$ if $c(x) = \emptyset$ then $s(x) = \text{Libero}$ verifica coda	Esito OK	$c(x) = \emptyset$ $s(x) = \text{Libero}$ verifica coda

Serializzabilità.

La regola che garantisce la serializzabilità (*metodo 2PL*) dice che: una transazione, dopo aver rilasciato un lock, non può acquisirne altri. Inoltre, una regola aggiuntiva (*metodo 2PL stretto*) dice che: una transazione può rilasciare i lock solo quando ha eseguito correttamente un commit o un rollback. Confrontando 2PL con le tecniche di serializzabilità viste in precedenza è facile dimostrare che $2PL \subset CSR$.



Blocco critico.

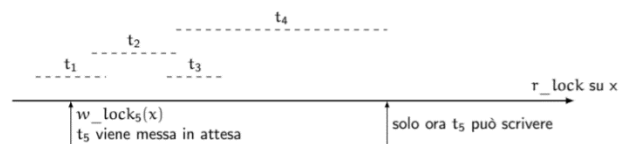
È una situazione di blocco nell'esecuzione di transazioni concorrenti dovuto alla gestione dei lock sulle risorse. Il *blocco critico* si verifica ad esempio quando due transazioni hanno bloccato delle risorse che servono rispettivamente all'una e all'altra per sbloccarsi. Alcune *tecniche* per risolvere il blocco critico sono: *timeout* (la transazione in attesa su una risorsa viene abortita una volta trascorso); *prevenzione* (la transazione deve bloccare tutte le risorse in una sola volta); *rilevamento* (si esegue un'analisi periodica della tabella di lock).

Starvation.

La scelta di gestire i lock in lettura come lock condivisi produce un ulteriore problema, ossia il *blocco di una transazione* (o starvation). Infatti, se una risorsa x viene costantemente bloccata da una sequenza di transazioni che acquisiscono r_lock su x , un'eventuale transazione in attesa di scrivere su x rimane bloccata fino alla fine della sequenza di letture. Anche se poco probabile, questa evenienza può essere scongiurata con tecniche simili a quanto visto per il blocco critico. In particolare, è possibile analizzare la tabella delle relazioni di attesa per verificare da quanto tempo le transazioni stanno attendendo la risorsa e di conseguenza sospendere temporaneamente la concessione di lock in lettura condivisi per consentire la scrittura da parte della transazione che è rimasta bloccata.

t_1	Mem. Centr.	Database			Mem. Centr.	t_2
		$c(x)$	$s(x)$	Val		
bot		\emptyset	L	2		
$r_lock_1(x)$		$\{1\}$	RL	2		
$r_1(x)$	2	$\{1\}$	RL	2		
$x = x + 1$	3	$\{1\}$	RL	2		
	3	$\{1\}$	RL	2		
	3	$\{1, 2\}$	RL	2		
	3	$\{1, 2\}$	RL	2	2	bot
	3	$\{1, 2\}$	RL	2	3	$r_lock_2(x)$
	3	$\{1, 2\}$	RL	2	3	$r_2(x)$
	3	$\{1, 2\}$	RL	2	3	$x = x + 1$
$w_lock_1(x)$	3	$\{1, 2\}$	RL	2	3	$w_lock_2(x)$
→ ATTESA						→ ATTESA

BLOCCO CRITICO



Esempio di blocco critico sulla risorsa $x = 2$

Starvation

Gestione della concorrenza in SQL.

Poiché risulta molto oneroso per il sistema gestire l'esecuzione concorrente di transazioni, SQL prevede la possibilità di rinunciare in tutto o in parte al *controllo di concorrenza* per aumentare le prestazioni del sistema. Ciò significa che è possibile, a livello di singola transazione, decidere di tollerare alcune *anomalie* di esecuzione concorrente a seconda del livello di isolamento scelto. Nel dettaglio:

Livello di isolamento	Perdita di update	Lettura sporca	Lettura inconsistente	Update fantasma	Inserimento fantasma
Serializable	NO	NO	NO	NO	NO
Repeatable read	NO	NO	NO	NO	SI
Read committed	NO	NO	SI	SI	SI
Read uncommitted	NO	SI	SI	SI	SI

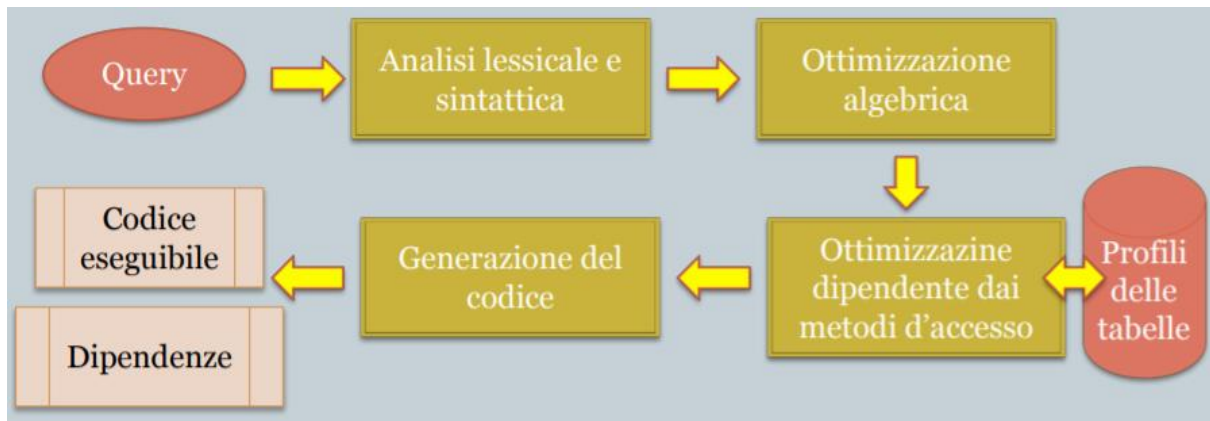
Livelli di isolamento.

Tutti i *livelli di isolamento* richiedono il 2PL stretto per le scritture. *Serializable* lo richiede anche per le letture e applica il lock di predicato per evitare l'inserimento fantasma. *Repeatable read* applica il 2PL stretto per tutti i lock in lettura applicati a livello di tupla e non di tabella (consente inserimenti e quindi non evita l'anomalia di inserimento fantasma). *Read committed* richiede lock condivisi per le letture ma non il 2PL stretto. *Read uncommitted* non applica lock in lettura.

❖ Ottimizzazione delle interrogazioni

Compilazione di un'interrogazione.

Ogni interrogazione espressa in *linguaggio dichiarativo* (ad es. SQL) sottomessa al DBMS, va tradotta in una equivalente espressione in *linguaggio procedurale* (ad es. l'algebra relazionale) per poter generare un piano di esecuzione. L'espressione algebrica va inoltre ottimizzata in base alle regole dell'algebra relazionale (*ottimizzazione algebrica* con anticipo di selezioni e proiezioni) e in base ai *metodi di accesso* disponibili.



Ottimizzazione dipendente dai metodi di accesso – *scansione*.

Una *scansione* si può presentare in queste varianti:

- Scansione e *proiezione* senza eliminazione dei duplicati.
- Scansione e *selezione* in base ad un predicato.
- Scansione e *inserimento/cancellazione/modifica*.

Il costo di una scansione sulla relazione R , però, è sempre pari a $NP(R)$, ovvero al numero di pagine dati della relazione R .

Ottimizzazione dipendente dai metodi di accesso – *ordinamento*.

L'*ordinamento* viene utilizzato per:

- Ordinare il risultato di un'interrogazione (clausola *ORDER BY*).
- Eliminare i duplicati (clausola *SELECT DISTINCT*).
- Raggruppare tuple (clausola *GROUP BY*).

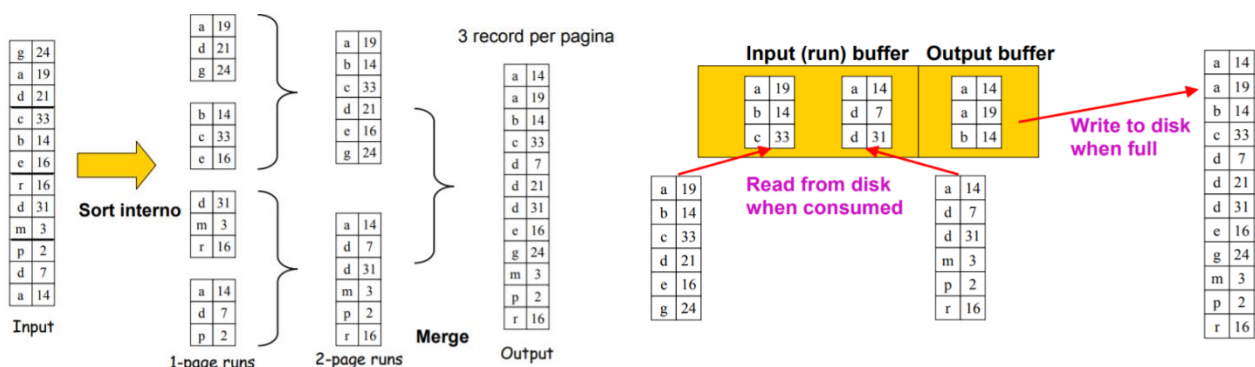
L'*ordinamento* in memoria secondaria avviene secondo l'algoritmo *Z-way Sort-Merge*, il quale si compone delle seguenti fasi:

1. Si leggono una alla volta le pagine della tabella.
2. Le tuple di ogni pagina vengono ordinate facendo uso di un algoritmo di *sort interno*.
3. Ogni pagina così ordinata (detta *run*) viene scritta su memoria secondaria in un file temporaneo.
4. Le *run* vengono unite, applicando uno o più passi di fusione (*merge*), fino a produrre un'unica *run*.

Riguardo il *costo*, si può osservare che nella fase di *sort interno* si leggono e si riscrivono NP pagine e ad ogni passo di *merge* si leggono e si riscrivono NP pagine. Il costo complessivo è pertanto pari a

$$2 \cdot NP \cdot (\lceil \log_2 NP \rceil + 1)$$

in quanto ad ogni passo di merge il numero di run si dimezza (considerando il caso base a due vie $Z=2$).



Z-way Sort-Merge (esempio con $Z=2$ e $NB=3$)

Ottimizzazione dipendente dai metodi di accesso – *accesso diretto via indice.*

L'indice viene utilizzato nelle seguenti interrogazioni:

- Selezioni con condizione atomica di uguaglianza ($A = v$), in cui si usa un indice hash o B⁺-tree.
- Selezioni con condizione di range ($A \geq v_1 \text{ AND } A \leq v_2$), in cui si usa un indice B⁺-tree.
- Selezioni con congiunzione di condizioni di uguaglianza ($A = v_1 \text{ AND } B = v_2$), in cui si sceglie di utilizzare l'indice sulla condizione più selettiva.
- Selezioni con disgiunzione di condizioni di uguaglianza ($A = v_1 \text{ OR } B = v_2$), in cui è possibile utilizzare più indici in parallelo facendo poi un merge dei risultati per eliminare i duplicati.

Ottimizzazione dipendente dai metodi di accesso – *algoritmi di join.*

Le implementazioni più diffuse degli algoritmi di join si riconducono ai seguenti operatori fisici:

- *Nested Loop Join.*
- *Merge Scan Join.*
- *Hash-based Join.*

Anche se logicamente l'operatore di join è commutativo, dal punto di vista fisico vi è una chiara distinzione, che influenza anche le prestazioni, tra operando sinistro (*relazione esterna*) e operando destro (*relazione interna*).

Nested Loop Join.

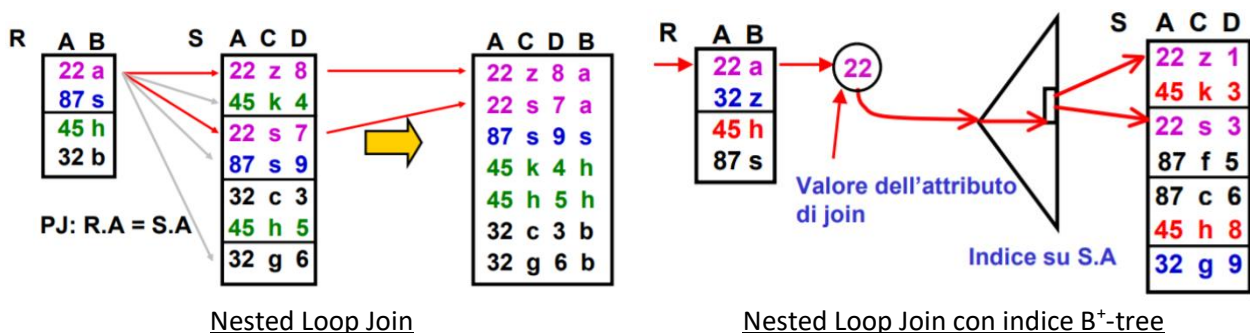
Date due relazioni in input R e S , tra cui sono definiti i predicati di join PJ , e supponendo che R sia la relazione esterna, l'algoritmo *Nested Loop Join* opera come segue:

1. Si considerano le tuple t_r in R .
2. Si considerano le tuple t_s in S .
3. Se la coppia (t_r, t_s) soddisfa PJ , allora si aggiunge tale coppia al risultato.

Il costo di esecuzione dipende dallo spazio a disposizione nei buffer. Nel caso base in cui vi sia un buffer per R e un buffer per S , il costo totale è pari a $NP(R) + NR(R) \cdot NP(S)$ accessi a memoria secondaria, perché devo leggere R una volta e S un numero di volte pari alle righe di R . Nella versione *Block Nested Loop Join*, il costo totale può diventare $NP(R) + NP(R) \cdot NP(S)$ accessi a memoria secondaria, perché posso leggere S tante volte quanti sono i blocchi della relazione esterna R .

Nested Loop Join con indice B⁺-tree.

Data una tupla della relazione esterna R , la scansione completa della relazione interna S può essere sostituita da una scansione basata su un indice A costruito sugli attributi di join di S . Il costo totale diventa quindi pari a $NP(R) + NR(R) \cdot (ProfIndice + NR(S) / VAL(A, S))$ accessi a memoria secondaria, dove $VAL(A, S)$ rappresenta il numero di valori distinti dell'attributo A che compaiono nella relazione S .

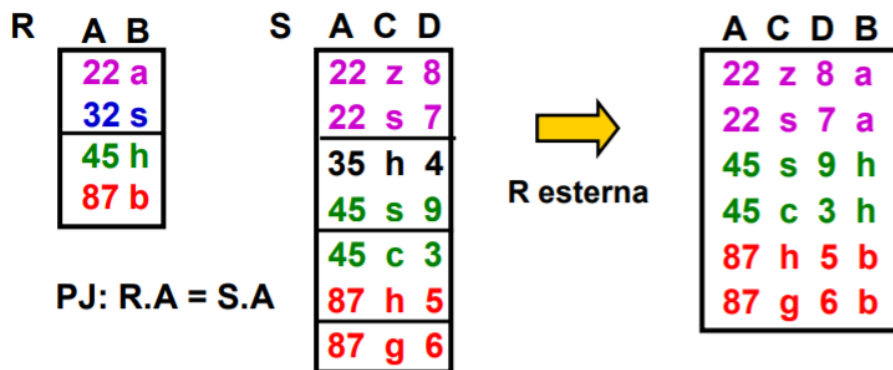


Merge Scan Join.

Il *Merge Scan Join* è applicabile quando entrambi gli insiemi di tuple in input sono *ordinati* sugli attributi di join, il quale deve essere un *equi-join*. Ciò accade se per entrambe le relazioni di input R e S vale almeno una delle seguenti condizioni:

- La relazione è fisicamente ordinata sugli attributi di join (file sequenziale ordinato).
- Esiste un indice sugli attributi di join della tabella che consente una scansione ordinata delle tuple.

La logica dell'algoritmo sfrutta il fatto che entrambi gli input sono ordinati per evitare di fare inutili confronti, il che fa sì che il numero di letture totali sia dell'ordine di $NP(R) + NP(S)$ accessi a memoria secondaria. Con un *indice* B⁺-tree su una relazione, il costo può arrivare al massimo ad essere pari al numero di righe della relazione, se l'indice è già nel buffer. Il costo totale di accessi alla memoria secondaria può quindi diventare $NR(R) + NP(S)$, $NP(R) + NR(S)$ o $NR(R) + NR(S)$ a seconda che l'indice sia sulla prima, sulla seconda o su entrambe le relazioni.



Hash-based Join.

Questo algoritmo è applicabile solo in caso di *equi-join* e non richiede né la presenza di indici né input ordinati. Risulta quindi particolarmente vantaggioso in caso di relazioni molto grandi. L'algoritmo si basa su una funzione di hash H che viene applicata agli attributi di join delle due relazioni (ad es. $R.J$ e $S.J$). A partire da questa idea si hanno diverse implementazioni, che hanno in comune il fatto che R e S vengono partizionate sulla base dei valori della funzione di hash H , e che la ricerca di "matching tuples" avviene solo tra partizioni relative allo stesso valore di H in quanto:

- È possibile che $t_{R.J} = t_{S.J}$ solo se $H(t_{R.J}) = H(t_{S.J})$.
- Se $H(t_{R.J}) \neq H(t_{S.J})$ allora sicuramente vale che $t_{R.J} \neq t_{S.J}$.

Il costo dipende fortemente dal numero di buffer a disposizione e dalla distribuzione dei valori degli attributi di join. Considerando un caso uniforme in cui l'accesso alle "matching tuples" usa il *Block Nested Loop Join*, si ottiene un costo totale pari a $[NP(R) + NP(S)] + [NP(R) \cdot NP(S)]$ accessi a memoria secondaria, in cui la prima parte si riferisce alla costruzione della *hash map*.

Scelta finale del piano di esecuzione.

Data una espressione ottimizzata in algebra:

- Si generano tutti i possibili *piani di esecuzione alternativi* (o comunque un sottoinsieme di questi) considerando operatori alternativi per l'accesso ai dati (ad es. scan sequenziale o via indice), operatori alternativi applicabili nei nodi (ad es. Nested Loop Join o Hash-based Join) e l'ordine delle operazioni da compiere (associatività).
- Si valuta con formule approssimate (stima) il *costo di ogni alternativa* in termini di accessi a memoria secondaria richiesti tenendo conto del profilo delle relazioni, il quale è solitamente memorizzato nel *data dictionary* contenente la stima della cardinalità, la stima della dimensione di una tupla, la stima del numero di valori distinti per ogni attributo e la stima del valore massimo e minimo per ogni attributo.
- Si sceglie l'albero con *costo approssimato minimo*.

❖ Interazione con le applicazioni e cenni ad altre tecnologie

Interazione tra basi di dati e applicazioni.

Esistono diversi *tipi di interazione*:

- *Interazione via cursore* (metodo classico), che prevede una API dedicata messa a disposizione dal DBMS la quale consente di sottomettere comandi SQL al sistema e ottenere risultati di interrogazioni rappresentati come cursori (iteratori su liste di tuple).
- *Interazione via cursore con API standardizzata*, che funziona come la precedente ma ha una certa indipendenza dal DBMS (ad es. libreria JDBC di Java, ODBC di Microsoft).
- *Object Relational Mapping (ORM)*, che consente di gestire nell'applicazione oggetti "persistenti" e di astrarre dalla base di dati relazionale creando un livello di interazione con il DB che maschera l'SQL (ad es. Java Persistence API, Hibernate).

Interazione con DB in Java attraverso JDBC.

In Java è possibile interagire con un DBMS attraverso l'uso della *libreria JDBC* (Java DataBase Connectivity), la cui API fornisce un insieme di classi Java che consentono un accesso standardizzato a qualsiasi DBMS, purché questo fornisca un driver JDBC. Il *driver JDBC* è il modulo software in grado di interagire con il DBMS, in quanto traduce ogni invocazione dei metodi delle classi JDBC in comandi SQL accettati dal DBMS a cui è dedicato.

7 passi per interagire con un DBMS.

- | | |
|--|--|
| 1. Caricare il <i>driver JDBC</i> per il DBMS che si utilizza: | <pre>import java.sql.*; Class.forName("org.postgresql.Driver");</pre> |
| 2. Definire una <i>connessione</i> identificando l'URL della base di dati a cui ci si vuole connettere: | <pre>String URL = "jdbc:postgresql://dbserver/did2014";</pre> |
| 3. Stabilire una connessione istanziando un oggetto della classe <i>Connection</i> : | <pre>String us = "Utente-postgres"; String pw = "Password-utente"; Connection con = DriverManager.getConnection(URL, us, pw);</pre> |
| 4. Creare un oggetto della classe <i>Statement</i> per poter sottomettere comandi al DBMS: | <pre>Statement stat = con.createStatement();</pre> |
| 5. Eseguire un' <i>interrogazione SQL</i> o un <i>comando SQL</i> di aggiornamento di una tabella: | <pre>String query = "SELECT * FROM PERSONA"; ResultSet res = stat.executeQuery(query); String update = "UPDATE PERSONA" + "SET NOME = 'Rosa' WHERE id = 1"; stat.executeUpdate(update);</pre> |
| 6. Processare il risultato dell'interrogazione attraverso l'applicazione dei metodi della classe <i>ResultSet</i> (cursore che consente di accedere alle tuple risultato): | <pre>String query = "SELECT * FROM PERSONA"; ResultSet res = stat.executeQuery(query); while(res.next()) { ... }</pre> |
| 7. Chiudere la connessione: | <pre>con.close();</pre> |

Accesso alle proprietà di una tupla.

È possibile accedere alle proprietà della tupla corrente di un *ResultSet* con i seguenti metodi:

- `getXxx(par)`, dove `Xxx` è un tipo base di Java e `par` può essere un indice di posizione o il nome di un attributo della relazione risultato dell'interrogazione; il metodo restituisce quindi il valore in posizione `par` oppure il valore dell'attributo di nome `par` della tupla corrente.
- `wasNull()`, che si riferisce all'ultima invocazione di `getXxx(par)` e restituisce *true* se il valore letto era uguale al valore nullo.

Caratteristiche della classe *PreparedStatement*.

Usando la classe *PreparedStatement* è possibile ottimizzare l'esecuzione di una interrogazione che deve essere rifatta più volte. Tale classe consente di inserire parametri nell'interrogazione (attraverso l'uso del simbolo `?`) e di valorizzarli (attraverso specifici metodi del tipo `setXxx(pos, valore)`) prima dell'effettiva esecuzione dell'interrogazione stessa. In questo modo si lascia ai metodi della classe *PreparedStatement* il compito di convertire i valori dai tipi Java ai tipi SQL.

```
String q = "SELECT Nome, Cognome" +  
          "FROM Persona" + "WHERE id = ?";  
PreparedStatement pstat =  
    con.prepareStatement(q);  
pstat.setInt(1, 15);  
ResultSet res = pstat.executeQuery();
```

5. Variante:

Transazioni in JDBC.

È possibile eseguire *transazioni* in JDBC come segue:

```
con.setAutoCommit(false);  
...  
con.commit();  
con.setAutoCommit(true);
```

Durante una transazione è possibile attivare il trasferimento per blocchi di tuple in un *ResultSet*. Infatti, l'impostazione di default è che l'intero insieme di tuple risultato dell'interrogazione venga trasferito nel *ResultSet*. Per alterare tale situazione si usa il metodo `setFetchSize(rows)` di *Statement* per fare in modo che il trasferimento delle tuple risultato di un'interrogazione dal DBMS all'applicazione avvenga in lotti di al massimo `rows` tuple (quindi eventualmente prevedendo più interazioni per scaricare tutto il risultato).

Object Relational Mapping.

L'architettura *ORM* è composta da uno strato software (*gestore della persistenza*) che si interpone tra l'applicazione e il DB. L'applicazione vede quindi il DB come insieme di oggetti persistenti. Viene poi specificato in un *file di mapping* la corrispondenza tra oggetti persistenti e tabelle del DB. Tra i *vantaggi* troviamo che:

- Il gestore della persistenza genera parte delle interrogazioni SQL necessarie per generare gli oggetti estraendo dati dalla base di dati in modo trasparente al programmatore.
- La navigazione nei dati segue puntatori tra oggetti.

Tra gli *svantaggi*, invece, troviamo che:

- Le interrogazioni più complesse, che non possono sfruttare i puntatori tra oggetti, vanno gestite ad hoc con SQL nativo.
- Le interrogazioni ad elevata cardinalità e i meccanismi di generazione automatica degli oggetti riferiti da altri possono caricare l'intero DB in memoria, con il rischio di saturare la memoria dedicata all'applicazione.

Tecnologie No-SQL.

Esistono anche i seguenti *approcci* (alternativi a SQL) per l'interazione tra DB e applicazioni:

- *Sistemi Key-Value*, in cui tutti i dati sono rappresentati per mezzo di coppie "chiave-valore" dove la chiave identifica le istanze e il valore può avere una qualsiasi struttura (anche non omogenea nella collezione).
- *Sistemi Document-Store*, in cui i dati sono rappresentati come collezioni di documenti ovvero oggetti con struttura complessa formati da dati incapsulati senza schema fisso e che consentono anche la definizione di indici secondari sulle loro proprietà (ad es. MongoDB, CouchDB).
- *Sistemi Extensible-Record-Store*, in cui i dati sono rappresentati da collezioni di record (tabelle) che possono essere nidificate e a struttura variabile (ad es. BigTable di Google, HBase, Cassandra).
- *Sistemi Graph-Store*, in cui le collezioni di dati sono rappresentati come grafi dove i nodi rappresentano gli oggetti e gli archi le relazioni o proprietà (ad es. Neo4j, OrientDB).

Sistemi Document-Store.

Le caratteristiche principali dei *modelli dei dati* per i sistemi Document-Store sono le seguenti:

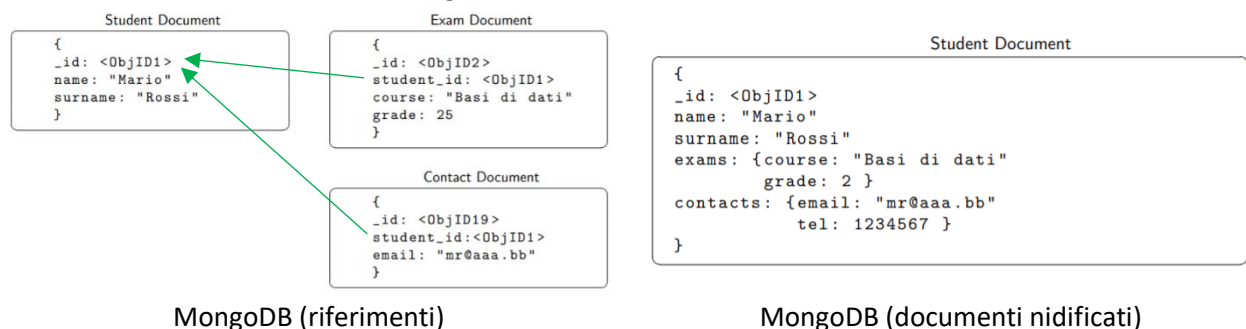
- I dati sono rappresentati da collezioni di oggetti dette *documenti*.
- Gli oggetti hanno *struttura complessa* (non sono tuple piatte ma contengono dati nidificati).
- La *nidificazione* (encapsulation) dei dati è un aspetto chiave.
- Gli oggetti di una collezione non devono necessariamente avere tutti la stessa struttura (tuttavia dovrebbero condividere un nucleo di proprietà comuni).

Progettazione dei dati come documenti.

La decisione chiave nel processo di progettazione dei dati usando questo nuovo approccio riguarda la *struttura dei documenti* (grado di incapsulamento) e la *rappresentazione delle relazioni* (legami) tra i documenti. Esistono in particolare due approcci per rappresentare le relazioni tra documenti:

- *Attraverso riferimenti*, il cui uso porta a schemi normalizzati.
- *Attraverso documenti nidificati*, il cui uso porta a schemi non normalizzati e parzialmente ridondanti (ma molto efficienti in lettura).

Il primo approccio (quello normalizzato e senza ridondanza) consente di rappresentare il dato in modo simile a quanto avviene nel modello relazionale. Il secondo approccio (quello embedded), invece, consente di rappresentare in un'unica istanza quanto viene rappresentato di solito nel modello E-R con una entità insieme alle sue entità deboli. Vale a dire che posso rappresentare nello stesso documento un oggetto con altre istanze di informazione in esso logicamente contenute.



Linguaggio di interrogazione.

Sistemi document-store: interrogazioni

14

Interrogazioni semplici:

- Trovare tutti gli studenti:
`db.studenti.find({})`
- Trovare tutti gli studenti di cognome "Rossi":
`db.studenti.find({cognome: "Rossi"})`
- Trovare tutti gli studenti di cognome "Rossi" e nome "Mario":
`db.studenti.find({cognome: "Rossi", nome: "Mario"})`
- Trovare tutti gli studenti di cognome "Rossi" o "Bianchi":
`db.studenti.find({cognome: {$in: ["Rossi", "Bianchi"]}})`

2019/2020

Sistemi document-store: interrogazioni

15

Interrogazioni su dati incapsulati

- Trovare tutti gli studenti che hanno registrato almeno un esame con voto 30:
`db.studenti.find({esami.voto: 30})`
- Trovare tutti gli studenti che hanno registrato almeno un esame con voto maggiore di 22:
`db.studenti.find({esami.voto: { $gt : 22 }})`

2019/2020

Sistemi document-store: interrogazioni

16

Interrogazioni con proiezione

- Trovare il cognome degli studenti di nome "Mario":
`db.studenti.find({nome: "Mario"}, {cognome: 1})`
- Trovare il cognome degli studenti di nome "Mario" escludendo il campo `_id`:
`db.studenti.find({nome: "Mario"}, {cognome: 1, _id: 0})`
- Trovare la matricola e i voti di tutti gli studenti:
`db.studenti.find({}, { _id: 1, esami.voto: 1 })`

2019/2020

Sistemi document-store: interrogazioni

17

Interrogazioni con join (non consigliate dal sistema)

Per eseguire un join tra due collezioni di documenti è necessario usare la funzione `$lookup`

Interrogazione sull'esempio 2.3

```
db.studenti.aggregate([
  { $lookup:
    { from: esami,
      localField: _id,
      foreignField: stud_id,
      as: "esami_fatti"
    }
  }
])
```

Documenti correlati:

```
{_id: "VR00010"
 nome: "Mario"
 cognome: "Rossi" }
{_id: "ins01"
 nome: "Basi di dati"
 annoac: "2016/2017"
 docente: "Belussi" }
{_id: "ins02"
 nome: "Algebra"
 annoac: "2015/2016"
 docente: "Gregorio" }
{_id: "01" stud_id: "VR00010"
 ins_id: "ins01"
 voto: 22
 data: "1/7/2016" }
{_id: "02" stud_id: "VR00010"
 ins_id: "ins02"
 voto: 26
 data: "5/7/2015" }
```

2019/2020

❖ XML e XML Schema

XML.

XML (eXtensible Markup Language) è un linguaggio di marcatura proposto dal W3C che definisce una sintassi generica per contrassegnare i dati di un documento elettronico con marcatori (tag) semplici e leggibili. La sintassi XML viene utilizzata in contesti molto diversi: pagine web, grafica vettoriale, cataloghi di prodotti, sistemi di gestione di messaggi vocali, ma principalmente per lo *scambio di dati tra applicazioni web*.

Evoluzione.

SGML (Standard Generalized Markup Language – 1986):

- Linguaggio di marcatura strutturato.
- Usato per la rappresentazione elettronica di documenti di tipo testuale.

HTML (HyperText Markup Language – 1995):

- Applicazione di SGML che permette di descrivere gli aspetti di presentazione del documento in una interfaccia (browser).
- Ha un insieme fisso di tag.
- Usato solo per la costruzione di pagine web.

XML (eXtensible Markup Language – 1998):

- Versione leggera di SGML che consente una formattazione semplice e molto flessibile.
- Ha un insieme non fisso di tag, i quali possono essere personalizzati.
- Descrive il contenuto informativo del documento.
- Usato in molti domini diversi.

Documenti ben formati.

XML è più restrittivo di HTML per quanto riguarda il posizionamento dei tag e il modo in cui vengono scritti. Le caratteristiche di un documento XML *ben formato* sono:

- Avere una sola radice.
- L'annidamento dei tag deve essere corretto.
- Tutti i tag aperti devono essere chiusi.
- I valori degli attributi devono essere specificati tra virgolette.

Sintassi base di un documento XML.

Un documento XML è un *file di testo* in cui ogni elemento è caratterizzato da:

- Un *tag iniziale* `<nome_elemento>`.
- Un *tag finale* `</nome_elemento>`.
- Un contenuto che può essere un *valore atomico* (anche vuoto) o un *valore strutturato* attraverso altri elementi XML (elementi figli).

XML è sensibile alla differenza tra maiuscole e minuscole (*case sensitive*). Inoltre, ogni documento XML può essere visto come un *albero di elementi*.

Attributi.

Un elemento può essere corredato di *attributi*, ovvero coppie nome-valore associate al tag iniziale dell'elemento stesso `<nome_elemento nome_attr="valore_attr">`. Non ci sono regole prefissate su quando usare attributi e quando usare elementi per rappresentare informazioni in XML. Tuttavia se XML viene usato come sintassi per la specifica di *informazione semi-strutturata* (dati a struttura complessa e a schema variabile) è evidente che gli elementi forniscono una rappresentazione più accurata del dato semi-strutturato. Gli *svantaggi* dell'uso di attributi rispetto agli elementi sono che:

- Non possono contenere più valori.
- Non possono contenere informazioni ulteriormente strutturate ad albero.
- Non possono essere facilmente estesi.
- Sono difficili da leggere e gestire.

Meglio quindi usare gli elementi per le informazioni e gli attributi per la meta-informazione (*identificatori*).

```
<person sex="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

Contenuto rappresentato con attributi

```
<person>
  <sex>female</sex>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

Contenuto rappresentato con elementi

Sintassi dei nomi degli elementi XML.

I *nomi* degli elementi possono essere costituiti da qualsiasi carattere alfanumerico e possono includere underscore, trattino e punto. Inoltre, devono iniziare solo con lettere, ideogrammi o con il carattere underscore e non possono in alcun caso includere altri caratteri di punteggiatura, virgolette, apostrofi o spazi.

Nomi ben formati:

- `<Nome_persona> Maria </Nome_persona>`
- `<Giorno-Mese-Anno> 10/06/2004 </Giorno-Mese-Anno>`
- `<_indirizzo> Via Stella 10 </_indirizzo>`

Nomi NON ben formati:

- `<Nome persona> Maria </Nome persona>`
- `<Giorno/Mese/Anno> 10/06/2004 </Giorno/Mese/Anno>`
- `<citta'> Verona </citta'>`
- `<1_telefono> 045 1234567 </1_telefono>`
- `<%vendita> 20 </%vendita>`

Intestazione dei documenti XML.

Ogni documento XML dovrebbe iniziare con la seguente dichiarazione XML

```
<?xml version="1.0" encoding="US-ASCII" standalone="yes"?>
```

che indica: la versione di XML; la codifica dei caratteri; se il documento è conforme o meno ad una sintassi esterna (ovvero se non è o meno stand-alone).

Validazione dei documenti XML.

Per ogni insieme di documenti XML che debba rappresentare una certa categoria di informazione è possibile definire un documento XML che ne descrive la sintassi specifica (quali sono i tag ammessi, qual è la struttura dei tag e dei loro attributi, eccetera). Tale documento è detto *XML-Schema* (o file XSD). Un documento XML sarà quindi *valido* se è *ben formato* e *rispetta la sintassi* specificata nel suo file XSD.

XML-Schema.

Tutti gli XML-Schema hanno "schema" come elemento radice. In questo elemento bisogna specificare: il *namespace di W3C* (che contiene gli elementi e i tipi di dato che verranno usati per costruire lo schema); il *target namespace* (che contiene gli elementi che sto definendo nello schema); il *namespace di default* (che di solito non ha un alias perché coincide con il target namespace).

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://www.books.org"
             xmlns="http://www.books.org"
             elementFormDefault="qualified">
  <xsd:element name="BookStore">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Book" minOccurs="1" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Book">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Title" minOccurs="1" maxOccurs="1"/>
        <xsd:element ref="Author" minOccurs="1" maxOccurs="1"/>
        <xsd:element ref="Date" minOccurs="1" maxOccurs="1"/>
        <xsd:element ref="ISBN" minOccurs="1" maxOccurs="1"/>
        <xsd:element ref="Publisher" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Title" type="xsd:string"/>
  <xsd:element name="Author" type="xsd:string"/>
  <xsd:element name="Date" type="xsd:string"/>
  <xsd:element name="ISBN" type="xsd:string"/>
  <xsd:element name="Publisher" type="xsd:string"/>
</xsd:schema>
```

Dichiarazione di elementi e di attributi in XML-Schema.

Summary of Declaring Elements (two ways to do it)

1 `<xsd:element name="name" type="type" minOccurs="int" maxOccurs="int"/>`

A simple type
(e.g., xsd:string)
or the name of
a complexType
(e.g., BookPublication)

A nonnegative
integer

A nonnegative
integer or "unbounded"

Note: *minOccurs* and *maxOccurs* can only
be used in nested (local) element declarations.

2 `<xsd:element name="name" minOccurs="int" maxOccurs="int">
 <xsd:complexType>
 ...
 </xsd:complexType>
</xsd:element>`

2019/2020 36

Summary of Declaring Elements (three ways to do it)

1 `<xsd:element name="name" type="type" minOccurs="int" maxOccurs="int"/>`

2 `<xsd:element name="name" minOccurs="int" maxOccurs="int">
 <xsd:complexType>
 <xsd:restriction base="type"/>
 ...
 </xsd:restriction>
</xsd:element>`

3 `<xsd:element name="name" minOccurs="int" maxOccurs="int">
 <xsd:simpleType>
 <xsd:restriction base="type"/>
 ...
 </xsd:restriction>
</xsd:element>`

2019/2020

Summary of Declaring Attributes (two ways to do it)

1 `<xsd:attribute name="name" type="simple-type" use="how-its-used" default/fixed="value"/>`

xsd:string
xsd:integer
xsd:boolean

required
optional
prohibited

The "use" attribute must be
optional if you use
default or fixed.

2 `<xsd:attribute name="name" use="how-its-used" default/fixed="value">
 <xsd:simpleType>
 <xsd:restriction base="simple-type">
 <xsd:facet value="value"/>
 </xsd:restriction>
 </xsd:simpleType>
</xsd:attribute>`

2019/2020

Summary of Declaring Elements

1. Element with Simple Content.

Declaring an element using a built-in type:

```
<xsd:element name="numStudents" type="xsd:positiveInteger"/>
```

Declaring an element using a user-defined simpleType:

```
<xsd:simpleType name="shapes">  
  <xsd:restriction base="xsd:string">  
    <xsd:enumeration value="triangle"/>  
    <xsd:enumeration value="rectangle"/>  
    <xsd:enumeration value="square"/>  
  </xsd:restriction>  
</xsd:simpleType>  
<xsd:element name="geometry" type="shapes"/>
```

An alternative formulation of the above shapes
example is to inline the simpleType definition:

```
<xsd:element name="geometry">  
  <xsd:simpleType>  
    <xsd:restriction base="xsd:string">  
      <xsd:enumeration value="triangle"/>  
      <xsd:enumeration value="rectangle"/>  
      <xsd:enumeration value="square"/>  
    </xsd:restriction>  
  </xsd:simpleType>  
</xsd:element>
```

2019/2020

Summary of Declaring Elements (cont.)

2. Element Containing Child Elements

Defining the child elements inline:

```
<xsd:element name="Person">  
  <xsd:complexType>  
    <xsd:sequence>  
      <xsd:element name="Title" type="xsd:string"/>  
      <xsd:element name="FirstName" type="xsd:string"/>  
      <xsd:element name="Surname" type="xsd:string"/>  
    </xsd:sequence>  
  </xsd:complexType>  
</xsd:element>
```

An alternate formulation of the above Person example is to create a named complexType and then use that type:

```
<xsd:complexType name="PersonType">  
  <xsd:sequence>  
    <xsd:element name="Title" type="xsd:string"/>  
    <xsd:element name="FirstName" type="xsd:string"/>  
    <xsd:element name="Surname" type="xsd:string"/>  
  </xsd:sequence>  
</xsd:complexType>  
<xsd:element name="Person" type="PersonType"/>
```

2019/2020

Summary of Declaring Elements (cont.)

3. Element Containing a complexType that is an Extension of another complexType

```
<xsd:complexType name="Publication">  
  <xsd:sequence>  
    <xsd:element name="Title" type="xsd:string" maxOccurs="unbounded"/>  
    <xsd:element name="Author" type="xsd:string" maxOccurs="unbounded"/>  
    <xsd:element name="Date" type="xsd:gYear"/>  
  </xsd:sequence>  
</xsd:complexType>  
<xsd:complexType name="BookPublication">  
  <xsd:complexContent>  
    <xsd:extension base="Publication"/>  
  </xsd:complexContent>  
</xsd:complexType>  
<xsd:element name="Book" type="BookPublication"/>
```

2019/2020

Summary of Declaring Elements (cont.)

4. Element Containing a complexType that is a Restriction of another complexType

```
<xsd:complexType name="Publication">  
  <xsd:sequence>  
    <xsd:element name="Title" type="xsd:string" maxOccurs="unbounded"/>  
    <xsd:element name="Author" type="xsd:string" maxOccurs="unbounded"/>  
    <xsd:element name="Date" type="xsd:gYear"/>  
  </xsd:sequence>  
</xsd:complexType>  
<xsd:complexType name="SingleAuthorPublication">  
  <xsd:complexContent>  
    <xsd:restriction base="Publication">  
      <xsd:sequence>  
        <xsd:element name="Title" type="xsd:string" maxOccurs="unbounded"/>  
        <xsd:element name="Author" type="xsd:string"/>  
        <xsd:element name="Date" type="xsd:gYear"/>  
      </xsd:sequence>  
    </xsd:restriction>  
  </xsd:complexContent>  
</xsd:complexType>  
<xsd:element name="Catalogue" type="SingleAuthorPublication"/>
```

2019/2020

Summary of Declaring Elements (concluded)

5. Element Containing Simple Content and Attributes

```
<xsd:element name="apple">  
  <xsd:complexType>  
    <xsd:simpleContent>  
      <xsd:extension base="xsd:string"/>  
      <xsd:attribute name="variety" type="xsd:string" use="required"/>  
    </xsd:simpleContent>  
  </xsd:complexType>  
</xsd:element>
```

Example. `<apple variety="Cortland">Large, green, sour</apple>`

2019/2020 121

Credits

Basato sulle slide fornite dal *prof. Alberto Belussi*

Basato sulla dispensa disponibile nel seguente repository GitHub: <https://github.com/davbianchi/dispense-info-univr/tree/master/triennale/basi-di-dati-teoria>

Repository GitHub personale: <https://github.com/zampierida98/UniVR-informatica>

Indirizzo e-mail personale: zampieri.davide@outlook.com