UNIVERSITÀ
di VERONA
Dipartimento
di INFORMATICA

TRASCRIZIONE DELLE SLIDE DI "Laboratorio di Basi di Dati"

prof. Roberto Posenato

A.A. 2019 – 2020

A cura di: Davide Zampieri

➢ **Data Definition Language (DDL)** – *linguaggio per la definizione delle strutture dati e dei vincoli di integrità*

CREATE TABLE [cap. 5.1 5.2].
L'istruzione principale del DDL è la CREATE TABLE:

- Definisce lo schema di una relazione (o tabella) e ne crea un'istanza vuota.
- Specifica attributi (identificatori), tipo (domini) e vincoli ovvero i concetti fondamentali del DDL.

Identificatori SQL.
Sono stringhe che iniziano con una lettera (UTF-8) o un underscore (_) e che possono contenere anche cifre (0-9). Gli identificatori SQL non sono sensibili al minuscolo/maiuscolo: idPersona e idpersona rappresentano un solo identificatore. Tutti i nomi di tabella, attributo, eccetera sono identificatori SQL.

Domini elementari (predefiniti) [cap. 8].

- Carattere (CHAR/VARCHAR): singoli caratteri o stringhe anche di lunghezza variabile.
- Bit (BOOLEAN/BIT): bit singoli (booleani) o stringhe di bit.
- Numerici esatti (INTEGER/NUMERIC/SERIAL) e approssimati (FLOAT/REAL).
- Date (DATE), orario (TIME/TIMESTAMP) e intervalli di tempo (INTERVAL).

Tipi carattere.
Permettono di rappresentare singoli caratteri oppure stringhe di lunghezza fissa o variabile. I valori devono essere racchiusi tra apici (').

- `CHARACTER` o `CHAR`: carattere singolo.
- `CHARACTER(20)` o `CHAR(20)`: stringa di lunghezza fissa. Se si assegna una stringa di lunghezza inferiore a 20, la stringa viene riempita di spazi fino a renderla lunga 20.
- `CHARACTER VARYING(20)` o `VARCHAR(20)`: stringa di lunghezza variabile (max 20).
- `TEXT`: stringa di lunghezza variabile senza limite fissato. Questa è un'estensione PostgreSQL.

Tipo BOOLEAN o BOOL.
Permette di rappresentare i valori booleani TRUE/FALSE, rappresentati anche come T/F, 1/0, YES/NO, più lo stato unknown (valore nullo), rappresentato come NULL.

Tipi numerici esatti.
Permettono di rappresentare valori interi o valori decimali in virgola fissa.

- `SMALLINT`: valori interi (2 bytes, da −32.768 a +32.767). Equivalente al tipo short in Java.
- `INTEGER`: valori interi (4 bytes, da −231 to +231 − 1). Equivalente al tipo int in Java.
- `NUMERIC [(precisione [, scala])]`: precisione è il numero totale di cifre significative (cioè di cifre a sinistra e a destra della virgola), scala è il numero di cifre dopo la virgola.
- `DECIMAL [(precisione [, scala])]`: equivalente al precedente.
- I tipi NUMERIC e DECIMAL sono equivalenti al tipo java.math.BigDecimal in Java.
- *Esempio* – NUMERIC(5,2) permette di rappresentare valori come 100.01, 100.999 (arrotondato a 101.00), ma non valori come 1000.01!

Tipi numerici approssimati.
Permettono di rappresentare valori in virgola mobile.

- `REAL`: una precisione di 6 cifre decimali.
- `DOUBLE PRECISION`: una precisione di 15 cifre decimali.

REAL e DOUBLE PRECISION sono comunque valori approssimati. Sono implementati secondo lo standard IEEE Standard 754 for Binary Floating-Point Arithmetic. Se si devono rappresentare importi di denaro che contengono anche decimali, MAI usare questi tipi ma usare NUMERIC!

Tipi tempo.
Permettono di rappresentare istanti di tempo.

- `DATE`: istanti del tipo 'years-months-days'. Un valore DATE si rappresenta tra apici (') e lo standard ISO prescrive il formato YYYY-MM-DD.
- `TIME [(precisione)] [WITH TIME ZONE]`: istanti del tipo 'hours:minutes:seconds'; precisione è il numero di cifre decimali da usare per rappresentare frazioni del secondo; WITH TIME ZONE permette di specificare anche [+-]hour:minute, che rappresenta la differenza con l'ora di Greenwich, o nomeFusoOrario.
- `TIMESTAMP [(precisione)] [WITH TIME ZONE]`: tipo DATE + tipo TIME.
- *Esempi* – DATE: '2016-01-15'; TIME(3): '04:05:06.789'; TIME WITH TIME ZONE: '04:05:06-08:00' o '12:01:01 CET'; TIMESTAMP WITH TIME ZONE: '2016-01-24 00:00:00+01'.

Per rappresentare valori come durata di un viaggio, tempo di esecuzione, eccetera si usa il tipo `INTERVAL [fields] [(p)]`:

- fields permette di limitare l'estensione dell'intervallo (YEAR, MONTH, ..., YEAR TO MONTH, DAY TO HOUR, ..., DAY TO SECOND, MINUTE TO SECOND).
- p permette di specificare la precisione delle frazioni di secondo se fields comprende i secondi.

Ci sono diversi modi per specificare valori di intervalli:

- Modalità ISO 8601: 'P quantity unit [...] [T [quantity unit] [...]]' dove l'unità può essere: 'Y', 'M', 'W', 'D', 'H', 'S'.
- Formato ISO alternativo: 'P [years-months-days] [T hours:minutes:seconds]'.

Tipo definito dall'utente.
È possibile definire dei domini specifici assegnando vincoli sui domini di base:

```
CREATE DOMAIN nome AS tipoBase [vincolo];
```

Vincoli.
Vincoli di attributo/intrarelazionali specificano proprietà che devono essere soddisfatte da ogni tupla di una singola relazione della base di dati [cap. 5.3].

Vincolo NOT NULL e DEFAULT.
Il vincolo NOT NULL determina che il valore nullo non è ammesso come valore dell'attributo. Nel caso di vincoli NOT NULL può essere utile specificare un valore di default per l'attributo. Il vincolo DEFAULT specifica un valore di default per un attributo quando un comando di inserimento dati non specifica nessun valore per quell'attributo.

Vincolo UNIQUE.
Il vincolo UNIQUE impone che i valori di un attributo (o di un insieme di attributi) siano una superchiave. Attenzione: UNIQUE su una coppia è diverso da due UNIQUE su due attributi.

Vincolo PRIMARY KEY.
Il vicolo PRIMARY KEY identifica l'attributo che rappresenta la chiave primaria della relazione. Si usa una sola volta per tabella ed implica il vincolo NOT NULL. Due forme di specifica: nella definizione dell'attributo, se è l'unico componente della chiave primaria; come definizione separata a livello di tabella (vincolo di tabella), se invece la chiave primaria è composta di più attributi.

Vincolo CHECK.
Il vincolo CHECK specifica un vincolo generico che devono soddisfare le tuple della tabella. Un vincolo CHECK è soddisfatto se la sua espressione è vera o nulla. In molti casi, un'espressione è nulla se uno degli operandi è nullo. Conviene mettere sempre NOT NULL insieme al vincolo CHECK!

Vincoli di integrità referenziale.
Un vincolo di integrità referenziale crea un legame tra i valori di un attributo (o di un insieme di attributi) A della tabella corrente (detta interna/slave) e i valori di un attributo (o di un insieme di attributi) B di un'altra tabella (detta esterna/master):

- Impone che, in ogni tupla della tabella interna, il valore di A, se diverso dal valore nullo, sia presente tra i valori di B nella tabella esterna.
- L'attributo B della tabella esterna deve essere soggetto a un vincolo UNIQUE (o PRIMARY KEY).
- L'attributo (o insieme di attributi) B può quindi anche NON essere la chiave primaria, ma deve essere identificante per le tuple della tabella esterna.

Un vincolo di integrità referenziale si dichiara nella tabella interna e ha due possibili sintassi [cap. 5.3.5]:

- REFERENCES, vincolo di attributo, da usare quando il vincolo è su un singolo attributo della tabella interna, $|A| = 1$.
- FOREIGN KEY, vincolo di tabella, da usare quando il vincolo coinvolge più attributi della tabella interna, $|A| > 1$.

Modifica della struttura di una tabella.
La struttura di una tabella si può modificare dopo la sua creazione con il comando ALTER TABLE [cap. 5.5]. Di seguito le modifiche più comuni.

- Aggiunta di un nuovo attributo: `ALTER TABLE tabella ADD COLUMN attributo tipo;`
- Rimozione di un attributo: `ALTER TABLE tabella DROP COLUMN attributo;`
- Aggiunta di un vincolo DEFAULT o NOT NULL: `ALTER TABLE tabella ALTER COLUMN attributo { SET DEFAULT valore | DROP DEFAULT | SET NOT NULL | DROP NOT NULL };`
- Aggiunta di un vincolo di integrità referenziale: `ALTER TABLE nome_tabella ADD [CONSTRAINT nome_vincolo] def_vincolo;`
- Rimozione di un vincolo di integrità referenziale: `ALTER TABLE nome_tabella DROP CONSTRAINT nome_vincolo;`

Il nome_vincolo è il nome dato durante la dichiarazione. Se non si definisce un nome, il DBMS ne assegna uno. In PostgreSQL, "\d nome_tabella" mostra la struttura della tabella con i nomi dei vincoli.

➢ Data Manipulation Language (DML) – *linguaggio per manipolare i dati*

Inserimento dati in una tabella.
Una tabella viene popolata con il comando INSERT INTO [cap. 6.1]:

```
INSERT INTO tabella [(elencoAttributi)] VALUES (elencoValori) [, ...];
```

Ogni INSERT aggiunge una o più tuple (righe) in una tabella.

Aggiornamento dati in una tabella.
Una tupla di una tabella può essere modificata con il comando UPDATE [cap. 6.2]:

```
UPDATE tabella SET attributo = espressione [, ... ] [ WHERE condizione ];
```

dove condizione è un'espressione booleana che seleziona quali righe aggiornare. Se WHERE non è presente, tutte le tuple saranno aggiornate.

Cancellazione dati in una tabella.
Le tuple di una tabella vengono cancellate con il comando DELETE [cap. 6.3]:

```
DELETE FROM tabella [WHERE condizione];
```

dove condizione è un'espressione booleana che seleziona quali tuple cancellare. Se WHERE non è presente, tutte le tuple saranno cancellate. Una tabella viene cancellata con il comando DROP TABLE:

```
DROP TABLE tabella;
```

➢ Data Query Language (DQL) – *linguaggio per interrogare*

Comando SELECT [cap. II.7 VI.I-SELECT].
Il processo per recuperare i dati da una base di dati è chiamato interrogazione (query). In SQL, esiste solo un comando per interrogare una base di dati: SELECT. SELECT ha una sintassi che permette di specificare interrogazioni anche molto complesse e che possono richiedere anche ore di computazione.

```
SELECT [DISTINCT]
[ * | expression [[AS] output_name] [, ...] ]
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ DISTINCT ]
    other_select ]
[ ORDER BY expression [ ASC | DESC | USING operator
    ]]
```

Sintassi.

- expression è un'espressione che determina un attributo.
- from_item è un'espressione che determina una sorgente per gli attributi.
- condition è un'espressione booleana per selezionare i valori degli attributi.
- grouping_element è un'espressione per poter eseguire operazioni su più valori di un attributo e considerare il risultato.

Scopo.
L'esecuzione di una SELECT produce una relazione risultato che ha come schema tutti gli attributi elencati nella clausola SELECT e ha come contenuto tutte le tuple t ottenute proiettando sugli attributi dopo SELECT le tuple t0 appartenenti al prodotto cartesiano delle tabelle ottenute dopo il FROM che soddisfano l'eventuale condizione nella clausola WHERE/HAVING/GROUP BY.

Clausola SELECT.

- * è un'abbreviazione per indicare tutti gli attributi delle tabelle.
- expression è un'espressione che coinvolge gli attributi della tabella (può anche essere semplicemente il nome di un attributo).
- output_name è il nome assegnato all'attributo che conterrà il risultato della valutazione dell'espressione expression nella relazione risultato.
- DISTINCT rimuove le tuple duplicate.

Clausola FROM.
from_item può essere UNA delle seguenti clausole:

- `table_name [[AS] alias [(column_alias [, ...])]]` in cui se sono presenti due o più tabelle, si esegue il prodotto cartesiano tra tutte le tabelle. Se ci sono attributi con lo stesso nome su tabelle diverse, nel SELECT questi attributi sono identificati anteponendo il nome della tabella e un punto (.), ad esempio nomeTabella.nomeAttributo.
- `(other_select) [AS] nomeRisultato [(column_alias [, ...])]` è una SELECT innestata. Il risultato della SELECT interna è lo schema con nome nomeRisultato su cui fare la SELECT corrente.
- `from_item [NATURAL] join_type from_item [ON condition]` è la clausola JOIN, la quale permette di selezionare un sottoinsieme del prodotto cartesiano di due o più tabelle; join_type specifica il tipo di join.

Clausola WHERE.

- condition è un'espressione booleana che combina condizioni semplici Ci con gli operatori AND, OR, NOT.
- Ci = expr [θ {expr | const}], dove expr è un'espressione che contiene riferimenti agli attributi delle tabelle che compaiono nella clausola FROM, θ ∈{=,<>,<,>,<=,>=}, const è un valore dei domini di base.
- Una riga soddisfa condition se l'espressione è vera quando valutata con i valori della riga.
- Una riga che non soddisfa condition è scartata dal risultato finale.

Operatore LIKE.
Nella clausola WHERE può apparire l'operatore LIKE per il confronto di stringhe. LIKE è un operatore di pattern matching. I pattern si costruiscono con i caratteri speciali _ e %:

- _ = 1 carattere qualsiasi.
- % = 0 o più caratteri qualsiasi.

Operatore SIMILAR TO [cap. 9.7.2].
L'operatore SIMILAR TO è un LIKE più espressivo che accetta, come pattern, un sottoinsieme delle espressioni regolari POSIX (versione SQL). Esempi di componenti di espressioni regolari:

- _ = 1 carattere qualsiasi.
- % = 0 o più caratteri qualsiasi.
- * = ripetizione del precedente match 0 o più volte.
- + = ripetizione del precedente match UNA o più volte.
- {n,m} = ripetizione del precedente match almeno n e non più di m volte.
- [...] = elenco di caratteri ammissibili.

Operatore BETWEEN.
Nella clausola WHERE può apparire l'operatore BETWEEN … AND … per testare l'appartenenza di un valore ad un intervallo (gli estremi sono inclusi).

Operatore IN.
Nella clausola WHERE può apparire l'operatore IN per testare l'appartenenza di un valore ad un insieme.

Operatore IS NULL.
Nella clausola WHERE può apparire l'operatore IS NULL per testare se un valore è NULL o meno. In SQL, NULL non è uguale a NULL. Evitare di usare '=' o '<>' con il valore NULL!

Operatore ORDER BY.
La clausola ORDER BY ordina le tuple del risultato rispetto agli attributi specificati.

```
ORDER BY attributo [{ASC | DESC}] [, ... ];
```

ASCendente è lo standard.

Operatori di aggregazione [cap. 9.20].
Sono operatori che permettono di determinare UN valore considerando i valori ottenuti da una SELECT. Quando si usano gli operatori aggregati, dopo la SELECT non possono comparire espressioni che usano i valori presenti nelle singole tuple perché il risultato è sempre e solo una tupla.

COUNT.
Restituisce il numero di tuple significative nel risultato dell'interrogazione:

```
COUNT ({ * | expr | ALL expr | DISTINCT expr }])
```

- expr è un'espressione che usa attributi e funzioni di attributi ma non operatori di aggregazione.
- COUNT(*) ritorna il numero di tuple nel risultato dell'interrogazione.
- COUNT(expr) o COUNT(ALL expr) ritorna il numero di tuple in ciascuna delle quali il valore expr è non nullo.
- COUNT(DISTINCT expr) è come COUNT(expr) ma con l'ulteriore condizione che i valori di expr siano distinti.

SUM/MAX/MIN/AVG.
Determinano un valore numerico (SUM/AVG) o alfanumerico (MAX/MIN) considerando le tuple significative nel risultato dell'interrogazione. Ammettono DISTINCT con significato uguale a quanto detto per COUNT.

Interrogazioni con raggruppamento – GROUP BY.
Un raggruppamento è un insieme di tuple che hanno medesimi valori su uno o più attributi caratteristici del raggruppamento. La clausola GROUP BY permette di determinare tutti i raggruppamenti delle tuple della relazione risultato (tuple selezionate con la clausola WHERE) in funzione degli attributi dati. In una interrogazione che usa GROUP BY, la clausola SELECT contiene solamente gli attributi (da 0 a tutti) utilizzati per il raggruppamento ed eventuali espressioni con operatori di aggregazione su attributi non raggruppati (NON SI POSSONO SPECIFICARE attributi che non sono raggruppati). Nel GROUP BY non si possono usare espressioni con operatori di aggregazione. In PostgreSQL 10, GROUP BY può accettare alias di colonna!

Interrogazioni con raggruppamento – HAVING.
La clausola WHERE permette di selezionare le righe che devono far parte del risultato. La clausola HAVING permette di selezionare i raggruppamenti che devono far parte del risultato. La sintassi è `HAVING bool_expr`, dove bool_expr è un'espressione booleana che può usare gli attributi usati nel GROUP BY e/o gli altri attributi mediante operatori di aggregazione. Quindi si possono specificare espressioni con operatori di aggregazione su attributi non raggruppati anche nella clausola HAVING.

Interrogazioni con JOIN.
La clausola FROM ammette come argomento `table_name [[AS] name]] [, ...]`. Si è visto che se sono presenti due o più nomi di tabelle, si esegue il prodotto cartesiano tra tutte le tabelle e lo schema del risultato può contenere tutti gli attributi del prodotto cartesiano. Il prodotto cartesiano di due o più tabelle è un CROSS JOIN. A partire da SQL-2, esistono altri tipi di JOIN (join_type): [INNER] JOIN, LEFT [OUTER] JOIN, RIGHT [OUTER] JOIN e FULL [OUTER] JOIN. La sintassi diventa quindi:

```
table_name [NATURAL] join_type table_name [ON join_condition [, ...]]
```

dove join_condition è un'espressione booleana che seleziona le tuple del join da aggiungere al risultato. Le tuple selezionate possono essere poi filtrate con la condizione della clausola WHERE.

Tipi di JOIN.

- INNER JOIN – Rappresenta il tradizionale θ join dell'algebra relazionale. Combina ciascuna riga r1 di table1 con ciascuna riga r2 di table2 che soddisfa la condizione della clausola ON.
- LEFT OUTER JOIN – Si esegue un INNER JOIN. Poi, per ciascuna riga r1 di table1 che non soddisfa la condizione con qualsiasi riga r2 di table2, si aggiunge una riga al risultato con i valori di r1 e assegnando NULL agli altri attributi. Il LEFT [OUTER] JOIN non è simmetrico! Con le medesime tabelle si possono avere risultati diversi invertendo l'ordine delle tabelle nel join!
- RIGHT OUTER JOIN – Si esegue un INNER JOIN. Poi, per ciascuna riga r2 di table2 che non soddisfa la condizione con qualsiasi riga r1 di table1, si aggiunge una riga al risultato con i valori di r2 e assegnando NULL agli altri attributi.
- FULL OUTER JOIN – È equivalente a fare INNER JOIN + LEFT OUTER JOIN + RIGHT OUTER JOIN. Non è equivalente al CROSS JOIN!

Interrogazioni nidificate.

Un'interrogazione è nidificata quando è presente all'interno di un'altra interrogazione. Un'interrogazione dentro la clausola FROM è un esempio di interrogazione nidificata. SQL permette di fare un'interrogazione nidificata anche dentro la clausola WHERE. L'interrogazione nidificata può essere usata per rendere più sofisticate le selezioni di righe della query principale. La selezione è spesso basata sul confronto tra il valore di un attributo e i valori di altri attributi della stessa riga o valori costanti. Se si usano interrogazioni nidificate, il confronto è tra un valore di un attributo (valore singolo) e il risultato di una interrogazione (possibile insieme di valori). Quindi gli operatori di confronto tradizionali (<,>,<>,=,...) NON possono essere usati. Si devono invece usare dei nuovi operatori (EXISTS, IN, ALL, ANY/SOME) che estendono quelli tradizionali a questo tipo di confronti.

Operatore EXISTS [cap. 9.22.1].

La sintassi è `EXISTS (subquery)` dove subquery è una SELECT. EXISTS ritorna falso se subquery non contiene righe; vero altrimenti. EXISTS è significativo quando nella subquery si selezionano righe usando i valori della riga corrente nella SELECT principale, ovvero quando si fa il data binding. Se si usano attributi esterni nella subquery (data binding), la subquery deve essere valutata per ogni riga della SELECT principale.

Operatore IN [cap. 9.22.2].

La sintassi è `[ROW] (expr [,...]) IN (subquery)` dove:

- expr è un'espressione costruita con un attributo della query principale. Ci possono essere una o più espressioni.
- La subquery deve restituire un numero di colonne pari al numero di espressioni in (expr [,...]).
- I valori delle espressioni vengono confrontati con i valori di ciascuna riga del risultato di subquery. Il confronto ritorna vero se i valori sono uguali ai valori di almeno una riga della subquery.

Operatore ANY/SOME [cap. 9.22.4].

La sintassi è `expression operator ANY (subquery)` dove:

- subquery è una SELECT che deve restituire UNA sola colonna.
- expression è un'espressione che coinvolge attributi della SELECT principale.
- operator è un operatore di confronto.
- ANY ritorna vero se expression è operator rispetto al valore di una qualsiasi riga del risultato di subquery. SOME si può usare come sinonimo di ANY.

Operatore ALL [cap. 9.22.5].
La sintassi è `expression operator ALL (subquery)` dove:

- subquery è una SELECT che deve restituire UNA sola colonna.
- expression è un'espressione che coinvolge attributi della SELECT principale.
- operator è un operatore di confronto.
- ALL ritorna vero se expression è operator rispetto al valore di ciascuna riga del risultato di subquery.

Interrogazioni di tipo insiemistico [cap. 7.4].
SQL mette a disposizione anche degli operatori insiemistici per manipolare i risultati di più query. Gli operatori insiemistici si possono utilizzare solo al livello più esterno di una query, in quanto operano sui risultati di due clausole SELECT. Gli operatori insiemistici sono: UNION, INTERSECT e EXCEPT. Si possono avere sequenze di UNION/INTERSECT/EXCEPT. La sintassi è:

```
query1 {UNION | INTERSECT | EXCEPT} [ALL] query2
```

- Gli operatori si possono applicare solo quando query1 e query2 producono risultati con lo stesso numero di colonne e di tipo compatibile fra loro.
- Tutti gli operatori eliminano i duplicati dal risultato a meno che ALL non sia stato specificato.
- UNION aggiunge il risultato di query2 a quello di query1.
- INTERSECT restituisce le righe che sono presenti sia nel risultato di query1 sia in quello di query2.
- EXCEPT restituisce le righe di query1 che non sono presenti nel risultato di query2. In pratica esegue la differenza insiemistica.

Viste.
Le viste sono tabelle "virtuali" il cui contenuto dipende dal contenuto delle altre tabelle della base di dati. In SQL le viste vengono definite associando un nome ed una lista di attributi al risultato dell'esecuzione di un'interrogazione. Ogni volta che si usa una vista, si esegue la query che la definisce. Nell'interrogazione che definisce la vista possono comparire anche altre viste. SQL non ammette però: dipendenze immediate (definire una vista in termini di sé stessa) o ricorsive (definire una interrogazione di base e una interrogazione ricorsiva); dipendenze transitive circolari (V1 definita usando V2, V2 usando V3, ..., Vn usando V1). La sintassi è `CREATE [TEMP] VIEW nome [(col_name [, ...])] AS query` dove:

- TEMP indica che la vista è temporanea. Quando ci si sconnette, la vista viene distrutta. È un'estensione di PostgreSQL.
- column_name sono i nomi delle colonne che compongono la vista. Se non si specificano, si ereditano dalla query.
- query deve restituire un insieme di attributi pari e nel medesimo ordine a quello specificato con (col_name [, ...]) se presente.

➢ Analisi delle prestazioni delle query SQL – *indici e comando EXPLAIN*

Indici.
Gli indici sono delle strutture dati ausiliare che permettono di accedere ai dati di una tabella in maniera più efficiente. Dato che un indice è una struttura dati ausiliaria, deve essere sempre mantenuto aggiornato in base al contenuto della tabella in cui è definito. Il costo di aggiornamento può essere significativo quando ci sono molti indici definiti sulla medesima tabella. Gli indici quindi devono essere definiti considerando la loro efficacia.

Comando \timing.
Il comando "\timing" attiva/disattiva la visualizzazione del tempo di pianificazione e calcolo di una query. Il tempo è visualizzato subito dopo la visualizzazione del risultato della query.

Sintassi [cap. VI.I-CREATE INDEX].

La sintassi è

```
CREATE INDEX [ nome ] ON nomeTabella [ USING method ] ({nomeAttr|(expression)}
                                [ASC|DESC] [, ...])
```

dove:

- method è il tipo di indice (più avanti i dettagli).
- nomeAttr o expression indicano su quali attributi o espressioni di attributi si deve creare l'indice.
- ASC/DESC specifica se l'indice è ordinato in modo ascendente o discendente.

ALTER INDEX e DROP INDEX permettono di modificare/rimuovere indici creati precedentemente.

Caso pratico.

Un indice, una volta creato, è usato dal sistema ogni volta che l'ottimizzatore di query determina sia opportuno. Un DBMS crea gli indici in modo automatico solo per attributi dichiarati PRIMARY KEY. Per tutti gli altri attributi si deve fare una dichiarazione esplicita di CREATE INDEX per attivarlo. Un indice può velocizzare anche i comandi UPDATE/DELETE quando nella clausola WHERE ci sono attributi indicizzati. Il comando `ANALYZE [nomeTabella]` aggiorna le statistiche circa il contenuto delle tabelle (e indici). È eseguito in modo automatico dal sistema a intervalli regolari. L'ottimizzatore di query usa queste statistiche per decidere quando usare gli indici.

Creazione degli indici.

Quanto è possibile rendere più veloce una query definendo degli indici? Ricordando che gli indici sono usati per indirizzare tuple che contengono i valori dati sugli attributi specificati, una possibilità è quella di definire gli indici su tutti gli attributi usati per fare i join o le selezioni.

Tipi.

PostgreSQL ammette diversi tipi di indici: B-tree, hash, GiST, SP-GiST, GIN e BRIN. Ciascun tipo usa una tecnica algoritmica diversa e risulta migliore di altri per specifici tipi di query. Il tipo si specifica dopo la parola chiave USING. Inoltre, se non si specifica il tipo di indice, il sistema crea di default un indice di tipo B-tree.

Indice B-tree.

L'ottimizzatore di query considera un indice B-tree ogni volta che l'attributo indicizzato è coinvolto in un confronto che usa degli operatori (<, <=, =, >=, >). Un indice B-tree può essere considerato anche quando ci sono BETWEEN, IN, IS NULL e LIKE. Con la localizzazione it_IT.UTF-8 la comparazione di stringhe con operatori diversi da '=' e con LIKE ha regole diverse (minuscole/maiuscole) rispetto a UTF-8. Un indice su un attributo VARCHAR in una base di dati con LC_LOCALE = it_IT.UTF-8 dovrebbe essere dichiarato come:

```
CREATE INDEX nome ON nomeTab(nomeAttr varchar_pattern_ops);
```

se si vuole che l'indice sia usato anche quando il confronto usa <, >, <>, LIKE. La parola chiave varchar_pattern_ops abilita l'uso degli operatori rispettando le regole imposte dal locale (it_IT.UTF-8).

Indice hash.

Il tipo hash ha un uso limitato perché può essere usato solo quando i confronti sono di eguaglianza. La sua gestione poi è più complicata in base di dati replicate o in caso di crash: il gruppo di PostgreSQL ne sconsiglia l'uso.

Altri tipi di indice.

Gli altri tipi di indice sono indicati per particolari strutture dati. Per esempio, GiST è indicato quando un attributo è di tipo geometrico bi-dimensionale (tipo non standard). Vedere il cap. 11.2 del manuale di PostgreSQL per maggiori dettagli.

Indici multi-attributo.
Se si hanno query che hanno condizioni su coppie (ma anche terne, eccetera) di attributi di una tabella, un indice multi-attributo definito usando la coppia (o terna, eccetera) di attributi potrebbe essere più utile rispetto agli indici definiti sui singoli attributi. NON sempre gli indici multi-attributo possono essere usati: per esempio in espressioni con OR non è possibile.

Indici di espressioni.
Query con condizioni su espressioni/funzioni di uno o più attributi di una tabella possono essere velocizzate creando indici sulle medesime espressioni/funzioni di attributi. È possibile definire indici su espressioni di attributi con la seguente sintassi:

```
CREATE INDEX nome ON nomeTabella(expression);
```

dove expression è un'espressione su uno o più attributi. Per opportunità, si considerano espressioni che sono frequenti nelle interrogazioni usate (ad esempio la funzione LOWER).

Costo e regola pratica d'uso.
Gli indici costano tempo e memoria. Se si considera solo il tempo, il costo maggiore è mantenere gli indici aggiornati. Infatti, per ogni operazione di INSERT/UPDATE/DELETE su una tabella con indici, il DBMS deve aggiornare anche gli indici della tabella. Una regola pratica è quella definire gli indici in base alle query più frequenti [cap. 11.11]. Il comando PostgreSQL "\di" visualizza l'elenco degli indici definiti in una base di dati. Il comando PostgreSQL "\d nomeTabella" visualizza la definizione di tabella e degli indici associati.

Explain.
Ogni DBMS ha un ottimizzatore di query che determina il piano per eseguire una query nel minor tempo possibile. Il comando EXPLAIN [cap. 14.1] permette di vedere il piano di esecuzione di una query che l'ottimizzatore determina senza eseguire la query. La corretta interpretazione dell'output di un EXPLAIN richiede una certa esperienza e conoscenza dei meccanismi dell'ottimizzatore e serve ad individuare le cause per cui le query hanno basse prestazioni.

Struttura di un explain.
Il piano di esecuzione di una query è un albero di nodi di esecuzione. Le foglie sono nodi di scansione: le esecuzioni di questi nodi restituiscono indirizzi di righe della tabella. Esistono differenti tipi di scansioni: sequenziali, indicizzate e mappate su bit. Se una query contiene JOIN, GROUP BY, ORDER BY o altre operazioni sulle righe, allora ci saranno altri nodi di esecuzione sopra le foglie nell'albero. L'output di EXPLAIN ha una riga per ciascun nodo nell'albero di esecuzione dove viene indicato il tipo di operazione e una stima del costo di esecuzione. Ulteriori proprietà del nodo possono essere mostrate con una riga indentata subito sotto. La prima riga dell'output ha la stima del costo totale di esecuzione della query. Questo è il costo che l'ottimizzatore cerca di minimizzare. Tale costo è dato in termini di numero di accessi alla memoria secondaria (page disk). Inoltre, la prima riga dell'output contiene anche il numero totale di righe valutate dall'esecutore, che però non è sempre uguale al numero totale di righe del risultato vero e proprio.

Scansioni mappate su bit.
Prima viene eseguito il nodo foglia (Bitmap INDEX Scan): grazie all'indice B-tree, l'ottimizzatore determina un vettore di indirizzi di righe da considerare (in un B-tree se la chiave cercata K non è pari a nessuna chiave Ki ma è compresa tra Ki e Ki+1, può essere presente nel sottoalbero Pi). Tale vettore viene poi passato al nodo padre (Bitmap Heap Scan) che carica le righe ed esegue la selezione finale ricontrollando la condizione (e facendo eventuali altre selezioni).

Uso di due indici sulla stessa tabella.
Se ci sono due indici sulla stessa tabella posso avere due casi (a seconda che si tratti di una congiunzione o di una disgiunzione):

- Ciascuna foglia scansiona un indice. Il padre delle foglie (BitmapAnd) fa l'intersezione degli indirizzi delle righe trovate. La root (Bitmap Heap Scan) carica le righe e ricontrolla che le condizioni siano soddisfatte.
- Ciascuna foglia scansiona un indice. Il padre delle foglie (BitmapOr) esegue l'unione degli indirizzi delle righe trovate. La root (Bitmap Heap Scan) carica le righe e ricontrolla che le condizioni siano soddisfatte.

Join mediante hash table.
Il nodo Hash prepara una hash table con una scansione sequenziale (Seq Scan). Il nodo padre Hash Join, per ogni riga fornita dal primo figlio (Seq Scan), cerca nella hash table (secondo figlio) la riga da unire secondo la condizione. L'unico Seq Scan che si può ottimizzare è quello del primo figlio.

Join mediante nested loop.
Il join viene fatto usando un loop (Nested Loop): ogni riga prodotta dal primo nodo figlio si unisce con ogni riga prodotta dal secondo nodo figlio. Entrambi i nodi figlio possono essere Seq Scan o INDEX Scan.

Join mediante merge join.
Merge Join esegue il join ordinando prima le due tabelle rispetto gli attributi di join. Se una tabella non è ordinata appare un nodo Sort, il cui Seq Scan si potrebbe ottimizzare creando un indice (perché eviterebbe il Sort).

Versione avanzata di explain.
Il comando EXPLAIN ANALYZE mostra il piano di esecuzione, esegue la query senza registrare eventuali modifiche e mostra, infine, una stima verosimile dei tempi di pianificazione e di esecuzione. Per ogni nodo del piano mostra inoltre: dettaglio righe rimosse, memoria usata e tempi di esecuzione in ms.

➢ Controllo di concorrenza in SQL – *livelli di isolamento e lock espliciti*

Transazioni.
Una transazione SQL è una sequenza di istruzioni SQL che può essere eseguita in concorrenza con altre transazioni in modo isolato. Se si implementasse un isolamento completo, il grado di parallelismo sarebbe limitato. Si accettano quindi diversi livelli di isolamento per favorire un maggior grado di parallelismo. Tuttavia, diversi livelli di isolamento possono determinare delle anomalie di esecuzione:

- Perdita di aggiornamento (lost update).
- Lettura sporca (dirty read).
- Lettura inconsistente (non-repeatable read).
- Aggiornamento fantasma (ghost update).
- Inserimento fantasma (phantom).
- Mancata serializzazione (serialization anomaly).

In PostgreSQL una transazione inizia con il comando BEGIN e termina o con il comando COMMIT (per confermare tutte le istruzioni della transazione) o con il comando ROLLBACK (per annullare tutte le operazioni).

Controllo della concorrenza.
PostgreSQL mantiene la consistenza dei dati usando un modello multi-versione (Multi-Version Concurrency Control, MVCC). MVCC è una metodologia più versatile del locking a due fasi (stretto) tipica dei DBMS tradizionali. In generale, le transazioni in PostgreSQL sono gestite come:

- Ciascuna transazione vede un'istantanea della base di dati.
- Le letture su questa istantanea sono sempre possibili e non sono mai bloccate anche se ci sono altre transazioni che stanno modificando la base di dati.
- Le scritture possono essere sospese. Una scrittura è sospesa quando, in parallelo, un'altra transazione (non ancora chiusa) ha modificato la sorgente dei dati che si vuole aggiornare.
- Al COMMIT, il sistema registra sempre l'istantanea aggiornata come nuova base di dati.

In questo modo ci sono meno conflitti e una performance più ragionevole in ambito multiutente.

Livelli di isolamento.
I 4 livelli di isolamento offerti da PostgreSQL 10 sono, in ordine decrescente di isolamento:

1. Serializable – Garantisce che un'intera transazione è eseguita in un qualche ordine sequenziale rispetto ad altre transazioni: completo isolamento da transazioni concorrenti. Anomalie possibili: nessuna!
2. Repeatable Read – Garantisce che i dati letti durante la transazione non cambieranno a causa di altre transazioni: rifacendo la lettura dei medesimi dati, si ottengono sempre gli stessi risultati. Anomalie possibili: mancata serializzazione.
3. Read Committed – Garantisce che qualsiasi SELECT di una transazione vede solo i dati confermati (COMMITTED) prima che la SELECT inizi. È il livello di isolamento di default usato da PostgreSQL. Anomalie possibili: lettura inconsistente, aggiornamento fantasma, inserimento fantasma e mancata serializzazione.
4. Read Uncommitted – In PostgreSQL 10 è implementato come Read Committed. Quindi NON esiste questo livello di isolamento.

Il cambio di livello di isolamento si effettua con SET TRANSACTION appena dopo il BEGIN della transazione o in modo più compatto con: `BEGIN TRANSACTION ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED };`

Read Committed.
È il livello di default, quindi è sufficiente BEGIN. SELECT vede solo i dati registrati (COMMITTED) in altre transazioni e quelli modificati da eventuali comandi precedenti nella medesima transazione. UPDATE e DELETE vedono i dati come SELECT. Inoltre, se i dati che devono essere aggiornati sono stati modificati ma non registrati in transazioni concorrenti, il comando deve:

- Attendere il COMMIT o ROLLBACK della transazione dove è stato fatto il cambio.
- Riesaminare le righe selezionate per verificare se soddisfano ancora i criteri del comando.

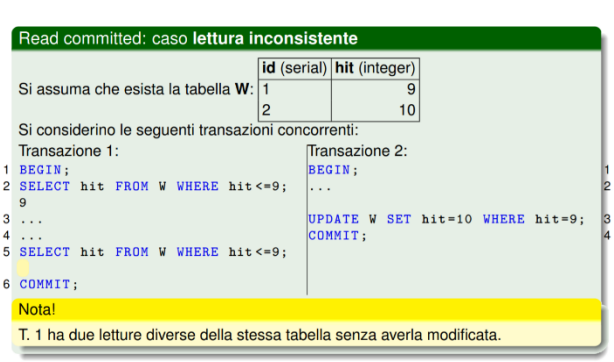
Repeatable Read.
Differisce da Read Committed per il fatto che i comandi di una transazione vedono sempre gli stessi dati. PostgreSQL associa alla transazione un'istantanea della base di dati all'esecuzione del suo primo comando. Repeatable Read in PostgreSQL è più stringente di quanto richiesto dallo standard SQL. Due SELECT identiche successive vedono sempre gli stessi dati. UPDATE e DELETE vedono i dati come SELECT. Se i dati che devono essere aggiornati sono stati modificati ma non registrati in transazioni concorrenti, il comando deve attendere il COMMIT/ROLLBACK della transazione dove è stato fatto il cambio. In caso di ROLLBACK, UPDATE e DELETE possono procedere riesaminando le righe selezionate. In caso di COMMIT, i dati sono cambiati, quindi UPDATE e DELETE vengono bloccati con un errore. Se si usa questo livello di isolamento, si deve quindi prevedere la possibilità di transazioni abortite per aggiornamenti concorrenti.

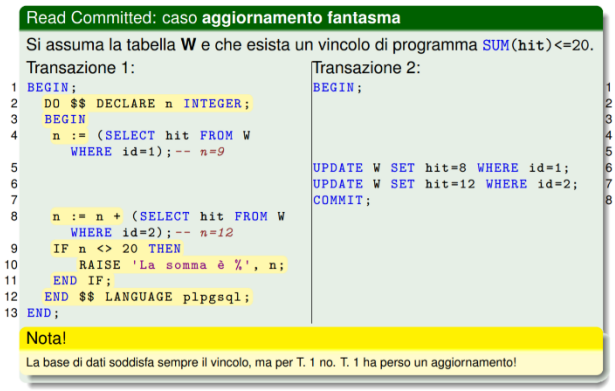
Serializable.
È il più restrittivo: nessuna anomalia è permessa. Se si usa questo livello, si deve prevedere la possibilità di transazioni abortite per aggiornamenti concorrenti (come nel caso di Repeatable Read). L'uso di transazioni Serializable rende più semplice lo sviluppo di programmi SQL: è sufficiente dimostrare che una transazione, da sola, determina sempre uno stato corretto per avere la garanzia che essa continuerà ad essere corretta anche in ambiente concorrente dichiarandola Serializable. D'altra parte però dichiarare tutte le transazioni Serializable determina, in generale, una limitazione alle prestazioni (throughput) di un DMBS.

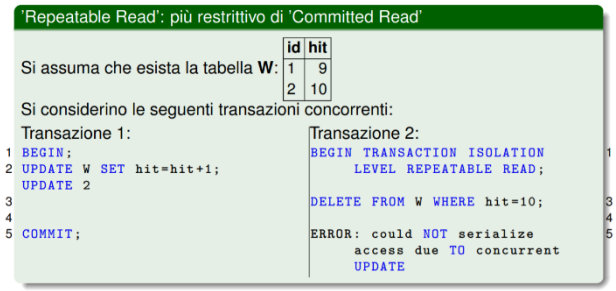
Cosa fare in caso di failure.
È fondamentale prevedere e gestire gli errori di concorrenza che possono occorrere con livello di isolamento Repeatable Read o Serializable. Se una transizione fallisce, il codice di fallimento è SQLSTATE = '40001'. SQLSTATE è una variabile di ambiente che si può leggere sia da programmi interni sia da programmi esterni. In caso di errore, si deve semplicemente ritentare l'esecuzione della transazione. Il costo computazionale di rieseguire una transazione è solitamente meno oneroso di quello che si avrebbe se si gestissero le transazioni usando i lock espliciti.

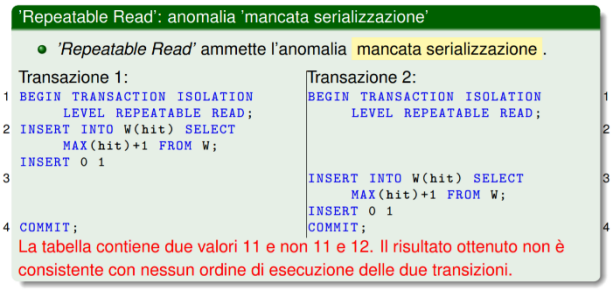
Lock espliciti [cap. 13.3].
PostgreSQL permette di attivare lock espliciti di tabelle e anche di righe di tabelle. I lock espliciti dovrebbero essere gestiti a livello di applicazione quando il modello MVCC non garantisce il comportamento richiesto (soprattutto a livello di prestazioni).

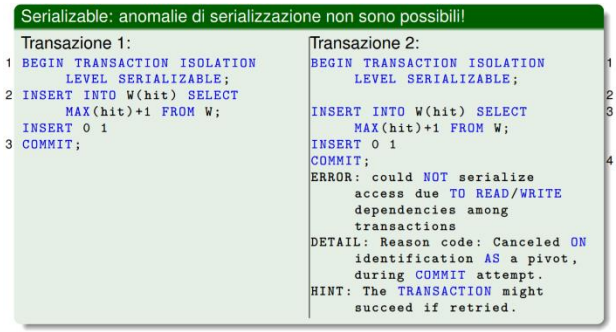
Esempi di anomalie.











➢ Accesso ad una base di dati PostgreSQL – *programmi Python e Java*

Python e database.
DB-API v2.0 è la Application Program Interface (API) ufficiale (Python Enhancement Proposals, PEP 249) che descrive come un modulo Python deve accedere a una base di dati esterna. Diversi gruppi di sviluppo hanno reso disponibili moduli (librerie) DB-API per diversi tipi di DBMS. A questa pagina c'è l'elenco aggiornato dei moduli disponibili. Per PostgreSQL ci sono molte implementazioni diverse di DB-API v2.0.

Introduzione a psycopg2.
Psycopg2 è una libreria scritta quasi completamente in C che implementa DB-API v2.0 mascherando la libreria C ufficiale libpq del gruppo di PostgreSQL. Implementa i cursori lato client e lato server, le comunicazioni asincrone, le notifiche e il comando COPY. Molti tipi di dati Python sono supportati e mappati nei tipi di dati PostgreSQL. La mappatura dei tipi di dati può essere personalizzata in modo semplice.

Fondamenti di DB-API v2.0 – connessione.
L'accesso a un database avviene tramite un oggetto di tipo Connection. Il metodo connect(...) accetta i parametri necessari per la connessione e ritorna un oggetto Connection:

```
connector=psycopg2.connect(host="…", database="…", user="…", password="…")
```

Metodi principali della classe Connection.

- `cursor()`: ritorna un cursore della base di dati. Un oggetto cursore permette di inviare comandi SQL al DBMS e di accedere al risultato del comando restituito dal DMBS.
- `commit()`: registra la transazione corrente. Normalmente una connessione apre una transazione al primo invio di comandi. Se non si esegue un commit prima di chiudere, tutte le eventuali modifiche/inserimenti vengono persi.
- `rollback()`: abortisce la transazione corrente.
- `close()`: chiude la connessione corrente. Implica un rollback automatico delle operazioni non registrate.
- `autocommit`: proprietà r/w. Se True, ogni comando inviato è una transazione isolata. Se False (default) il primo comando inviato inizia una transazione, che deve essere chiusa con commit o rollback.
- `readonly`: proprietà r/w. Se True, nella sessione non si possono inviare comandi di modifica dati. Il default è False.
- `isolation_level`: proprietà r/w. Modifica il livello di isolamento per la prossima transazione. Valori leciti: 'READ UNCOMMITTED', 'READ COMMITTED', 'REPEATABLE READ', 'SERIALIZABLE', 'DEFAULT'. Meglio assegnare questa variabile subito dopo la creazione della connessione.

Fondamenti di DB-API v2.0 – cursore.
Un cursore gestisce l'interazione con la base di dati: mediante un cursore è possibile inviare un comando SQL e accedere all'esito e ai dati di risposta del comando. Di seguito i metodi principali della classe cursore.

- `execute(comando, parametri)`: prepara ed esegue un 'comando' SQL usando i 'parametri', i quali devono essere passati come tupla o come dict. Il comando ritorna None. Eventuali risultati di query si devono recuperare con un comando di tipo fetch.
- `executemany(comando, parametri)`: prepara ed esegue un 'comando' SQL per ciascun valore presente nella lista 'parametri'. Per come è attualmente implementato, executemany è meno efficiente di un ciclo for con execute o, meglio ancora, di un unico INSERT con più tuple.
- `fetchone()`: ritorna una tupla della tabella risultato. Si può usare dopo un execute. Se non ci sono tuple, ritorna None.

- `fetchmany(<numero>)`: ritorna una lista di tuple della tabella risultato di lunghezza max <numero>. Si può usare dopo un execute. Se non ci sono tuple, ritorna una lista vuota.
- Dopo un execute, il cursore è un iterabile sulla tabella risultato. È possibile quindi accedere alle tuple del risultato anche con un ciclo del tipo `for record in cur`.
- `rowcount`: campo di sola lettura. Indica il numero di righe prodotte dall'ultimo comando. -1 indica che non è possibile determinare il valore.
- `statusmessage`: campo di sola lettura. Contiene il messaggio ritornato dall'ultimo comando eseguito.

I comandi di tipo execute accettano '%s' come indicatore di posizione parametro. La conversione dal tipo Python al dominio SQL è automatica per tutti i tipi fondamentali. Per maggiori dettagli si veda questa pagina.

SQL Injection.

Alcuni pensano sia più efficiente e facile scrivere:

```
cur.execute("SELECT 1 FROM users WHERE name='" +user+ "' AND pw='" +pw+ "'")
```

anziché:

```
cur.execute("SELECT 1 FROM users WHERE name=%s AND pw=%s", (user, pw))
```

dove user e pw sono letti da input. Altri, più intelligenti, possono sfruttare questa ingenuità per fare altro. Infatti, assegnando `' OR TRUE --` a user e un qualsiasi carattere a pw il risultato è:

```
SELECT 1 FROM users WHERE name='' OR TRUE --' AND pw='a'
```

Ancora peggio, assegnando `''; DROP TABLE users CASCADE; --` a user e un qualsiasi carattere a pw il risultato è:

```
SELECT 1 FROM users WHERE name=''; DROP TABLE users CASCADE; --' AND pw='a'
```

Schema per usare psycopg2.

Lo schema tipico di un modulo Python che comunica con un DBMS PostgreSQL via psycopg2 deve:

1. Aprire una connessione tramite `conn = psycopg2.connect(...)`.
2. Eventualmente modificare le proprietà di livello di isolamento, auto-commit, readonly.
3. Creare un cursore tramite `cur = conn.cursor()`.
4. Eseguire le operazioni previste.
5. Se la sessione non è in auto-commit, eseguire un `conn.commit()` se si sono dati comandi SQL di aggiornamento per registrare le modifiche (o `conn.rollback()` per annullarle).
6. Chiudere il cursore tramite `cur.close()` e la connessione tramite `conn.close()`.

Dalla versione 2.5 della libreria, la gestione dei close e dei commit è semplificata se si usa il costrutto with:

- Quando si usa una connessione con il with, all'uscita del blocco viene fatto un commit automatico e la connessione non viene chiusa.
- Quando si usa/crea un cursore con il with, all'uscita del blocco viene fatto un close automatico del cursore.

Connessioni e cursori.

Aprire una connessione costa in tempo (e in spazio). Meglio quindi aprire/chiudere poche connessioni in un'esecuzione. Con un oggetto connessione, infatti, si possono creare più cursori che condividono la stessa connessione. Psycopg2 garantisce solo che le istruzioni inviate dai cursori vengono sequenzializzate e quindi non si possono gestire transazioni concorrenti usando questo meccanismo. Regola pratica: usare più cursori sulla medesima connessione quando si fanno transazioni in auto-commit o solo transazioni di sola lettura.

Java e database.

Java DataBase Connectivity (JDBC) è la Application Program Interface (API) ufficiale che descrive come un programma Java deve accedere a una base di dati esterna. Diversi gruppi di sviluppo hanno reso disponibili librerie JDBC per diversi tipi di DBMS. A questa pagina c'è l'elenco aggiornato delle società che forniscono driver JDBC riconosciuti dalla Oracle. Per PostgreSQL esiste la libreria JDBC-42.

Fondamenti di JDBC.

Un programma Java che vuole accedere a una base di dati PostgreSQL deve:

1. Caricare il driver con l'istruzione `Class.forName(...)`. In questo modo, il driver viene registrato dentro la classe DriverManager.
2. Accedere al database mediante la creazione di un oggetto di tipo Connection. Il metodo `DriverManager.getConnection(<uri>,<user>,<pw>)` accetta i parametri necessari per la connessione e ritorna un oggetto Connection.

Metodi fondamentali della classe Connection.

- `createStatement()`: ritorna un oggetto Statement, che permette di inviare query statiche.
- `prepareStatement(<query>)`: ritorna un oggetto PreparedStatement, che rappresenta la query ma che permette di re-inviare la stessa più volte anche con parametri diversi.
- `commit()`: registra la transazione corrente. Normalmente in JDBC le connessioni sono in auto-commit, quindi un commit è sempre eseguito automaticamente dopo ogni esecuzione di comando.
- `rollback()`: abortisce la transazione corrente.
- `close()`: chiude la connessione corrente.

Classe PreparedStatement.

Un oggetto di tipo Statement è sufficiente per inviare query semplici senza parametri. Un oggetto di tipo PreparedStatement è da preferire quando una stessa query deve essere riusata più volte con parametri diversi o anche, più semplicemente, quando non si vuole fare la conversione esplicita dei valori dei parametri da Java nei corrispondenti SQL. Di seguito i principali metodi di tale oggetto.

- Un oggetto di tipo PreparedStatement è creato solitamente con il comando `prepareStatement(<query>)` di Connection, dove <query> è un comando SQL che contiene il carattere '?' in ogni posizione dove deve essere inserito un valore.
- I valori vengono inseriti invocando dei metodi `set<tipo>(<indice>, <valore>)` di PreparedStatement. Ci sono metodi per tutti i tipi supportati da PostgreSQL.
- La query viene inviata con il comando `executeQuery()` se interroga (restituisce un oggetto di tipo ResultSet che contiene lo stato e l'eventuale tabella risultato della query) o `executeUpdate()` se aggiorna, inserisce o cancella (restituisce il numero di righe che sono state modificate).
- Nel caso in cui la query restituisce una tabella, l'oggetto ResultSet è un cursore sulla tabella risultato. Il metodo `next()` di ResultSet posiziona il cursore alla prossima riga non letta della tabella e restituisce vero se esiste, falso altrimenti.
- I metodi `get<tipo>(<indice>)` e `get<tipo>(<nome>)` di ResultSet permettono di recuperare il valore della colonna identificata da <indice> o <nome> dalla riga corrente.

➢ Approfondimenti

NULL, booleani e operatori.

Se si confronta (operatori =,<,>,<>,>=,<=) un valore con NULL, si ottiene NULL:

- SELECT 1 <> NULL; → NULL
- SELECT NULL = NULL; → NULL
- SELECT NULL <> NULL; → NULL

Se si combina il NULL con operatori logici, se il valore dell'espressione finale dipende da NULL essa è NULL, altrimenti essa è pari al valore che si ottiene indipendentemente dalla presenza di NULL:

- SELECT NOT NULL → NULL
- SELECT TRUE AND NULL → NULL
- SELECT TRUE OR NULL → TRUE
- SELECT FALSE AND NULL → FALSE
- SELECT FALSE OR NULL → NULL

Vediamo vari confronti con operatori di tipo ANY:

`select *`
`from (values (1), (null)) as t(col1)`
`where t.col1 >= any (`
`values (1), (null)`
`);`
	Quando t.col1 è 1, ANY trova che 1>=1 è vero, quindi si ferma e restituisce vero.
	Risultato: una tabella con 1 riga, 1 colonna. Valore 1.

`select *`
`from (values (1), (null)) as t(col1)`
`where t.col1 > any (`
`values (1), (null)`
`);`
	Quando t.col1 è 1, ANY trova che 1>1 è false, 1>NULL è NULL, quindi restituisce NULL.
	Analogo quando t.col1 è NULL.
	Risultato: una tabella VUOTA.

`select *`
`from (values (1), (null)) as t(col1)`
`where t.col1 <> any (`
`values (null::int), (null)`
`);`
	Qui t.col1 viene confrontato con NULL e con NULL.
	Se t.col1 è 1, 1<>NULL è NULL, 1<>NULL è NULL, quindi ANY restituisce NULL.
	Analogo quando t.col1 è NULL.
	Risultato: una tabella VUOTA.

`select *`
`from (values (1), (null)) as t(col1)`
`where t.col1 <> any (`
`select 1 where 1=2`
`);`
	La sottoquery crea una tabella vuota.
	ANY, per definizione, restituisce sempre falso.
	Risultato: una tabella VUOTA.

Politiche di reazione.

Una violazione di un vincolo di chiave esterna, a differenza degli altri vincoli, non genera necessariamente un errore. È infatti possibile stabilire diverse politiche per reagire ad una violazione. La violazione può avvenire in due modi:

- Nella tabella secondaria, inserisco una nuova riga o modifico la chiave esterna. Si noti che la cancellazione di una riga dalla tabella secondaria non viola mai il vincolo di chiave esterna.
- Nella tabella principale, cancello una riga o modifico la chiave riferita. Si noti che l'inserimento di una nuova riga dalla tabella principale non viola mai il vincolo di chiave esterna.

Nel primo caso viene sempre generato un errore. Nel secondo posso stabilire delle politiche di reazione. Per l'operazione di modifica vi sono le seguenti politiche:

- CASCADE, il nuovo valore dell'attributo della tabella principale viene riportato su tutte le corrispondenti righe della tabella secondaria.
- SET NULL, alla chiave esterna della tabella secondaria viene assegnato il valore nullo al posto del valore modificato nella tabella principale.
- SET DEFAULT, alla chiave esterna della tabella secondaria viene assegnato il corrispondente valore di default al posto del valore modificato nella tabella principale.
- NO ACTION, nessuna azione viene intrapresa e viene generato un errore.

Per l'operazione di cancellazione posso reagire come segue:

- CASCADE, le corrispondenti righe della tabella secondaria vengono cancellate.
- SET NULL, alla chiave esterna della tabella secondaria viene assegnato il valore nullo al posto del valore cancellato nella tabella principale.
- SET DEFAULT, alla chiave esterna della tabella secondaria viene assegnato il corrispondente valore di default al posto del valore cancellato nella tabella principale.
- NO ACTION, nessuna azione viene intrapresa e viene generato un errore.

La sintassi per specificare queste politiche usa i costrutti ON UPDATE e ON DELETE. Come regola generale, per le modifiche si usa la politica ON UPDATE CASCADE. Per le cancellazioni, invece, si usa la politica ON DELETE CASCADE per chiavi esterne di tabelle che corrispondono a relazioni concettuali oppure ad entità deboli e la politica ON DELETE SET NULL negli altri casi. La ragione è che nel primo caso vi è un forte collegamento tra la tabella master e la tabella slave e dunque una cancellazione nella tabella master dovrebbe provocare corrispondenti cancellazioni nella tabella slave.

Cast [cap. VI.I-CREATE CAST].

Un cast può essere invocato solo tramite una richiesta esplicita del tipo:

- `CAST(x AS typename)`
- `x::typename`

Funzioni e operatori per le stringhe [cap. 9.4].

Function	Return Type	Description	Example	Result
string `||` *string*	text	String concatenation	`'Post' || 'greSQL'`	PostgreSQL
string `||` *non-string* or *non-string* `||` *string*	text	String concatenation with one non-string input	`'Value: ' || 42`	Value: 42
`bit_length(`*string*`)`	int	Number of bits in string	`bit_length('jose')`	32
`char_length(`*string*`)` or `character_length(`*string*`)`	int	Number of characters in string	`char_length('jose')`	4
`lower(`*string*`)`	text	Convert string to lower case	`lower('TOM')`	tom
`octet_length(`*string*`)`	int	Number of bytes in string	`octet_length('jose')`	4
`overlay(`*string* `placing` *string* `from int [for int])`	text	Replace substring	`overlay('Txxxxas' placing 'hom' from 2 for 4)`	Thomas
`position(`*substring* `in` *string*`)`	int	Location of specified substring	`position('om' in 'Thomas')`	3
`substring(`*string* `[from int] [for int])`	text	Extract substring	`substring('Thomas' from 2 for 3)`	hom
`substring(`*string* `from` *pattern*`)`	text	Extract substring matching POSIX regular expression. See Section 9.7 for more information on pattern matching.	`substring('Thomas' from '...$')`	mas
`substring(`*string* `from` *pattern* `for` *escape*`)`	text	Extract substring matching SQL regular expression. See Section 9.7 for more information on pattern matching.	`substring('Thomas' from '%#"o_a#"_' for '#')`	oma
`trim([leading | trailing | both] [`*characters*`] from` *string*`)`	text	Remove the longest string containing only characters from *characters* (a space by default) from the start, end, or both ends (`both` is the default) of *string*	`trim(both 'xyz' from 'yxTomxx')`	Tom
`trim([leading | trailing | both] [from]` *string* `[,` *characters*`] )`	text	Non-standard syntax for `trim()`	`trim(both from 'yxTomxx', 'xyz')`	Tom
`upper(`*string*`)`	text	Convert string to upper case	`upper('tom')`	TOM

Funzioni e operatori per le date [cap. 9.9].

Function	Return Type	Description	Example	Result
`age(timestamp, timestamp)`	`interval`	Subtract arguments, producing a "symbolic" result that uses years and months, rather than just days	`age(timestamp '2001-04-10', timestamp '1957-06-13')`	`43 years 9 mons 27 days`
`age(timestamp)`	`interval`	Subtract from `current_date` (at midnight)	`age(timestamp '1957-06-13')`	`43 years 8 mons 3 days`
`clock_timestamp()`	`timestamp with time zone`	Current date and time (changes during statement execution); see Section 9.9.4		
`current_date`	`date`	Current date; see Section 9.9.4		
`current_time`	`time with time zone`	Current time of day; see Section 9.9.4		
`current_timestamp`	`timestamp with time zone`	Current date and time (start of current transaction); see Section 9.9.4		
`date_trunc(text, timestamp)`	`timestamp`	Truncate to specified precision; see also Section 9.9.2	`date_trunc('hour', timestamp '2001-02-16 20:38:40')`	`2001-02-16 20:00:00`
`date_trunc(text, interval)`	`interval`	Truncate to specified precision; see also Section 9.9.2	`date_trunc('hour', interval '2 days 3 hours 40 minutes')`	`2 days 03:00:00`
`extract(field from timestamp)`	`double precision`	Get subfield; see Section 9.9.1	`extract(hour from timestamp '2001-02-16 20:38:40')`	`20`
`extract(field from interval)`	`double precision`	Get subfield; see Section 9.9.1	`extract(month from interval '2 years 3 months')`	`3`
`isfinite(date)`	`boolean`	Test for finite date (not +/-infinity)	`isfinite(date '2001-02-16')`	`true`
`isfinite(timestamp)`	`boolean`	Test for finite time stamp (not +/-infinity)	`isfinite(timestamp '2001-02-16 21:28:30')`	`true`
`isfinite(interval)`	`boolean`	Test for finite interval	`isfinite(interval '4 hours')`	`true`
`justify_days(interval)`	`interval`	Adjust interval so 30-day time periods are represented as months	`justify_days(interval '35 days')`	`1 mon 5 days`
`justify_hours(interval)`	`interval`	Adjust interval so 24-hour time periods are represented as days	`justify_hours(interval '27 hours')`	`1 day 03:00:00`
`justify_interval(interval)`	`interval`	Adjust interval using `justify_days` and `justify_hours`, with additional sign adjustments	`justify_interval(interval '1 mon -1 hour')`	`29 days 23:00:00`
`make_date(year int, month int, day int)`	`date`	Create date from year, month and day fields	`make_date(2013, 7, 15)`	`2013-07-15`