

Basi di dati a oggetti

Integrano la tecnologia delle basi di dati con il paradigma a oggetti proprio dell'ingegneria del software. Ogni entità del mondo reale può essere ricondotta ad un oggetto. Sono emerse due correnti principali:

- **OODBMS** (Object-Oriented DBMS): approccio rivoluzionario, basi di dati ad oggetti
- **ORDBMS** (Object-Relational DBMS): approccio evolutivo, sistemi relazionali a oggetti (estendendo quindi il modello relazionale)

I database ad oggetti consentono di specificare strutture dati complesse con relazioni semantiche, di descrivere in modo integrato dati ed operazioni e permettono un'integrazione stretta con i linguaggi di programmazione OO.

Tipi: consentono la definizione delle proprietà degli oggetti (statiche e dinamiche), utilizzando costruttori di tipo, insiemi di dati atomici o tipi enumerativi. Ogni definizione associa un nome (etichetta) ad un tipo : *Indirizzo : string*

I tipi di dato complessi adottano una definizione di tipo ricorsivo. Consentono di modellare la complessità del mondo reale.

Record: è un tipo tale da contenere tuple di valori complessi A_i di tipo T_i **T=record-of($A_1:T_1, \dots, A_n:T_n$)**

- **Set:** collezioni non ordinate senza duplicati (tipo **set-of($A_1:T_1, \dots, A_n:T_n$)**)
- **Bag:** collezioni non ordinate con duplicati (tipo **bag-of($A_1:T_1, \dots, A_n:T_n$)**)
- **List:** collezioni ordinate (tipo **list-of($A_1:T_1, \dots, A_n:T_n$)**)

Un oggetto di tipo T è un **istanza** di T. I costruttori di tipo possono essere applicati in modo arbitrario per ottenere oggetti di complessità arbitraria. Generalmente una definizione di tipo inizia con un costruttore record-of. Oggetto x di tipo $T = \text{record-of}(A_1:T_1, \dots, A_n:T_n)$, allora A_1, \dots, A_n sono detti **proprietà** di x . Si accede alle proprietà con la **dot notation**.

In questo modo l'oggetto non viene spezzato in più tabelle come nel modello relazionale. Per evitare ridondanza in oggetti annidati si utilizzano i riferimenti tra oggetti (OID), associabili alle proprietà object-valued, che consentono la condivisione di oggetti da parte di altri oggetti tramite i riferimenti.

La parte strutturale di un oggetto è una coppia (OID, valore). Il valore è lo *stato* dell'oggetto. OID garantisce individuazione univoca di un oggetto.

I riferimenti sono analoghi ai puntatori dei linguaggi di programmazione e alle chiavi esterne del modello relazionale. Tuttavia:

1. Non possono essere corrotti (vengono invalidati automaticamente dal sistema)
2. I valori non sono visibili dall'utente in modo diretto
3. Modificando il valore di un oggetto referenziato non si perde il riferimento stesso.

- **Identità:** gli oggetti hanno lo stesso identificatore ($O_1=O_2$)
- **Uguaglianza superficiale:** due oggetti devono avere lo stesso stato, cioè lo stesso valore per proprietà omologhe ($O_1==O_2$)
- **Uguaglianza profonda:** due oggetti devono avere gli stessi valori sostituendo ricorsivamente gli oggetti raggiungibili tramite OID agli OID stessi, ma non necessariamente lo stesso

stato (01===02)

Classe: raccoglie oggetti dello stesso tipo (contenitore di oggetti, dinamicamente aggiunti o tolti alla classe). Formata da **interfaccia** ed **implementazione**. Nel DB l'incapsulamento non è esattamente tale nell'accezione tradizionale del termine, perchè normalmente le strutture dati sono visibili (e lo stato è quindi esposto).

I tipi descrivono stato ed il comportamento, proprietà astratte, per ogni classe ho un solo tipo. Le classi descrivono rappresentazione ed implementazione degli oggetti, la realizzazione delle proprietà ed inoltre possono corrispondere più classi ad un solo tipo.

Metodi: sono utilizzati per manipolare gli oggetti di un OODBMS: **costruttori**, **distruttori**, **accessori**, **trasformatori** più altri metodi per esigenze applicative specifiche. Può esserci la distinzione public/private.

E'importante permettere il riuso del codice (grazie alle caratteristiche peculiari del modello Object Oriented). Generalmente un metodo è associato ad una sola classe (anche se alcuni sistemi premettono metodi multi-target).

Mismatch d'impedenza: nei DBMS relazionali c'è un mismatch tra i linguaggi applicativi (che manipolano variabili scalari) ed SQL che estrae insiemi di n-uple (si risolve usando i cursori). Negli ODBMS il problema è risolto in quanto gli oggetti possono essere manipolati direttamente dai metodi.

Classe: implementazione di un tipo. **Estensione:** è una collezione di oggetti aventi lo stesso tipo.

Gerarchia ed ereditarietà: lo stesso dei linguaggi OO, tuttavia gli oggetti di un DBMS tipicamente sono persistenti (a differenza degli oggetti di un linguaggio OO) e bisogna fare attenzione. Infatti durante la sua vita, un oggetto può cambiare tipo: bisogna introdurre operazioni di **specializzazione** (si diventa membri di una sottoclasse) e di **generalizzazione** (si perde l'appartenenza ad una sottoclasse). In alcuni sistemi una classe può ereditare da più classi (ereditarietà multipla). Ereditarietà multipla può generare conflitti di nome (due o più sopra-classi posseggono proprietà/metodi con lo stesso nome).

Gli oggetti possono essere **temporanei** oppure **persistenti**.

Overriding: ridefinizione di metodi (overloading, late binding). La modifica dell'interfaccia dei metodi è un punto critico: ridefinendo metodi di una sottoclasse, si possono ridefinire i suoi parametri in due modi: **co-variante** (parametri sono sotto-tipi dei parametri di superclasse) oppure **contro-variante** (i parametri sono sopra-tipi dei parametri di superclasse).

Manifesto degli OODBMS:

• Golden rules

1. dati derivati, viste
2. funzionalità DBA
3. Vincoli di integrità
4. Funzionalità per la modifica schemi

• Opzionali

1. Ereditarietà multipla
2. Verifica tipi e inferenza

- 3. Distribuzione
- 4. Design transactions
- 5. Gestione delle versioni
- **Scelte libere**
 - 1. Paradigma di programmazione
 - 2. Realizzazione
 - 3. Sistema di tipi
 - 4. Uniformità

Standard ODMG (**Object Database Management Group**): nato negli anni 80 per permettere la nascita di uno standard che accelerasse lo sviluppo dei sistemi database a oggetti.

ODL (Object Definition Language): descrive tipi (non classi, infatti ad un tipo possono corrispondere più classi che lo implementano), è indipendente dal linguaggio di programmazione scelto. I riferimenti tra tipi sono detti *relationship* e sono bidirezionali (per cui viene sempre definita la relazione inversa). Vengono descritte anche le interfacce dei metodi (e relativi parametri o la clausola *raises* che indica quando lanciare un'eccezione).

OQL (Object Query Language): è considerato un'estensione di SQL, non comprende primitive per la modifica dello stato degli oggetti (se ho incapsulamento forte infatti questo deve avvenire solo tramite metodi pubblici forniti dalle classi, richiamabili da OQL).

Esempi di query OQL

- **select distinct x.Targa from x in automobile where x.colore="rosso"**
"Estrarre le targhe delle automobili rosse" Restituisce struttura di tipo **set(string)**
- **select x.Targa from x in AutoSportivaStorica where x.colore="rosso" and "MilleMiglia34" in x**
"Estrarre le targhe delle auto rosse che hanno vinto la MilleMiglia34" L' ereditarietà permette di invocare proprietà delle super classi
- **select x.targa from x in AutoSportivaStorica where x.Pilota.Nome = "Fangio" and "Maranello" in x.Costruttore.Stabilimenti.Nome**
"Estrarre le targhe delle auto storiche costruite a Maranello e pilotate da Fangio"
 Utilizzo delle *path-expressions* in qualunque espressione dove compaia una proprietà di un oggetto.
- **Select x.Pilota.Nome from x in AutoSportivaStorica where x.Pilota = x.Costruttore.Presidente**
"Estrarre le persone che siano nel contempo piloti e costruttori delle stesse auto sportive"
 Risultato di tipo **bag(String)**. Si richiede identità della persona, non l'uguaglianza del nome!
- **Select a.targa from a in autoSportivaStorica, c in Costruttore, s in Stabilimento where c=a.Costruttore and s in c.Stabilimenti and s.città<>"Maranello" and c.nome="Ferrari" and a.MaxVelocità > 250**
"Estrarre le auto sportive Ferrari non costruite a Maranello e che superano i 250 Km/h"
 Introduzione di predicati che legano le variabili, utilizzando più di una variabile nella clausola *from*.

- **Select distinct struct**(Mod: x.Modello, Col: x.colore) from x in AutoSportivaStorica, where "MilleMiglia39" in x.GareStoricheVinte
"Estrarre distinti modelli e colori delle auto sportive vincitrici della MilleMiglia39"
 Risultato di tipo **set(record(string,string))**
- **Select distinct struct**(Nome: x.Costruttore.Nome, Stab: (Select struct (Cit: y.città, Add:y.Addetti) from y in Stabilimento where y in x.Costruttore.Stabilimenti)) from x in AutoSportivaStorica where x.Prezzo > 100.000
"Estrarre il nome dei costruttori che vendono auto sportive ad un prezzo superiore ai 100.000 euro: per ognuno elencare le città e il numero di addetti degli stabilimenti"
 Utilizzo di clausole select più complesse (es. Select annidate).
- **select count (select distinct x.Modello from x in (select y from y in Automobile, z in Costruttore where z=y.Costruttore and sum(z.Stabilimenti.Addetti)>4500))**
*"Estrarre il numero dei modelli delle auto costruite da costruttori che hanno un numero globale di addetti, nei vari stabilimenti, superiori a 4.500 unità"*E'possibile inserire clausole select anche in clausole from o in funzioni aggregate (count, min, max, avg)
- **sort x in Automobile by x.Targa**
"Ordinare le automobili per targa"
- **group a in Automobile by (Costr: a.Costruttore) with (NumeroAuto: count (select x from x in partition))**
"Estrarre il numero di auto partizionate in base al loro costruttore"
 Utilizzo della parola chiave **partition** che denota ciascuna partizione ottenuta tramite la clausola **group ... by** . Tipo del risultato: **set (struct(string, set(OID),integer))**
- **group a in Autosportiva by (veloci: a.MaxVelocità < 200, rapide: a.MaxVelocità >= 200 and a.MaxVelocità<250, super: a.MaxVelocità>=250)**
"Classificare le auto sportive in veloci, rapide e super a seconda della massima velocità raggiunta" Il partizionamento avviene tramite predicati di partizione generici. Il risultato è un insieme di record con n+1 attributi (n=numero di partizioni): i primi n campi sono booleani falsi, tranne quello che indica l'appartenenza all'n-esima partizione. L'n-esimo attributo è un set contenente gli oggetti che fanno parte di ciascuna partizione. Il risultato è di tipo **Set (struct(boolean,boolean,boolean, set(OID))**
- **select struct (veloci: x.veloci, rapide: x.rapide, super: x.super, tot:count(x.partition)) from x in (group a in AutoSportiva by (veloci: a.MaxVelocità < 200, rapide: a.MaxVelocità >= 200 and a.MaxVelocità < 250, super: a.MaxVelocità >= 250))**
 A partire dall'esempio precedente, si applica una funzione aggregata che calcoli il numero di elementi presenti in ciascuna partizione.

SQL-3 (SQL: 1999): è la versione più recente di SQL, estensione compatibile di SQL-2. Consente di esprimere la maggior parte dei concetti esprimibili con OODBMS.

1. Definire tabelle

2. Definire tipi: **tipi ennupla con struttura anche complessa, gerarchie**. Utilizzabili per definire tabelle con lo stesso schema, come componenti e nell'ambito di relationship, oppure **tipi astratti**.

I **tipi ennupla** sono gli oggetti, le **relazioni** sono le classi. Gli identificatori sono manipolabili ed è possibile utilizzare i riferimenti e/o incorporare oggetti. E'presente

l'ereditarietà (parola chiave **under**), è permessa la definizione di **tipi astratti** (utilizzabili per i singoli attributi, e possono avere funzioni associate).

Interrogazioni: è possibile:

- seguire i riferimenti
 - citare gli OID (se visibili)
 - accedere alle strutture interne
 - nidificare (**group by**) e denidificare (unnest)
 - accedere ad attributi tramite **dot notation**
 - Dereferenziazione per attributi di oggetti referenziati con l'operatore ->
- ** (basi di dati multimediali .. finire)
-

Basi di dati attive (triggers)

Una base di dati si definisce **attiva** quando dispone di un sottosistema integrato per definire e gestire regole di produzione (regole attive), seguenti il paradigma *Evento-Condizione-Azione*. Ciascuna regola infatti:

1. Reagisce ad alcuni *eventi*
2. Valuta una *condizione*
3. In base al valore di verità scarituro da 2) esegue una *reazione*

In questo modo si può codificare nella base di dati una parte dell'applicazione normalmente codificata tramite programmi esterni. Realizzo così indipendenza della conoscenza, infatti la conoscenza di tipo reattivo viene sottratta ai programmi applicativi e codificata sotto forma di regole attive (creando una logica condivisa). Questa dimensione di indipendenza si aggiunge all' *indipendenza fisica* e *logica* già proprie di un sistema DBMS.

Definizione dei trigger: fa parte del DDL, anche se si possono attivare / disattivare dinamicamente in base ad alcune condizioni. Paradigma ECA:

1. **eventi:** primitive di manipolazione dati (*insert, delete, update*)
2. **condizione:** predicato booleano (espresso in SQL)
3. **azione:** sequenza generica di primitive SQL (eventualmente arricchita da linguaggio di programmazione)

Generalmente i trigger:

- si riferiscono ad una tabella (target)
- vengono **attivati da un evento**, **valutati durante la verifica** condizione ed **eseguiti a seguito di valutazione positiva**
- Hanno due livelli di granularità, di tupla (row-level) e di primitiva (statement-level)
- Possono essere in modalità **immediata** (valutati immediatamente prima **-before-** oppure dopo **-after-** l'elemento che li ha attivati) oppure in modalità **differita** (a seguito di un comando **commit**).
- E'possibile l'attivazione **a cascata** dei trigger (l'azione di un trigger è anche l'evento di un altro trigger). Bisogna evitare un'attivazione ciclica (infinita) dei trigger!

Comportamento dei trigger in SQL:1999 (SQL-3):

Ogni trigger è attivato da un solo evento (primitiva di modifica dati SQL). I trigger sono attivati immediatamente (before o after). Granularità di tupla o di primitiva. Sintassi di definizione simile a quella adottata da DB2. E' tuttavia presente un'indicazione esplicita per le azioni che constano di molti comandi SQL, inclusi tra le parole chiave **begin atomic** ed **end**. L'espressione **for each statement** può essere omessa in quanto è presa per default:

```
create trigger NomeTrigger Modo Evento on TabellaTarget  
[referencing Referenza]  
for each Livello when (predicatoSQL)  
StatementProceduraleSQL
```

La clausola **referencing** permette di introdurre nomi di variabili. **Old** e **new** sono variabili definite implicitamente che indicano il BeforeState e l'AfterState delle tuple che subiscono

la modifica.

Il comportamento è praticamente identico a quello DB2:

- I trigger **before** vengono usati solo per determinare errori e modificare valori assegnati alle variabili new. Non possono contenere comandi DML che modifichino lo stato della base di dati, per cui non possono attivare altri trigger. Bisogna garantire che l'effetto dei before trigger sia precedente a quello della primitiva che li genera (anche se il trigger può valutare anticipatamente tali risultati tramite i valori **new**).
- Vari trigger a diversa granularità possono riferirsi allo stesso evento: vengono considerati in base ad un ordinamento implicito gestito dal sistema. I trigger a livello di tupla sono possono essere ordinati in modo arbitrario.
- La valutazione dei trigger avviene in modo preciso:
 1. A seguito di una primitiva **S** vengono valutati ed eseguiti i *before trigger* (i quali possono modificare i valori *new*)
 2. Vengono svolte le azioni legate al ripristino dell'integrità referenziale, le quali possono attivare molti trigger che si aggiungono agli *after trigger* attivati da **S**
 3. Il sistema considera ed esegue tutti i trigger in base alla loro priorità. Se l'esecuzione provoca l'attivazione di altri trigger, lo stato d'esecuzione dell'algoritmo viene salvato e il sistema reagisce in modo ricorsivo; al termine si ripristina lo stato e l'esecuzione riprende dal punto in cui era stata sospesa.

Caratteristiche evolute dei Triggers

- Gli eventi possono includere **eventi temporali** o **applicativi**;
- L'attivazione può dipendere da **generiche espressioni booleane di eventi** costruite usando operatori complessi (precedenze e congiunzione di eventi)
- Clausola **instead of**: l'azione viene eseguita *al posto* dell'evento. Poco usata
- Valutazione ed esecuzione **detached**: avvengono nel contesto di un'altra transazione (autonoma oppure coordinata con la transazione dove si è verificato l'evento)
- Conflitti tra regole (**conflict set**) risolto da priorità esplicite (ordinamenti parziali o totali definiti dall'utente)
- Regole organizzate in **gruppi** che possono essere attivati o disattivati a piacimento.

Proprietà dei trigger

Il comportamento collettivo dinamico dei trigger è complesso. Bisogna valutare proprietà quali **terminazione, confluenza e osservabilità deterministica**

- Un set di regole garantisce la **terminazione** quando, per ogni transazione che scatena l'esecuzione delle regole, tale esecuzione termina producendo uno stato finale (anche abort locale/globale)
- Un set di regole garantisce la **confluenza** quando, per ogni transazione che scatena l'esecuzione delle regole, tale esecuzione termina producendo un unico stato finale che non dipende dall'ordine di esecuzione delle regole non prioritizzate esplicitamente.
- Un set di regole garantisce l'**osservabilità deterministica** quando, per ogni transazione che scatena l'esecuzione delle regole, tale esecuzione è confluyente e tutte le azioni visibili

sono identiche e prodotte nello stesso ordine.

Di queste, solo la prima è essenziale per evitare cicli infiniti.

L'**analisi** delle regole consete di verificare, a tempo di creazione, che le proprietà sopracitate valgano per un dato set di regole. Si usa il **grafo di attivazione per determinare la terminazione: la presenza di cicli indica l'esistenza di una situazione sintattica di possibile mutua interazione** (non è garanzia di ciclo infinito!). In generale un ciclo può indicare la presenza di un trigger ricorsivo: per evitare la ciclicità infinita bisogna fare in modo che, prima o poi, l'esecuzione ciclica si blocchi perchè la condizione diventa falsa.

Classificazione dei trigger

- **regole interne:** sottosistema del DBMS per implementare alcune funzionalità. Molto spesso questi trigger sono generati dal sistema stesso e non sono visibili dall'utente (gestione vincoli d'integrità, computo di dati derivati, gestione di dati replicati, gestione di versioni, gestione privatezza e sicurezza dati, logging delle azioni e registrazione degli eventi)
- **regole esterne:** conoscenze di tipo applicativo non riconducibili a schemi predefiniti (tra cui le *business rules*)

Gestione integrità referenziale: bisogna esprimere i vincoli sotto forma di SQL. Il predicato corrisponderà alla *condizione* del trigger. Gli eventi che causano violazioni contribuiranno alla parte *eventi* del trigger. Bisogna poi stabilire che *azione* svolgere in caso di violazione di integrità referenziale.

XML e Xquery

XML: acronimo di eXtensible Markup Language. Metodo per rappresentare dati semistrutturati (cioè che rispettano in modo parziale il proprio schema o ne sono addirittura sprovvisti). Permette di definire **dati con un grado variabile di struttura** (rispettando semplici regole di ben formatezza oppure rispondendo ad uno schema preciso di definizione).

Rappresentazione di istanze come documenti contenenti dati e markup. Lo standard detta le regole sintattiche per la creazione di documenti ben formati, realizzati mescolando tag e contenuto informativo. Requisiti per la ben formatezza:

1. documento deve iniziare con una **direttiva standard** che specifica la versione di XML utilizzata, ad esempio `<?xml version="1.0"?>`
2. documento può contenere **elementi** contenenti testo, codice o altri elementi. Gli elementi sono racchiusi da **tag** `<...> </...>`. Gli elementi privi di contenuto hanno un solo tag `<.../>`
3. documento deve avere **elemento radice** che racchiude tutto. Bisogna garantire il corretto annidamento dei tag
4. i valori degli **attributi** degli elementi devono essere racchiusi dalle virgolette.

Namespace XML: una collezione di nomi (attributi, elementi ...) identificata da un URI (Universal Resource Identifier). E' un metodo per evitare conflitti di nome

I documenti possono essere accompagnati da una specifica della loro struttura (se il documento la rispetta, si dice **valido** rispetto a quella specifica). Ho due modelli di specifica: **DTD** (Document Type Definition) e **XML Schema**.

Il **DTD** detta i tag ammessi e le regole di annidamento dei tag (permette di **validare** i documenti XML secondo le regole in esso definite). Per cui un documento XML può essere **ben formato** (quando rispetta la sintassi XML) e **valido** (quando rispetta i vincoli aggiuntivi dettati da un DTD).

XML-Schema: proposto inizialmente da Microsoft, è diventato uno standard W3C. Il suo scopo è di definire un documento XML in modo più preciso rispetto al DTD. Un XML-Schema definisce regole riguardanti:

- Elementi e loro gerarchia
- Attributi
- Sequenza e cardinalità di elementi figli
- Tipi di dati e valori di default per elementi ed attributi

XML-Schema è un insieme di elementi XML standard per definire schemi di documenti detti XSD (XML Schema Definition) ovvero lo schema di un tipo di documenti.

Proprietà degli XSD:

1. Estendibili (tipi riusabili definiti dall'utente)
2. Rappresentabili come file XML
3. Più ricchi e completi dei DTD
4. Capaci di supportare tipi di dati diversi da PCDATA

5. Capaci di gestire namespace multipli

** Sintassi di DTD e XML-Schema

Interrogazioni dati in XML

Un singolo documento (o un insieme di documenti) sono considerati come depositi interrogabili di informazione. Esigenza di meccanismi di interrogazione e di memorizzazione persistente per dati XML.

Basi di dati XML. Suddivise principalmente in due famiglie:

1. XML-Native:

- Sfruttano tecnologie specifiche per XML per memorizzazione e indicizzazione
- Linguaggi di interrogazione specifici per XML (Xquery)
- Modello logico dei dati non-relazionale (DOM, XpathDM, XMLIS ...)
- Schemi proprietari di memorizzazione fisica (CLOB)
- Gestiscono tutte le caratteristiche sintattiche dei documenti
- Organizzano i dati in collezioni di documenti

2. Relazionali con supporto XML:

- Estensione del modello relazionale
- Sfruttano estensioni di SQL (SQL/XML)
- Memorizzazione interna sotto forma di tabelle
- Conversioni XML <--> Relational in ingresso e uscita
- Non conservano tutte le caratteristiche sintattiche di XML

Xquery: linguaggio definito da W3C per l'interrogazione di dati XML (simile a SQL)

Per “navigare” all'interno dei documenti XML si usano **espressioni Xpath**, ovvero una stringa contenente nomi di elementi ed operatori di navigazione/selezione. Una path expression può iniziare con **document(doc_pathname)** che restituisce l'elemento radice del documento specificato e tutto il suo contenuto. A partire da lì si specificano espressioni per selezionare il contenuto desiderato, usando gli operatori sottostanti.

- . nodo corrente
- .. nodo padre del nodo corrente
- / nodo radice o figlio del corrente
- // discendente del nodo corrente
- @ attributo del nodo corrente
- * qualsiasi nodo
- [] predicato
- [n] posizione

Esempi di utilizzo:

- Condizioni su elementi/attributi:
document("libri.xml")/Elenco/Libro[Editore="Bompiani"]/Titolo
"Insieme di tutti i titoli dei libri dell'editore Bompiani che sono disponibili in libri.xml"
- Ricerca di sotto-elementi a qualsiasi livello:
document("libri.xml")//Autore
"Insieme di tutti gli autori che si trovano nel documento libri.xml, annidati a qualunque livello"
- Condizione sull'ordine dei sottoelementi:
document("libri.xml")/Elenco/Libro[2]/*
"Insieme di tutti i sottoelementi contenuti nel secondo libro del documento libri.xml"

Un'interrogazione Xquery è un'espressione complessa che consente di estrarre parti di un documento XML e costruire un nuovo documento: si basa su quattro clausole (FLWOR):

1. **FOR**: iterazione
2. **LET**: collegare variabili
3. **WHERE**: esprimere predicati
4. **ORDER BY**: imporre un ordinamento
5. **RETURN**: generare il risultato (non presente in SQL)

Esempi:

- **FOR \$1 IN document("libri.xml")//Libro**
RETURN \$1
FOR valuta l'espressione che definisce \$1, e itera sull'insieme assegnando il nodo corrente alla variabile \$1
RETURN costruisce il risultato (in questo caso *"l'insieme di tutti i libri che si trovano in libri.xml"*)
- **FOR \$1 IN document("libri.xml")//Libro**
FOR \$a IN \$1/Autore
RETURN \$a
"Per ogni valore di \$1 (libro), per ogni valore di \$a (autore del libro corrente), inserisci nel risultato l'autore legato a \$a"
- **LET \$1 := document("libri.xml")//Libro**
RETURN \$1
LET permette di introdurre nuove variabili. Nel nostro caso, LET valuta l'espressione //Libro ed assegna a \$1 l'intero insieme di libri trovati
- **FOR \$1 IN document("libri.xml")//Libro**
WHERE \$1/Editore="Bompiani"AND \$1/@disponibilità="S"
RETURN \$1
La clausola WHERE esprime una condizione, solamente le tuple che soddisfano la condizione vengono utilizzate per invocare la clausola RETURN. Le condizioni possono contenere differenti predicati relazionati dalle parole chiave AND, OR, NOT. Nell'esempio precedente

“restituire l'insieme di tutti i libri pubblicati da Bompiani che sono disponibili”.

- **FOR \$1 IN document(“libri.xml”)//Libro[Editore=“Bompiani” AND @disponibilità=“S”]
RETURN \$1**
E'possibile omettere la clausola WHERE specificando una opportuna Path Expression
- La clausola RETURN genera l'output di una espressione FLWR (un nodo, una foresta ordinata di nodi o un valore). Può contenere costruttori di elementi, riferimenti a variabili ed espressioni annidate.
- **FOR \$1 IN document(“libri.xml”)//Libro
WHERE \$1/Editore=“Bompiani”
RETURN <LibroBompiani>\$1/Titolo</LibroBompiani>**
Esempio di ritorno di costruttore di elemento (Ritorna l'elemento<titolo> ed il suo contenuto)
- **RETURN <LibroBompiani>\$1/Titolo/text()</LibroBompiani>**
In questo caso alternativo viene estratto il contenuto PCDATA dell'elemento Titolo tramite l'operatore text()
- **FOR \$1 IN document(“libri.xml”)//Libro
ORDER BY &1/Titolo
RETURN <Libro>\$1/Titolo,\$1/Editore</Libro>**
I libri vengono ordinati rispetto al titolo
- **FOR \$e IN document(“libri.xml”)//Editore
LET \$1 := document(“libri.xml”)//Libro[Editore=\$e]
WHERE count(\$1)>100
RETURN \$e**
Funzioni aggregate: *“Restituire gli editori che compaiono in almeno 100 libri dell'elenco”*

Implementazione fisica dei dati

I dati in un DBMS sono organizzati a livelli: gli utenti si riferiscono al *livello logico*, mentre il DBMS opera sul *livello fisico* che identifica le particolari strutture di memorizzazione su dispositivi fisici da esso adottate. Queste caratteristiche sono ignorate dagli utenti grazie al principio di *indipendenza dei dati*.

Le basi di dati si affidano alla memoria secondaria per due motivi: **dimensioni** e **persistenza**. I dati in memoria secondaria possono essere utilizzati soltanto dopo essere stati trasferiti in memoria principale.

Memoria Secondaria: organizzata in blocchi di lunghezza fissa (~KB), operazioni di **lettura** o **scrittura** di una pagina (cioè dei dati di un blocco). Costo di accesso: mediamente > 10ms (4 ordini di grandezza superiore rispetto alle operazioni in memoria principale). Gli accessi a memoria secondaria sono il fattore limitante per le applicazioni I/O bound.

Per organizzare efficientemente i dati, si utilizzano particolari strutture fisiche di memorizzazione degli stessi. Sono codificate come *metodi d'accesso*, cioè come moduli software che forniscono primitive di manipolazione/estrazione dei dati. Principalmente ho tre tipi di strutture:

1. **sequenziali**
2. **basate su hashing**
3. **ad albero**

Ogni metodo d'accesso ha una differente organizzazione delle tuple nelle pagine. Casi

sequenziali e ad hashing:

- **block header/trailer** contengono informazioni di controllo per il *FileSystem*
- **page header/trailer** contengono informazioni di controllo per il metodo d'accesso
- **page dictionary**: puntatori all'inizio dei dati elementari in pagina
- **parte utile** contiene i dati (stack crescente contrapposto allo stack del dictionary)
- **bit di parità e verifica**

Primitive di gestione pagine:

1. **Inserzione/modifica tupla** (causa ristrutturazione se lo spazio è insufficiente)
2. **Cancellazione** (realizzata con flag d'invalidità)
3. **Accesso al campo x della tupla y**: identificata tramite offset e lunghezza del campo stesso, dopo aver identificato tupla tramite chiave o offset in pagina

Nelle **strutture sequenziali** le tuple sono memorizzate in sequenza:

- **entry-sequenced**: sequenza dipende dall'ordine di inserimento. Ottima per operazioni con località spaziale. Ottimo utilizzo dello spazio fisico. Non gestisce bene le ricerche e le modifiche che causano riorganizzazione.
- **array**: sono poste in sequenza e indicizzate, tuple di identica dimensione (fattore limitante). M blocchi adiacenti, contenenti ciascuno n tuple. Ogni tupla ha un indice numerico i (i -esima posizione nell'array).
- **sequenziale ordinata**: sequenza dipende dal valore assunto dal campo *chiave*. Problema principale: aggiornamento o l'inserzione di nuovi dati (si risolve tramite file differenziali, presenza di slot liberi, catene di overflow per i dati in eccesso)

Nelle **strutture ad hashing** ho un'accesso di tipo associativo ai dati, basato sul valore di un campo chiave. La struttura ha B blocchi (adiacenti). Un **algoritmo di hashing** applicato al valore della chiave, produce un valore x tale che $0 < x < B-1$. Il valore rappresenta la posizione che spetta al blocco nel file. Efficiente per query con predicati d'uguaglianza, non efficiente per query con intervalli di valori.

Blockid hash (fileid, key)

Implementazione:

- **folding**: normalizza i valori della chiave (interi uniformemente distribuiti in un ampio intervallo)
- **hashing**: trasforma il valore ottenuto in un numero tra 0 e $B-1$

Prestazioni ottimali se la struttura non è completamente piena: se B è il numero di tuple nel file, F è il numero medio di tuple nella stessa pagina, allora B ottimale è $B = T/(0.8xF)$

Collisioni: quando uno stesso #blocco corrisponde a troppe tuple, che eccedono lo spazio disponibile sul blocco -> **catena di overflow** contenente le tuple in eccesso, la sua lunghezza dipende dal rapporto $T/(FxB)$ (fattore di riempimento) e dal n°tuple per pagina F .

Strutture ad albero: più usate dai DBMS relazionali (realizzano gli indici SQL). Consentono accessi associativi a partire dal valore di una chiave.

Ogni albero ha 1 radice, nodi intermedi e nodi foglia. Ogni nodo è un blocco di memoria, i legami tra nodi sono realizzati tramite puntatori in memoria di massa. In un **albero bilanciato** le lunghezze dei cammini dalla radice alle foglie sono uguali.

B-Tree, B+Tree:

- **B+:** le foglie sono collegate a catena in base all'ordine della chiave
- **B:** non c'è la connessione a catena delle foglie. I nodi intermedi hanno due puntatori (uno al blocco contenente il valore chiave, l'altro ad un sottoalbero)

Ricerca : cerco una tupla con chiave V. Per ogni nodo **non foglia:**

1. Se $V < K_l$, seguo P₀
2. Se $V \geq K_f$, seguo P_F

Indice key-sequenced: le tuple sono nelle foglie

Indice indiretto: i nodi contengono puntatori alle tuple allocate altrove

Split: necessario quando l'inserzione di una nuova tupla non è svolgibile localmente in un nodo. Causa un incremento di puntatori nel nodo precedente e può causare un altro split

Merge: quando due nodi contigui hanno tuple che potrebbero essere contenute in un solo nodo, si effettua per mantenere un'elevato riempimento e per mantenere cammini minimi tra radice e foglie. Può causare un altro merge (diminuzione di puntatori nel predecessore)

Indici: in SQL posso associare ogni tabella a un indice:

1. **Indice primario:** key-sequenced, sulla chiave principale, unique;
2. **Indici secondari:** unique / non unique, sugli attributi usati per il join

Si aggiungono verificandone l'effettivo utilizzo da parte del sistema.

Modulo ottimizzatore: riceve in ingresso una query e produce un programma in codice oggetto che usa i metodi d'accesso (Analisi lessicale sintattica e semantica, traduzione in forma interna, ottimizzazione algebrica, ottimizzazione guidata dai costi e generazione di codice).

1. **Compile & store:** query compilata una volta ed eseguita molte volte, finchè le dipendenze sono valide.
2. **Compile & go:** esecuzione istantanea e unica

Si eseguono analisi statistiche del sistema tramite il **profiling** (cardinalità tabelle, dimensione degli attributi, numero di valori distinti, min e max di un attributo) per garantire una buona ottimizzazione da parte del relativo modulo per quanto riguarda le ottimizzazioni sui costi. Aggiornate periodicamente tramite una primitiva di sistema.

Rappresentazione interna delle query: rappresentazione ad albero (foglie corrispondono a strutture fisiche, nodi intermedi rappresentano operazioni fisiche sui dati).

Scan sequenziale: accesso a tutte le tuple di una tabella o di un risultato in sequenza (con proiezione su insieme di attributi, selezione su predicato semplice, ordinamento, inserzione/cancellazione/ modifica delle tuple cui si accede (*open, next, read, modify, insert, delete, close*)).

Sort: ordinamento in base a uno o più attributi. Può avvenire in memoria centrale o in file di grandi dimensioni.

Accessi via indice: usati quando una query contiene predicati semplici ($A_i=v$) o di intervallo ($v_1 < A_i < v_2$).

Metodi di Join: numerosi metodi di valutazione tra cui **nested-loop**, **merge scan**, **hashed**

Ottimizzazione cost-based: bisogna decidere

1. Le operazioni di accesso ai dati (scan vs uso indici)
2. L'ordine delle operazioni (es. Join)
3. La scelta del metodo di ogni operazione (es. Nested-loop per un particolare join)
4. Il grado di parallelismo ed il pipelining

Si usano profili e formule di costo approssimate, si costruisce un **albero di decisione** in cui ogni nodo corrisponde ad una scelta. Ogni nodo foglia corrisponde ad uno specifico **piano d'esecuzione**. Si associa un **costo** ad ogni piano d'esecuzione

$C_{tot} = (C_i/o * N_i/o) + (C_{cpu} * N_{cpu})$

Si sceglie il piano di costo minimo usando tecniche di branch & bound.

Glossario OQL

- *Definire una classe:*
Add class NomeClasse [**inherits** NomeSuperClasse]
type tuple(
 NomeAttributo : **tipo** ,
 ... ,
 set-of | **list-of** | **bag-of** (NomeElemento))
- *Definire un metodo:*
method NomeMetodo (NomeParametro: TipoParametro) : ClasseMetodo **is public** | **private**
- *Implementare un metodo:*
ref<NomeClasse> NomeClasse :: **NomeMetodo** (Parametri_C_Style){
Implementazione in C }
- *Implementare un costruttore:*
ref<NomeClasse> NomeClasse :: **init** (type Par1, type Par2,...type ParN){
 self->Par1_in_class = Par1;
 self->Par2_in_class = Par2;
 ...
 self->ParN_in_class = ParN;
}
- *Estendere un costruttore:*
ref<NomeClasseExt> NomeClasseExt :: **init** (type Par1, ...,type ParN, type NewPar1,... type NewParN){
 (**ref**<NomeClasse>)**self**->**init**(Par1, ..., ParN);
 self->NewPar1_in_class = NewPar1;
 ...
 self->NewParN_in_class = NewParN; }

Glossario TRIGGERS

Trigger in Oracle, sintassi

```
create trigger TriggerName
Mode Event {, Event} on TargetTable
[ [referencing Reference]
for each row
[when SQLPredicate]]
PL/SQLBlock
```

1. *Mode*: before o after
2. *Event*: insert, update, delete
3. **for each row** specifica la granularità
4. *Reference*: permette di definire nomi di variabili (utilizzabili solo per granularità di ennupla): old as OldVariable | new as NewVariable

Trigger in SQL-3, sintassi

```
<SQL3-trigger> ::=      CREATE TRIGGER <trigger-name>
                        {AFTER | BEFORE} <trigger-event>
                        ON <table-name>
                        [REFERENCING <references>]
                        [FOR EACH {ROW | STATEMENT}]
                        [WHEN <SQL-statements>]
                        <SQL-procedure-statements>

<trigger-event> ::= INSERT | DELETE | UPDATE [OF <column-names>]
<reference>      ::= OLD [AS] <old-value-tuple-name> | NEW [AS] <new-value-tuple-name> |
                  OLD_TABLE [AS] <old-value-table-name> | NEW_TABLE [AS] <new-value-table-
name>
```