Basi di Dati

Davide Lanza

2017-18

Handwritten Drafts:

Cap. 1: Introduzione

1. Il Database Management System - 1
2. Livelli di Astrazione - 1
3. Schema ed Istanze - 1
4. Modelli dei Dati - 1
5. Data Manipulation Language - 1
6. Data Definition Language - 2
7. Progettazione di un DB - 2
8. Architettura delle DB applications - 2
9. Funzioni del DBMS - 2
10. Struttura Generale del DBMS - 3

Cap. 2: Il modello relazionale

1. Le relazioni - 4
2. Chiavi - 4
3. Linguaggi Query (DQL) - 5
4. Algebra relazionale - 5
    a. Operatori Base - 5
    b. Operatori Addizionali - 6

Cap. 3: SQL

1. DDL (Data Definition Language) - 8
2. DQL (Data Query Language) - 8
3. DML (Data Modification Language) - 11
4. More DQL: Joined Relations - 11
5. Views - 12
6. Valori *null* - 13

Cap. 4: Advanced SQL

1. SQL Data Types - 14
2. Vincoli (Constraints) - 14
3. DCL (Data Control Language) - 15
4. Embedded SQL - 16
5. ODBC (Open DataBase Connectivity) - 17
6. JDBC (Java DataBase Connectivity) - 18
7. Funzioni e Procedure - 18
8. Ricorsione in SQL - 20
9. Funzioni Avanzate - 20

Cap 4: Advanced SQL (English version)

1 Built-in Data Types in SQL 1
1.1 User-Defined Types 1
2 Constraints 1
2.1 Domain Constraints 1
2.2 Large-Object Types 2
2.3 Integrity Constraints 2
2.4 Constraints on a Single Relation: 2
2.4.1 Not Null Constraint 2
2.4.2 The Unique Constraint 2
2.4.3 The check clause 2
2.5 Referential Integrity 3
2.6 Assertions 3
3 Data Control Language (DCL) 4
3.1 Authorization 4
3.1.1 Authorization Specification in SQL 4
3.2 Privileges in SQL 4
3.2.1 Revoking Authorization in SQL 5
4 Embedded SQL 5
4.1 Updates Through Cursors 6
4.2 Dynamic SQL 6
5 ODBC 6
5.1 ODBC More features 8
5.2 ODBC Conformance Levels 8
6 JDBC 9
7 Procedural Extensions and Stored Procedures 10
7.1 Functions and Procedures 10
7.2 SQL Functions 10
7.3 Table Functions 11
7.4 SQL Procedures 11
7.5 External Language Functions/Procedures 12
7.6 Security with External Language Routines 13
7.7 Recursion in SQL 13
7.7.1 Example of Fixed-Point Computation 14
8 Advanced Features 14

Cap 6. Modello E-R

1 Entità, attributi, relazioni 1
2 Cardinalità e chiavi 2
3 Diagrammi E-R 2
3.1 Cardinalità in relazioni a grado > 2 3
4 Problemi e dubbi comuni (1) 3
5 Entità deboli 4
6 Specializzazione e generalizzazione 5
6.1 Definizione dei vincoli nelle specializzazioni/generalizzazioni 5
7 Aggregazione 6
8 Problemi e dubbi comuni (2) 6
9 Simboli dei diagrammi E-R 7
10 Riduzione allo modello relazionale 7
10.1 Eliminazione delle ridondanze 7
10.2 Attributi composti e multi-valore 8
10.3 Riduzione delle specializzazioni 8
10.4 Ridurre le aggregazioni 9
11 UML 9
12 Dipendenze di esistenza 10
13 Notazione modello E-R alternativa 10

Cap 7. Progettazione DB relazionali

1 Decomposizione e prima forma normale 1
2 Dipendenze funzionali 2
2.1 Chiusura e BCNF 3
2.2 BCNF, la conservazione delle dipendenze e la 3NF 3
2.3 Obiettivi della normalizzazione 4
2.4 Forme normali ulteriori 4
3 Teoria formale delle dipendenze funzionali 4
3.1 Chiusura delle dipendenze funzionali 4
3.2 Chiusura degli attributi 5
3.3 Copertura canonica 5
3.4 Lossless-join decomposition 6
3.5 *Dependency preserving decomposition* 7
3.6 Boyce-Codd Normal Form 7
3.7 Terza forma normale (3NF) 8
4 Obiettivi della progettazione 9
5 Dipendenze Multi-valore (MVDs) 10
5.1 Quarta forma normale (4NF) 11
6 Forme normali ulteriori 12
7 Progettazione di un DB 12
8 Modellazione di dati temporali 13

Cap 8. Progettazione e sviluppo delle *applications*

1 Interfacce utente e strumenti: HTML ... 1
1.1 Uniform Resource locators (URLs) ... 1
1.2 HTML: Gestione degli input ... 2
1.3 Scripting "lato client" e applets ... 2
1.4 Scripting "lato client" e sicurezza ... 2
1.5 Server web ... 3
1.6 Sessioni ... 3
1.7 Servlet ... 4
1.8 Scripting "lato server" ... 4
1.9 Miglioramento delle prestazioni del server Web ... 5
2 Triggers ... 5
2.1 Attivare eventi ed azioni in SQL mediante i trigger ... 6
2.2 Statement Level Triggers ... 6
2.3 External World Actions ... 6
2.4 Quando non usare i trigger ... 7
3 Autorizzazioni in SQL ... 8
3.1 Tipi di Autorizzazioni ... 8
3.2 Autorizzazioni e Views ... 8
3.3 Concessione dei Privilegi ... 9
3.4 Specifiche di sicurezza in SQL ... 9
3.5 Ruoli ... 10
3.6 Revoca delle Autorizzazioni in SQL ... 10
3.7 Limitazioni delle Autorizzazioni SQL ... 10
4 Audit Trails ... 11
5 Sicurezza nell'Application ... 11
5.1 Crittografia ... 11
5.2 Autenticazione ... 11
5.3 Certificati digitali ... 12

Cap 9.1. Object-Oriented Databases

1 La necessità di avere *Data Types* complessi 1
2 L' *Object-Oriented Data Model* 1
2.1 Confronto con il modello E-R 1
2.2 Struttura di un oggetto 1
2.3 Messaggi e metodi 2
2.4 Classi di oggetti 2
2.5 Inheritance 2
2.6 Multiple Inheritance 3
2.7 *Object identity* 4
2.8 *Object Containment* 4
3 Linguaggi *Object-oriented* 5
4 Linguaggi di programmazione persistenti 5
4.1 *Persistence of Objects* 5
4.2 *Object Identity* e puntatori 6
4.3 Archiviazione e accesso di oggetti persistenti 6
5 Sistemi C ++ persistenti 6
5.1 ODMG: C ++ Object Definition Language 7
5.2 ODMG Types 7
5.3 ODL: Object Definition Language (ODMG C++) 7
5.4 ODL - Implementare le Relazioni 7
5.5 OML: Object Manipulation Language (ODMG C++) 8
5.6 OML - Funzioni Database e Funzioni Oggetto 8
5.7 ODMG C++: Altre caratteristiche 9
5.8 Rendere "trasparente" la *Pointer Persistence* 10
6 Sistemi Java persistenti 10

Cap 9.2: Object-Relational Databases

1 Relazioni nidificate ... 1
1.1 Decomposizione 4NF di una Relazione Nidificata ... 1
1.2 Problemi con lo schema 4NF ... 2
2 Tipi complessi e orientamento agli oggetti ... 2
2.1 Tipi complessi e SQL 1999 ... 2
2.2 Collection Types ... 2
2.3 Large Object Types ... 3
2.4 Structured and Collection Types ... 3
2.4.1 Tipi strutturati ... 3
2.5 Creazione dei Valori di Tipi Complessi ... 4
2.6 Inheritance ... 4
2.6.1 Multiple Inheritance ... 5
2.6.2 Table Inheritance ... 5
2.7 Reference Types ... 6
2.7.1 Inizializzazione dei valori di *Reference types* ... 6
2.8 User Generated Identifiers ... 7
3 Query con tipi complessi ... 7
3.1 Path Expressions ... 7
3.2 Query con i tipi strutturati ... 7
3.3 Collection-Value Attributes ... 8
3.4 Unnesting ... 8
3.5 Nesting ... 8
4 Funzioni e procedure ... 9
4.1 Funzioni SQL ... 9
4.2 Metodi SQL ... 9
4.3 Procedure in SQL ... 10
4.4 Funzioni/procedure in Linguaggi Esterni ... 10
4.5 Sicurezza nelle Routines in Linguaggi Esterni ... 10
4.6 Costrutti procedurali ... 11
5 Confronto tra *Object-Oriented* e *Object-Related* ... 11
5.1 Trovare tutti i dipendenti di un manager ... 12

Cap 9. Database Object-Based

1 Object-Relational Data Models 1
2 Relazioni Annidiate 1
3 Tipi complessi e SQL 1999 2
4 Structured Types in SQL 2
4.1 Structured Types e Metodi 3
4.2 Ereditarietà 3
4.3 Ereditarietà Multipla 3
4.4 Requisiti di coerenza per le sotto-tabelle 4
5 Array e Multiset Types in SQL 4
5.1 Creazione di Collection Values 4
5.2 Query sui Collection-Valued Attributes 5
6 Annidamento (nesting) 5
6.1 Unnesting 5
6.2 Nesting 5
7 Object-Identity & Reference Types 6
7.1 Inizializzare i Reference-Typed Values 6
7.2 User-Generated Identifiers 7
8 Path Expressions 7
9 Implementazione di funzionalità O-R 8
10 Linguaggi di programmazione persistenti 8
11 Object-Identity & Puntatori 8
12 Confronto dei database O-O e O-R 8

Cap 10. XML

1 Introduzione a XML 1
1.1 Confronto con i DB relazionali 1
2 Struttura dell'XML 2
2.1 Attributi 3
2.2 Attributi vs. Sotto-elementi 3
2.3 Namespace 3
2.4 Ulteriori Note sulla Sintassi XML 4
3 XML Document Schema 4
3.1 Document Type Definition (DTD) 4
3.2 Attribute Specification in DTD 5
3.3 ID e IDREFs in DTDs 5
3.4 Limiti dei DTDs 6
3.5 XML Schema 6
3.6 Altre funzionalità di XML Schema 7
4 Query e Traduzione Dati in XML 8
4.1 Modello ad Albero per i Dati XML 8
4.2 XPath 8
4.3 Funzioni in XPath 9
4.4 XQuery 9
4.5 Join in XQuery 10
4.6 Query annidate in XQuery 10
4.7 Sorting in XQuery 11
4.8 Altre Funzionalità di XQuery 11
4.9 XSLT 11
4.10 Creazione di output XML con XSLT 12
4.11 Ricorsione strutturale con XSLT 13
4.12 Join in XSLT 13
4.13 Sorting in XSLT 13
5 Application Program Interface 14
6 Archiviazione di dati XML 14
6.1 Database relazionali: *String representation* 14
6.2 Database relazionali: *Tree Representation* 15
6.3 Database relazionali: *Mapping XML Data to Relations* 15
6.4 Publishing and Shredding dei Dati XML 16
6.5 Storage nativo di Dati XML in un DB Relazionale 16
7 SQL / XML 16
8 XML Application: Web Services 17

Cap 11. Strutture di archiviazione e di file

1 Supporti fisici di archiviazione (*Physical Storage Media*) ... 1

1.1 Criteri di Classificazione ... 1

1.2 Tipologie ... 1

2 Gerarchia di archiviazione ... 3

3 Hard-Disk magnetici ... 3

3.1 Disk Subsystem ... 4

3.2 Misure di prestazione dei dischi ... 4

3.3 Ottimizzazione del *Disk-Block Access* ... 5

4 RAID (Redundant Arrays of Independent Disks) ... 5

4.1 Migliore Affidabilità sfruttando la Ridondanza ... 6

4.2 Migliore Prestazioni sfruttando il Parallelismo ... 6

4.3 Livelli RAID ... 7

4.4 Scelta del livello RAID ... 9

4.5 Problemi hardware ... 9

4.6 RAID nel settore industriale ... 10

5 Dischi ottici ... 10

6 Nastri magnetici ... 10

7 Accesso allo storage ... 11

7.1 Buffer Manager ... 11

8 Organizzazione dei file ... 12

8.1 Fixed-Length Records ... 12

8.2 Variable-Length Records ... 12

8.3 Organizzazione dei record nei file ... 13

8.4 Organizzazione file sequenziale ... 13

8.5 Multitable clustering file organization ... 14

8.6 Archiviazione del data dictionary ... 14

9 Extra ... 15

9.1 Rappresentazione di record fixed-length ... 15

9.2 Rappresentazione di record variable-length ... 15

9.3 Argomenti non trattati ... 16

Cap 12. Indicizzazione e hashing

1 Concetti basilari 1
1.1 Metriche di valutazione degli indici....1
2 Ordered indices....1
2.1 Dense Index e Sparse Index....2
2.2 Multilevel index 2
2.3 Aggiornamento degli indici 3
2.4 Esempio di indici secondari 4
2.5 Dettagli su indici primari e secondari 4
3 I B-Alberi (da Wikipedia) 4
3.1 Definizione del B-Albero 5
3.2 Vantaggi dei B-Alberi 5
3.3 Varianti del B-Tree 5
4 B +Tree Index Files 6
4.1 Perché usare un B +Tree 6
4.2 Struttura del B +Tree....7
4.3 Struttura del Nodo B +Tree....7
4.4 Nodi foglia nei B +Tree....7
4.5 Nodi non foglia nei B +Tree....8
4.6 Osservazioni su B +Tree 8
4.7 Query su B +Trees....9
4.8 Aggiornamenti su B +Trees 9
4.9 B +Tree File Organization.... 13
4.10 Stringhe di indicizzazione.... 13
5 B-Tree Index Files 13
5.1 Vantaggi e svantaggi di un B-Tree 14
6 Indicizzazione: caratteristiche avanzate 14
6.1 Multiple-key e Composite-key 14
6.2 *Search keys* non univoche 15
6.3 Altri problemi nell'indicizzazione.... 15
7 Hashing statico.... 16
7.1 Funzioni *hash* ideali e non ideali 17
7.2 Gestire i *bucket overflow*.... 18
7.3 Indici hash.... 18
7.4 Problemi (d*eficiencies*) dell'hashing statico.... 19
8 Hashing dinamico.... 19
8.1 Struttura generale *extendable hash*.... 19
8.2 Uso della *extendible hash structure* 20

8.2.1 Ricerca di un *bucket* 20
8.2.2 Inserimento di un *record* 20
8.2.3 Cancellazione di un *key-value* 21
8.3 Esempio di uso della struttura hash estendibile 21
8.4 Confronto *Extendible hashing* con ad altri schemi 22
8.5 Confronto tra l'o*rdered indexing* e l'hashing 22
9 Indici bitmap 22
9.1 Bitmap operations 23
9.2 Accorgimenti nell'uso di *bitmap* 23
9.3 Implementazione efficiente delle operazioni bitmap 24
10 Definizione degli indici in SQL 24
11 Appendice 24
11.1 Hashing partizionato 24
11.2 Grid Files 25

Cap. 1 : Introduzione

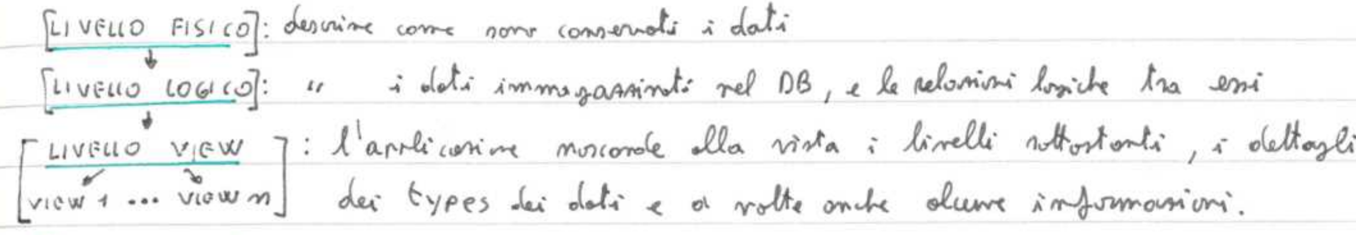
1. Il DataBase Management System

Il DBMS contiene: i dati e le loro relazioni, un insieme di programmi per accedere ad essi ed un "ambiente conveniente" ed efficiente da usare

Prima il DB era implementato direttamente sul "file system" -> PROBLEMI -> il DBMS li risolve:

DBMS risolve ridondanze ed inconsistenze, le difficoltà di accesso ed i problemi di integrità dei primi DB. In esso i dati sono ATOMICI, ACCESSIBILI A PIÙ UTENTI e SICURI.

2. Livelli di Astrazione



3. Schema ed Istanze

SCHEMA: la struttura logica del DB (analogo del "type" in programmazione)
- ↳ SCHEMA FISICO: DB design a livello fisico
- ↳ " LOGICO: " " " " logico

ISTANZA: il contenuto del DB ad un dato momento (analogo del "value" in programmaz.)

INDIPENDENZA FISICO-DATI: posso cambiare lo schema fisico senza dover cambiare lo schema logico

4. Modelli dei Dati

I DATA MODELS sono una serie di STRUMENTI usati per descrivere i dati, le loro relazioni, le loro semantiche ed i loro vincoli. Ve ne sono diversi:
- Modello RELAZIONALE, MODELLO E-R (-> principalmente per la progettazione di DBs), BASATI SUGLI OGGETTI (-> Object-oriented e Object-relational), SEMISTRUTTURATI (-> es: XML)

5 Data Manipulation Language

DML è il linguaggio che serve a MANIPOLARE ed ORGANIZZARE i dati nel Data Model.
~~È anche noto come LINGUAGGIO QUERY.~~ -> non è il DQL??

QUERY: INTERROGAZIONE AL DB PER ESTRARRE O AGGIORNARE I DATI CHE SODDISFANO UN CERTO CRITERIO DI RICERCA.

- DML PROCEDURALE: specifica i dati e come accederci
- DML DICHIARATIVO: " " " ma non come accederci

Il più usato è SQL (è un DML ma anche un DDL -> vedi dopo) e DQL

6. Data Definition Language

DDL è il linguaggio che serve a DEFINIRE la STRUTTURA LOGICA (LOGICAL SCHEMA) del DB, ma non fornisce gli strumenti per modificare i valori assunti dai dati o per interrogare i dati stessi (-> per quello si usa il DML ed il DQL)

Solitamente genera un insieme di TABELLE contenute in un DIZIONARIO (DATA DICTIONARY), il quale contiene anche i METADATI, quali:

- lo SCHEMA del DB, i vincoli di integrità, le autorizzazioni...

7. Progettazione di un DB

La PROGETTAZIONE DI UN DB (DB Design) definisce la struttura generale di esso:

- PROGETTAZIONE LOGICA: riguarda lo SCHEMA LOGICO del DB -> si devono trovare dei buoni insiemi di relazioni, adatti ai dati gestiti
- PROGETTAZIONE FISICA: riguarda lo SCHEMA FISICO del DB

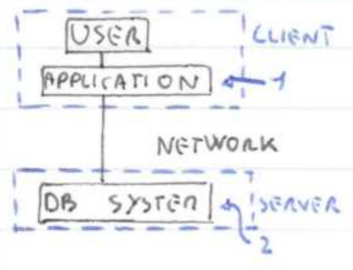
8. Architettura delle DB Applications

È l'ARCHITETTURA delle APPLICAZIONI che accedono al DB, riguarda il LIVELLO VIEW, determina come le applicazioni accedano ai dati del DB.

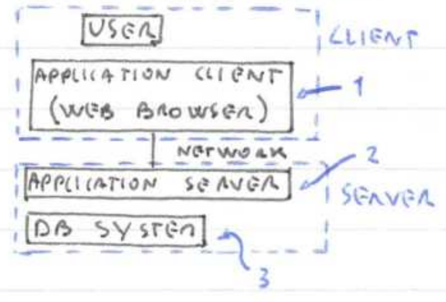
La più diffusa è la CLIENT-SERVER:

- separa la gestione della rappresentazione
- la connessione può essere fisica o attraverso un network

ARCHITETTURA A 2 STRATI (TIER):



ARCHITETTURA A 3 TIER:



9. Funzioni del DBMS

1. GESTIONE DELLA MEMORIZZA (STORAGE MANAGEMENT): modulo del programma che interfaccia i dati a BASSO LIVELLO del DB con le applicazioni e le query.

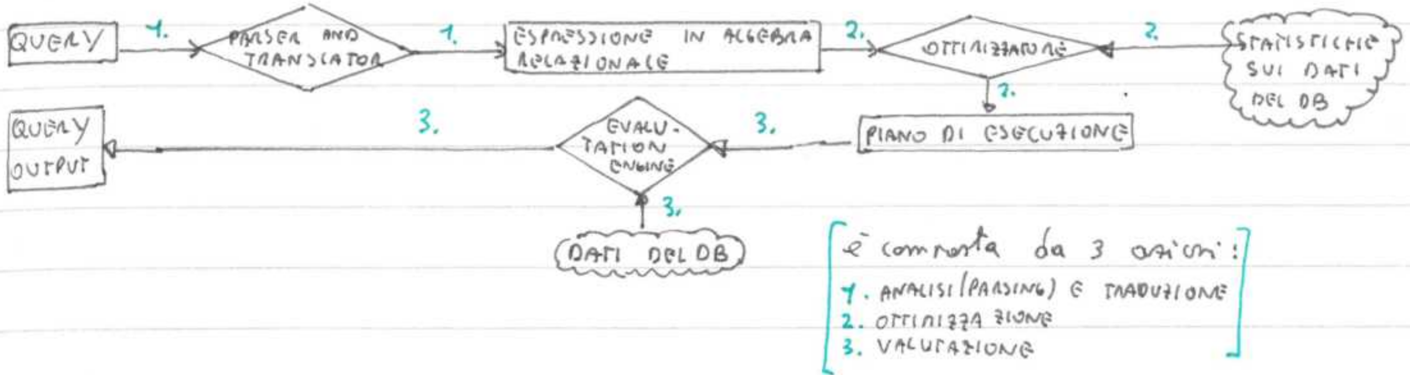
   ↳ Il FILE SYSTEM fa già tutto ciò però: 3 soluzioni:

   I.) il DBMS non se ne occupa, lo fa il FILE SYSTEM -> costo minore

   II.) " " se ne occupa, il FILE SYSTEM no

III.) i due "coesistono" ognuno su una sua PARTIZIONE di disco

2) PROCESSAZIONE QUERY:



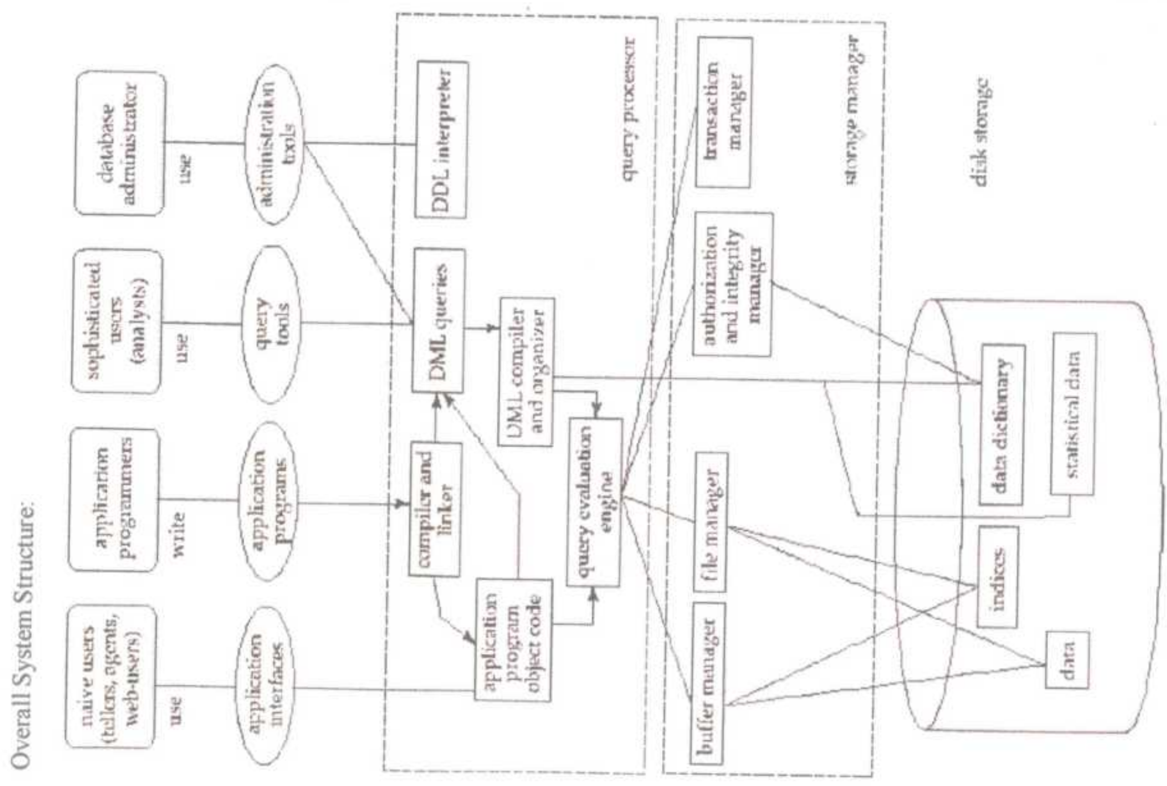
[è composta da 3 azioni:
1. ANALISI (PARSING) E TRADUZIONE
2. OTTIMIZZAZIONE
3. VALUTAZIONE]

↳ ci sono modi differenti da questo (es: ESPRESSIONI EQUIVALENTI, ALGORITMI DIFFERENTI PER OGNI OPERAZIONE...)
↳ è fatta una query alla volta → influisce molto su prestazioni e costo → dipende moltissimo dalle INFORMAZIONI STATISTICHE sulle relazioni che il DB deve mantenere

3) TRANSACTION MANAGEMENT:
una TRANSACTION [è una collezione di operazioni che eseguono una singola funzione logica in una DB Appl.]

- il TRANSACTION PROCESSOR assicura che il DB rimanga in uno stato consistente (corretto) anche se ci sono dei problemi al sistema o nelle transactions.
- il CONCURRENCY CONTROL MANAGER controlla l'interazione tra le transazioni in contemporanea, per assicurare la consistenza del DB

10. STRUTTURA GENERALE DEL DBMS



Cap. 2: Il Modello Relazionale

1. Le Relazioni

- Una RELAZIONE (R) è un insieme di n-uple (TUPLES) definite come $(a_1, \dots a_n)$ su un DOMINIO $D_1 \times D_2 \times \dots \times D_n$, dove $a_i \in D_i, \forall i$
- Gli a_i sono detti ATTRIBUTI. Ogni attributo ha un NOME ed un insieme di valori consentiti (DOMINIO DELL'ATTRIBUTO). Normalmente i valori devono essere ATOMICI (indivisibili)
- Un DOMINIO è ATOMICO se tutti i suoi membri sono atomici
- null è membro di ogni dominio!
- Lo SCHEMA DELLA RELAZIONE è fatto da:
  - definizioni degli attributi (nome, type/dominio)
  - vincoli di integrità
- L'ISTANZA DELLA RELAZIONE è specificata da una tabella:

ATTRIBUTO #1	(...)	ATTRIBUTO #m
TUPLA #1		
(...)	(...)	(...)
TUPLA #n		

Un DB contiene ed è fatto da più relazioni. → NORMALIZZAZIONE: come progettare uno schema relazionale (→ CAP. 7)

2. Chiavi

Data una $r(R)$ (→ relazione r su un dominio R), prendiamo $K \subseteq R$:

- K è una SUPERKEY di R se i suoi valori sono sufficienti ad identificare univocamente ogni possibile relazione $r(R)$
- K è una CANDIDATE KEY di R se è MINIMALE, ovvero ∄ superkey di K, ed essa è superkey di R
- La PRIMARY KEY è la CANDIDATE KEY scelta per identificare le TUPLE nella relazione ↳ deve essere fatta da attributi fissi, o che cambiano molto raramente.
- Una FOREGIN KEY è un attributo di una relazione che è PRIMARY KEY di un'altra, è un VINCOLO DI INTEGRITÀ REFERENZIALE tra due relazioni → identifica uno o più attributi di una relazione (RELAZIONE REFERENZIANTE) che referenzia uno o più attributi di un'altra relazione (RELAZIONE REFERENZIATA). [Solo i valori ammessi nella relazione referenziata, dove essa è PRIMARY KEY, sono ammessi nella relaz. referenziante]!

3. Linguaggi Query (DQL)

Sono i linguaggi con i quali l'utente richiede informazioni al DB. → Data Query Language (DQL)
Possono essere procedurali o dichiarativi.
"Pure languages": linguaggi che permettono di esaminare le query (Algebra relaz., Calcolo relaz. di Tuple/Domini ecc...)

4. Algebra Relazionale

È un linguaggio procedurale (contrariamente al calcolo relazionale, che è dichiarativo)
Gli operatori prendono delle relaz. in input e ne restituiscono una in output.

Operatori Base:

1. SELECT (relazionale) [$\sigma_P(r)$] → prende le tuple in cui è verificata la condizione P e le mette in una relazione che darà in output
   (in SQL è il Where)
   ↳ output: stesso grado (n° colonne, n° attributi) ma ≤ cardinalità (n° righe, n° tuple) → (σ seleziona le righe)

2. PROJECT [$\Pi_{D_i,...}(r)$] → prende le sole colonne D_i (domini degli attributi) e le dà in output
   (in SQL è il select)
   ↳ output: stessa cardinalità, ≤ grado → ("leva" delle colonne)

3. UNION [$r_1 \cup r_2$] → se le r hanno lo stesso grado e gli stessi tipi di attributi (gli stessi domini) restituisce una relazione avente tutte le tuple delle relaz. precedenti, ma senza ripetizione
   ↳ output: cardinalità ≥ → (mette insieme le tuple)

4. SET DIFFERENCE [$r_1 - r_2$] → se le r hanno lo stesso grado e gli stessi tipi di attributi restituisce una relaz con gli elementi di r_1 che non sono di r_2
   ↳ output: cardinalità ≤ → ("leva" delle tuple)

5. CARTESIAN PRODUCT [$r_1 \times r_2$] → restituisce la combinazione di tutti gli elementi delle due relazioni
   ↳ output: cardinalità e grado somma dei due precedenti

6. RENAME [$\rho_{r_2(A_1,...,A_m)}(r_1)$] o anche $\rho_{X(A_1,...,A_m)}(E)$ (rinomina attributi opzionale) → modifica lo schema di una relazione (di grado m) o di una espressione di alg. relaz. (di arità n) modificandone il nome, ed opzionalmente i nomi degli attributi, se desiderato

(il grado è per le relaz. → n° di attributi; l'arità è per le espressioni dell'algebra relaz → n° di argomenti)

Operatori Addizionali: si chiamano così perché possono essere espressi usando quelli base

7. SET INTERSECTION $[r_1 \cap r_2]$ → se le r hanno lo stesso grado e lo stesso tipo di attributi restituisce una relaz. con le tuple di r_1 E r_2
↳ OUTPUT: CARDINALITÀ ≤

8. NATURAL JOIN $[r_1 \bowtie r_2]$ → prende le due relaz. e ne fa il prodotto cartesiano, poi seleziona le tuple che hanno lo stesso valore nei domini comuni
↳ OUTPUT: GRADO ≥, CARDINALITÀ ≤ SOMMA delle due precedenti

esempio:

r

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

s

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ε

PRENDO GLI ATTRIBUTI COMUNI E PROIETTO UNA DELLE DUE RELAZ. → $\Pi_{B,D}(s)$

B	D
1	a
3	a
1	a
2	b
3	b

CONFRONTO CON r:
(α, 1, α, a) e (α, 1, γ, a) — E: α
NESSUNO — E: β
(α, 1, α, a) e (α, 1, γ, a) — E: γ
(δ, 2, β, b) — E: δ
NESSUNO — E: ε

⇒

A	B	C	D	E
α	1	α	a	α
α	1	γ	a	α
α	1	α	a	γ
α	1	γ	a	γ
δ	2	β	b	δ

o $r \bowtie s$

MI RIMANE DA ASSOCIARE L'ULTIMO ATTRIBUTO DI s

AGGREGATE FUNCTION: prende una collezione di valori e ne fa un'operazione:
$F(A)$ → funzione su un attributo
↳ AVG (ATTRIBUTO): media dei valori dell'attributo ; COUNT (A): numero di valori
MIN (A): valore minimo ; MAX (A): valore massimo ; SUM (A): somma dei valori

9. AGGREGATE OPERATION $[_{G_1,\dots,G_m}\mathcal{G}_{F_1(A\dots),\dots,F_m(A\dots)}(r)]$ → prende la relazione r, la raggruppa per il/i gruppo/i G_1/G_m e per ogni gruppo esegue la/le $F_i(A\dots)$ sui rispettivi attributi

esempio:

A	B	C	D
α	a	0	x
α	b	1	y
β	c	1	x
β	b	0	y
γ	c	2	y

$\mathcal{G}_{sum(C)}(r) = 4$

$_A\mathcal{G}_{sum(C)}(r)$ →

A	sum(C)
α	1
β	1
γ	2

A: RAGGRUPPA PER ELEMENTI DI A
sum(C): PER OGNI ELEMENTO CALCOLA LA SOMMA SU C

I risultati NON hanno un nome! Si possono rinominare così:
$[_A\mathcal{G}_{sum(C)}$ AS SUM_OF_C $(r)]$ → chiama la colonna "SUM_OF_C"

10. OUTER JOIN $[r_1 ⟕ r_2]_{LEFT}$ $[r_1 ⟖ r_2]_{RIGHT}$ $[r_1 ⟗ r_2]_{FULL}$ → Computa il join e per le tuple "lasciate fuori" da quello naturale aggiunge un valore "null" per gli attributi non condivisi e le inserisce nell'output (→ FULL OUTER JOIN). (Se lo fa solo con gli attributi di r_1 → ⟕, di r_2 ⟖).

esempio:

r:

A	B	C
∅	a	α
1	c	β
3	b	γ

s:

D	A
x	1
y	2
z	∅

$r \bowtie s$:

A	B	C	D
∅	a	α	z
1	c	β	x

$r ⟖ s$:

A	B	C	D
∅	a	α	z
1	c	β	x
2	null	null	y

$r ⟕ s$:

A	B	C	D
∅	a	α	z
1	c	β	x
3	b	γ	null

$r ⟗ s$:

A	B	C	D
∅	a	α	z
1	c	β	x
3	b	γ	null
2	null	null	y

VALORE null: significa "sconosciuto", ovvero il valore non esiste. Vediamo alcuni casi col null:

- ↳ AGGREGATE FUNCTIONS: ignorano il null
- ↳ ESPRESSIONI ARITMETICHE: restituiscono "null"
- ↳ ELIMINAZIONE DUPLICATI e GROUPING: trattato come qualsiasi altro valore
- ↳ COMPARAZIONI (es: null > 5): restituiscono "unknown"
- ↳ LOGICA A TRE VALORI:
  - OR: unknown ∨ T = T; unknown ∨ F = unknown; unknown ∨ unknown = unknown
  - AND: unk. ∧ T = unk.; unk. ∧ F = F; unk. ∧ unk. = unk.
  - NOT: NOT(unk.) = unk.
- ↳ SELECT: il predicato P di $\sigma_P(r)$ è valutato "falso" se vi è dentro un "null"

11. DIVISION $[r_1 \div r_2]$

Presi r_1 su $R_1(A,B)$ e $r_2(R_2)$ su $R_2(B)$, è (il più grande) insieme di tuple con schema A t.c., facendo il prod. cartesiano con r_2, ciò che si ottiene sia una relaz. contenuta in r_1

↳ [$r_1 \div r_2 = q$ è la più larga relazione che soddisfa la condizione $[q \times r_2] \subseteq r_1$]

esempio:

r:

A	B	
α	1	ok!
α	2	
α	3	
β	1	
γ	1	[No! non ci sono γ con 2]
δ	1	No!
δ	3	
ε	6	
ε	1	No!
β	2	ok!

s:

B
1
2

[Devo prendere tutti gli elementi di r che sono in relazione con tutti quelli di s] →

$(q = r \div s)$ ↓

q:

A
α
β

(Definizioni formali in appendice, slide 2.51 in poi...)

Cap 3: Structured Query Language (SQL)

(SQL: DDL[1] + DML[3] + DQL[2,4] + DCL + TCL)

1. DDL (Data Definition Language)

Permette di specificare lo SCHEMA di ogni relazione, i VINCOLI DI INTEGRITÀ, le AUTORIZZAZIONI per l'accesso alle informazioni ...

① CREATE TABLE: [create table r ($A_1 D_1$, ..., $A_n D_n$, (vincolo d'integrità$_1$), ... (vincolo d'integrità$_n$))]

es: [create table "branch" (branch_nome char(15), branch_city char(30), assets integer)]

DOMAIN TYPES: char(n): stringa di n caratteri ; varchar(n): stringa fino a n caratteri

int, smallint: dimensioni max finite dalla macchina.

numeric(p,d): p cifre prima della virgola, n cifre dopo.

real, double precision: come int, più precisi, con la virgola mobile.

float(n): numero a virgola mobile, precisione almeno di n cifre.

② VINCOLI DI INTEGRITÀ: es: [create table branch (branch_nome char(15), branch_ID integer not null, assets integer, primary key (branch_ID))]

[NOT NULL: l'attributo non può assumere il valore null

PRIMARY KEY: identifica la chiave primaria.]

↳ [da SQL 92 "primary key" sottointende "not null"]

③ INSERIMENTO TUPLE: es: [insert into account values (nomebranch, nomecittà, numeroassets)]

↳ l'inserimento fallisce se un qualche vincolo d'integrità è violato

[③ e ④ sono del DML!]

④ CANCELLAZIONE DI TUTTE LE TUPLE: es: [delete from account] → [cancella tutte le tuple]

⑤ CANCELLAZIONE TABELLA: es [drop table account]

⑥ INSERIMENTO ATTRIBUTI: es [alter table r add A D] → [aggiunge alla relazione r l'attributo A di dominio D]

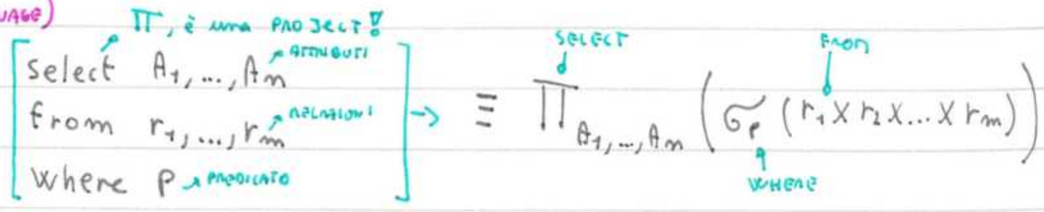
⑦ CANCELLAZIONE ATTRIBUTI: es [alter table r drop A] → [elimina dalla relaz r l'attributo A]

↳ non supportato da molti DB

2. DQL (Data Query Language)

Query modello base in SQL } →

↳ restituisce una relazione.



$$\equiv \prod_{A_1,\dots,A_m}\left(\sigma_P(r_1 \times r_2 \times \dots \times r_m)\right)$$

→ SELECT IN SQL È PROJECTION!!!

1. SELECT FROM:

```
select NOME_ATTRIBUTO
from RELAZIONE
```
$\longleftrightarrow \Pi_A(r)$

- select ≡ select all
  - ↳ mantiene i duplicati
- select distinct
  - ↳ elimina i duplicati

```
select *          ← selez. tutti gli attributi
from r1,r2        ← del PRODOTTO CARTESIANO r1 x r2
```

2. WHERE:

```
select NOME_ATTRIBUTO
from RELAZIONE
where PREDICATO
```
$\longleftrightarrow \sigma_p(r)$

→ WHERE è una SELECTION!!! (RELAZIONALE)

3. AS:

```
select r1.A1, r1.A2 as AttributoDue, r2.A1
from r1, r2
```
← [rinomina l'A_2 della r_1]

↳ Posso definire delle TUPLE VARIABLES rinominando le relazioni:

```
select T.A1, T.A2, S.A1
from r1 as T, r2 S
```
← [posso anche omettere "as"]

4. LIKE:

```
select ATTRIBUTO1
from RELAZIONE
where ATTRIBUTO2 like '%STRINGA_DI_TESTO%'
```

[like serve per fare paragoni di caratteri '_C_' o stringhe '%STRING%']

↳ SQL permette molte operazioni sulle stringhe.

5. ORDER BY:

```
select ATTRIBUTO
from RELAZIONE
order by ATTRIBUTO asc/desc
```

→ [ordina nei risultati le tuple in base al valore che hanno in "ATTRIBUTO", asc (di default, comando superfluo) o desc (descending).]

6. ∪, ∩, −:

```
RELAZIONE1 union all RELAZIONE2      ←→ ∪
   "       intersect all   "         ←→ ∩
   "       except all      "         ←→ −
```
} SU TUTTI GLI ATTRIBUTI DI r_1, r_2

```
(select ATTRIBUTO from RELAZIONE1)
union
(select ATTRIBUTO from RELAZIONE2)
```
→ SUI SINGOLI ATTRIBUTI

7. AGGREGATE FUNCT.: (avg, min, max, sum, count) come $F(A)$:

```
select F(ATTRIBUTO)
from (RELAZIONE)
```

es: [FA LA MEDIA DEI SALDI DEI CONTI]

```
select avg(balance)
from account
```

↳ GROUP BY:

es:
```
select ATTRIBUTO1, count(distinct ATTRIBUTO2)
from RELAZIONE
group by ATTRIBUTO1
```

→ [CONTA QUANTI VALORI DISTINTI DI ATTR.2 CI SONO IN OGNI VALORE DISTINTO DI ATTR.1]

[RAGGRUPPA I VALORI UGUALI DI ATTR.1 ED ESEGUE SU OGNI GRUPPO L'OPERAZIONE DESCRITTA DALLA $F(A) \rightarrow F(\text{GRUPPO})$]

↳ HAVING: è un where applicato NON prima (il where sì) della formazione dei gruppi, ma dopo:

```
select ATTRIBUTO 1, F(ATTR. 1)
from RELAZIONE
group by ATTRIBUTO 1
having P        (P ad es: F(ATTR. 1) > 1)
```

[PRIMA RAGGRUPPA I VALORI DI ATTR. 1, POI VERIFICA LA CONDIZIONE P
↓
IL WHERE PRIMA VERIFICA E POI RAGGRUPPA]

Ci sono poi alcuni costrutti che utilizzano NESTED SUBQUERIES in "where":

8. IN:

```
select ATTR. 1
from RELAZIONE 1
where ATTR. 1 in (select ATTR. 1
                  from RELAZ. 2)
```

→ in questo caso sceglie tutti gli "I VALORI DI A_1" A_1 di R_1 che sono anche VALORI DI A_1 di R_2

→ (SI PUÒ USARE ANCHE not in!)

9. SOME:

```
select ATTR. 1
from REL
where ATTR. 2 > some (select ATTR. 2
                      from REL
                      where P)
```

(some = "ALMENO UNO")

→ in questo caso sceglie gli "(I VALORI DI A_1)" A_1 di R che hanno (nella stessa TUPLA) un valore di A_2 maggiore di almeno uno dei valori di A_2 che soddisfano la condizione P

10. ALL:

```
Select ATTR 1
from REL
where ATTR. 2 > all (select ATTR. 2
                     from REL
                     where P)
```

(all = "TUTTI")

→ come some, ma al posto che "almeno uno" significa "tutti"

11. EXISTS:

```
Select ATTR 1
from REL 1
where exists (select ATTR 2
              from REL 2
              where P)
```

($ATTR_1$ e $ATTR_2$ sono entrambi di R_1)

→ sceglie (I VALORI DI) $ATTR_1$ che nella TUPLA hanno VALORI DI A_2 che soddisfano la condizione P_2

→ (SI PUÒ USARE ANCHE not exists ED except!)

12. UNIQUE:

```
Select ATTR 1
from REL 1
Where Unique select ATTR 2
             from REL 2
             Where P
```

→ sceglie i valori di A_1 che hanno nella stessa TUPLA dei valori di A_2 che sono UNICI nelle TUPLE DI R_2

→ (not unique!!! anche)

↳ [Unique (...) è VERO ⟺ se nella subquery non ci sono risultati duplicati nelle TUPLE]

Posso usare subqueries anche in "from" → DERIVED RELATIONS: [select ... from (subquery...)]

[SQL:
- DDL : CREATE/ALTER/DROP/RENAME/TRUNCATE/COMMENT
- DML : INSERT/UPDATE/DELETE/MERGE
- DQL : SELECT/JOIN/SUBQUERY
- DCL : GRANT/REVOKE
- TCL : COMMIT/ROLLBACK/SAVEPOINT] → (VEDI APPENDICE "CLASSIFICAZIONE ISTRUZIONI SQL")

3. DML (DATA MODIFICATION LANGUAGE)

① DELETE :

```
delete from RELAZ.
  where P
```

→ cancella tutte le TUPLE dalla RELAZ dove P
↳ condizionale del [comando ① visto in 3.1 (PG 8)]

② INSERT : [comando ③ visto in 3.1 (PG 8)] } condizionale

Posso anche usare SUBQUERY :

```
insert into RELAZ1
  select (ATTR1, ..., ATTRm)
    from RELAZ2
    where P
```

③ UPDATE (SET) :

```
update RELAZ
  set ATTR.1 = (...)
  where P
```

→ es (set ATTR.1 = (ATTR.1) - 2)

↳ CASE :

```
update RELAZ
set ATTR.1 = case
               when P then ATTR.1 = (...)
               else ATTR.1 = (...)
             end
```

4. More DQL: JOINED RELATIONS

Prendiamo due tabelle r(A,B,C) e s(D,E) così fatte:

r:

A	B	C
7	Q	α
3	R	β
6	P	γ

s:

D	E
J	7
S	3
h	5

CROSS JOIN: semplice PRODOTTO CARTESIANO del RAPPORTO tra le due tabelle

INNER JOIN: restituisce il risultato combinato di tutte le tuple che trovano corrisp. nelle tabelle (in tutte) } es: r INNER JOIN s ON r.A = s.E ⇒

A	B	C	D	E
7	Q	α	J	7
3	R	β	S	3

OUTER JOIN: seleziona i risultati anche in assenza di uno dei due (DESTRA, SINISTRA) o entrambi (FULL) lati corrispondenti } es: r LEFT OUTER JOIN s ON r.A = s.E ⇒

A	B	C	D	E
7	Q	α	J	7
3	R	β	S	3
6	P	γ	null	null

clausola ON: specifica su quale attributo, per ogni tabella, bisogna fare il JOIN

Prendiamo due tabelle r(A,B,C) e s(D,A) così fatte:

r:

A	B	C
7	Q	α
3	R	β
6	P	γ

s:

D	A
J	7
S	3
h	5

clausola NATURAL: esegue il JOIN sull'attributo comune (non c'è bisogno di mettere ON! Rispetto all'INNER JOIN unisce poi le due colonne in una) } es: r NATURAL INNER JOIN s ⇒

A	B	C	D	A
7	Q	α	J	7
3	R	β	S	3

(INNER JOIN ON s.A = r.A AVREBBE LASCIATO QUESTA COLONNA)

es: r NATURAL LEFT OUTER JOIN s ⇒

A	B	C	D
7	Q	α	J
3	R	β	S
6	P	γ	null

clausola USING : se voglio eseguire un NATURAL JOIN (quindi senza specificare "ON") ma ho più attributi uguali (ad es: r(A,B,C,D) e s(D,A)) con USING specifico su quale eseguirlo (es: r INNER JOIN s USING (D))

5. Views

VIEW: relazione che non è parte del modello concettuale ma è visibile all'utente come una "relazione virtuale" [create view v as <espressione query>]

es: ho 4 relazioni (depositor, account, borrower, loan) legate due a due (chi deposita ha un conto, chi chiede un prestito ha un debito). Voglio nascondere tutte le info contenute in queste relazioni per avere solo una lista di tutti i clienti e delle rispettive filiali (branch):

```
create view all_customers as
    (select branch_name, customer_name
        from depositor, account
        where depositor.account_ID = account.account_ID)
    union
    (select branch_name, customer_name
        from borrower, loan
        where borrower.loan_ID = loan.loan_ID)
```

Ho creato una RELAZIONE (TABELLA) VIRTUALE:

branch_name	customer_name
(...)	(...)

(all_customers)

Posso quindi usarla:

```
select customer_name
  from all_customers
  where branch_name = (...)
```

Si usano
- per nascondere informazioni sensibili o non necessarie all'utente finale.
- per rendere la scrittura del codice più facile e veloce (→ queries predefinite)

Una vista V_1
- DIPENDE DIRETTAMENTE da V_2 se nella sua definizione è presente V_2
- DIPENDE da V_2 se dipende direttamente o c'è un "percorso di dipendenza" tra le due
- è RICORSIVA se DIPENDE DIRETTAMENTE da se stessa.

VIEW EXPANSION: sostituzione, in una espressione di codice SQL, di tutte le views con le corrispettive subquery, fino a che non rimangono più views.

WITH:

es:
```
WITH max_balance as  select max(balance)
                     from account          -- VISTA TEMPORANEA

select account_ID
from account, max_balance
where account.balance = max_balance.value
```

definisce una view temporanea che è definita e disponibile solo nella query dove la WITH è presente

AGGIORNAMENTO DI UNA VIEW:

(loan):

loan_ID	branch_name	amount
(...)	(...)	(...)
L-37	Perryridge	null

→ view: (loan_branch):

loan_ID	branch_name
(...)	(...)
L-37	Perryridge

[VIEW, serve per nascondere l'amount]

[inserisco una tupla nella VIEW]:

```
insert into loan_branch
    values ('L-37', 'Perryridge')
```

(lo aggiunge anche nella relazione "VERA" e lascia null nei valori nascosti.) !

6. Valori "null"

null: significa un valore sconosciuto o che non esiste

is null: serve per verificare se un valore è null →

```
select ATTR_1
from RELAZ
where ATTR_2 is null
```

AZIONI CON IL null: →(CONFRONTA CON PAR 4, PG 7)

- ↳ AGGREGATE FUNCTIONS: ignorano il "null", ad eccezione di count(*)
- ↳ ESPRESSIONI ARITMETICHE: restituiscono il "null"
- ↳ COMPARAZIONI (es: null > 5): restituiscono "null"
- ↳ LOGICA A TRE VALORI:
  - UNK ∨ T = T ; UNK ∨ F = UNK ; UNK ∨ UNK = UNK
  - UNK ∧ T = UNK ; UNK ∧ F = F ; UNK ∧ UNK = UNK
  - NOT UNK = UNK
- ↳ "P is UNKNOWN" valuta se il predicato P è UNK, se sì restituisce T, se no F.
- ↳ WHERE: il predicato di where P è giudicato "falso" F se restituisce come valore UNK.

7. "Between"

È un comparatore aggiuntivo:

es:
```
select loan_number
from loan
where amount between 90'000 and 100'000
```

Cap4: Advanced SQL

1. SQL Data Types

(current_date = data di oggi)

DATA: [date 'yyyy-mm-dd'] → (4 cifre l'anno)   TEMPO: [time 'hh:mm:ss(.dc)'] → OPZIONALI

TIMESTAMP: (DATA + TEMPO): [timestamp 'yyyy-mm-dd hh:mm:ss.dc']

INTERVALLO Δt: [interval '1' day] ← (ad es) → (può essere sommato alla data/tempo/timestamp)
↳ (la sottrazione tra date/tempi/timestamps restituisce un Δt).

ESTRAZIONE del valore di un singolo campo da date/tempi/timestamps: [extract(year from r.starttime)] (r = RELAZIONE, starttime = ATTRIBUTO)

CAST di stringhe: [cast <valore-data-in-stringa> as time/date/timestamp]

↑ BUILT-IN   ↓ USER-DEFINED

TIPI PERSONALIZZATI: [create type Dollars as numeric(12,2) final]

DOMINI PERSONALIZZATI: [create domain Person-Name char(20) not null] ← VINCOLO (CONSTRAINT)
↳ tipi di dati derivati, basati sui tipi semplici, cui si possono associare particolari vincoli o informazioni

(LARGE-OBJECTS) ↓

BLOB (Binary Large OBjects) = sequenza di bit di elevate dimensioni (MB o GB).

CLOB (Character " ") = " di caratteri " " .

2. Vincoli (Constraints)

VINCOLI AL DOMINIO: testano i valori inseriti e le query per assicurarsi che le comparazioni continuino ad avere senso

VINCOLI DI INTEGRITÀ: fanno in modo che non si siano danni accidentali al DB, assicurando che le modifiche autorizzate al DB non si traducano in una perdita di consistenza/coerenza dei dati.

NOT NULL: dichiarazione di ATTRIBUTI: [Attr1 char(15) not null]
" " di DOMINI: [create domain D numeric(12,2) not null]

UNIQUE: determina gli attributi che sono CHIAVI CANDIDATE (→ possono essere null, la PRIMARY KEY scelta NO): [Unique(A1, ..., Am)]

CHECK(P): (P è un predicato) → permette che il dominio sia ristretto (ad esempio:

```
create domain hourly_wage numeric(5,2)
constraint value_test check (value >= 4.00)
```

(Assicura che la retribuzione all'ora minima sia almeno 4 Dollari l'ora)
↳ constraint indica quale vincolo un aggiornamento dei dati può violare, è

il "nome del vincolo". Non è necessario, ma utile.

INTEGRITÀ REFERENZIALE: assicura che un valore che appare in una relazione per un determinato insieme di attributi appaia anche per un certo insieme di attributi in un'altra relazione → (es: se esiste "Verona" come nome_divisione in una tupla della relazione impiegato, esisterà anche il valore "Verona" nello stesso attributo nome_divisione della relazione divisione.)

CHIAVI
- primary key($A_1,\dots,A_n$): elenca attributi che compongono la chiave primaria
- unique key($A_1,\dots,A_n$): elenca attributi che compongono una chiave candidata
- foreign key($A_1,\dots,A_n$) references(r): elenca attributi che compongono una chiave esterna alla relazione, e la relazione r che identificano → di default è la primary key di r.

[esempi su slide 4.12 - 4.15 - 4.16] !!! IMPORTANTI

ASSERZIONI: predicati che esprimono una condizione che vogliamo il DB soddisfi sempre.

[create assertion <nome-asserzione> check <predicato>]

una volta creata, il sistema controlla la sua validità, e la ricontrolla ad ogni aggiornamento dei dati → se sono molte il sistema è lento!

↳ per creare una asserzione del tipo [$\forall x, P(x)$] si usa $\nexists x$ t.c. not $P(x)$] → [VEDI ESEMPIO SLIDE 4.18]

3. DCL (DATA CONTROL LANGUAGE)

GRANT: [grant <lista dei privilegi> on <nome della relazione o della view> to <lista utenti>]

PRIVILEGI: select: permette di leggere i dati della relazione, o l'abilità di creare interrogazioni (query) mediante la view

insert: permette di inserire tuple nella relazione/vista

delete: " " eliminare " " " / "

all privileges → questi e gli altri del CAP.8

LISTA UTENTI: è composta da user-id oppure dalla parola "public", che permette i privilegi a tutti gli utenti.

(I privilegi su una VIEW NON sono trasmessi anche alla RELAZIONE SOTTOSTANTE)!!!

REVOKE: [revoke <lista privilegi> on <relaz/view> to <lista utenti>]

↳ "all" per togliere tutti quelli garantiti

4. EMBEDDED SQL

SQL può essere "ospitato" (HOSTED) in un linguaggio di programmazione (C, Java, Cobol ...)
↳ le strutture dell'SQL rimane nell'HOST LANGUAGE come l'EMBEDDED SQL.

EXEC SQL: comando (ma varia a seconda dei linguaggi) usato per identificare le richieste embedded SQL al preprocessore: [EXEC SQL <embedded SQL statement> END_EXEC]

esempio:
```
EXEC SQL
    declare c cursor for
        <SQL statement (select, from, where ...)>
END_EXEC
```
BISOGNA DICHIARARE UN CURSORE "C" PER QUESTA QUERY

QUESTI STATEMENT VARIANO A SECONDA DEI LINGUAGGI DI PROGRAMMAZIONE

OPEN c: valuta la query identificata dal cursore c

FETCH c: mette i risultati della query identificata nelle variabili del linguaggio di programmazione [EXEC SQL fetch c into :Var_1, :Var_2 (... ecc) END_EXEC]
(c = CURSORE; Var_1, Var_2 = VARIABILI)

Una variabile chiamata SQLSTATE nell'area comunicazione di SQL (SQLCA ↔ COMMUNICATION AREA) assume il valore '02000' per indicare che non vi sono più dati disponibili.

CLOSE c: fa eliminare la relazione temporanea che contiene il risultato della query c.

↳ [sia OPEN c che CLOSE c vanno messi dentro a EXEC]

AGGIORNAMENTI TRAMITE CURSORE:

```
declare c cursor for
    select *
    from account
    where branch_name = "Genova"
for update !
```
→ il cursore c ha dei valori dati dalla sua query, ovvero le tuple contenenti il valore "Genova"

→ Voglio aggiornare un valore di un'altro attributo di queste tuple (es: balance) →

```
update account
set balance = balance + 100
where current of c !
```

SQL DINAMICO: permette ai programmi di eseguire SQL query a runtime.

[Esempio in C:] →

```
char* sqlprog = "update account, set balace = balance * 1,05, where account_number = ?"
EXEC SQL prepare dynprog from :sqlprog;
char account [10] = "A-101";
EXEC SQL execute dynprog using :account
```
? è un PLACE HOLDER PER UN VALORE CHE CHE VERRÀ DATO MENTRE IL PROGRAMMA È IN ESECUZIONE

5. ODBC (Open DataBase Connectivity)

ODBC e JDBC sono due API (Application-Program Interface) per un programma al fine di interagire con un DB server. L'applicazioni effettua (1) chiamate per connettersi al DB server (2), invia comandi SQL ad esso ed (3) inserisce i dati delle tuple uno ad uno nelle variabili del programma.

→ ODBC (C#, C++, C) , JDBC (Java)

INTRO ODBC e JDBC ↵

(1) Ogni DBMS che supporta ODBC fornisce una "driver library" che deve essere connesso con il programma client.

(2) Quando il client fa una chiamata ODBC API il codice nella libreria comunica con il server per portare a termine l'azione richiesta ed ottenere i risultati.

(3) Il programma ODBC prima alloca un ambiente SQL (→ HSTMT ???), poi verrà gestita la connessione con il DB.

(4) ODBC inizia la connessione con "SQLConnect()", (parametri) in base alla capacità della connessione (CONNECTION HANDLE), al server bersaglio, all'user_ID ed alla password

↳ bisogna specificare anche il tipo di argomenti (SQL_NTS indica che il precedente argomento è un NullTerminated String)

[→ Esempio Importante SLIDE 4.32] !

(5) Il Programma invia comandi SQL al DB usando il comando "SQLExecDirect"

(6) Le tuple del risultato sono ottenute (FETCHED) usando "SQLFetch()"

(7) "SQLBindCol()" lega le variabili del linguaggio C agli attributi del risultato della query ↳

SQLBindCol(<statement handle>, <column Number>, <target_Type>, <target value Ptr>, <Buffer Lenght>, <StrLen_or_Ind>)

- statement handle → TIPO: "HSTMT"
- column Number → TIPO "SQLU SMALLINT"
- target_Type → TIPO "SQL SMALLINT"
- target value Ptr → TIPO: "SQLPOINTER"
- Buffer Lenght → TIPO "SQLLEN"
- StrLen_or_Ind → TIPO "SQLLEN*"

[PER DETTAGLI VEDI SITO MICROSOFT] ✓

Altre funzioni di ODBC:

STATEMENTS PREIMPOSTATI: sono già compilati nel database, basta inserire i valori nei place holders → [STMT (?,?,?)]

FUNZIONI METADATA: trova tutte le relazioni nel DB e i nomi ed i tipi delle colonne di un risultato di una query o di una relazione nel DB

AUTOCOMMIT DELLE TRANSACTION CHE??? SLIDE 4.35

LIVELLO DI CONFORMITÀ (CONFORMANCE): specifica i sottoinsiemi della funzionalità definita nello standard

↳ "CORE"
↳ "LEVEL 1": richiede supporto per le query di tipo METADATA, cataloghi
↳ "LEVEL 2": richiede l'abilità di inviare e ricevere array di valori di parametri e informazioni più dettagliate.

SQLCLI (CALL LEVEL INTERFACE): standard simile all'interfaccia ODBC, ma con qualche piccola differenza.

6. JDBC (JAVA DATABASE CONNECTIVITY)

JDBC è l'API JAVA per la connettività con i DBMS che supportano SQL. Supporta numerose funzioni per richiedere e modificare i dati. Supporta inoltre la richiesta di METADATI.

Come comunica col DB: ① Stabilisce una connessione ② crea un oggetto STATEMENT ③ esegue le query usando l'oggetto "Statement" per inviarle e raccoglierne i dati ④ gestisce le eccezioni con un EXCEPTION HANDLE MECHANISM.

[ESEMPI SU SLIDES 4.38 - 4.40]?

7. FUNZIONI E PROCEDURE

SQL è fornito di un "MODULE LANGUAGE" che permette la definizione di PROCEDURE in SQL, mediante if-then-else-while etc... che possono essere MEMORIZZATE NEL DB, ed eseguite con il comando call.
↳ permettono ai programmi esterni di operare nel DB senza conoscerne i dettagli interni

Alcuni DBMS supportano le TABLE-VALUED FUNCTIONS, ovvero funzioni che restituiscono una relazione come risultato.

```
create function <nome-funz.> (argomenti)
   returns <tipo>
   begin
        <Statement>
        return(...)
   end
```

```
create procedure <nome-proc.> (in <arg. input>,
                               out <arg. output>)
   begin
        <Statement>
   end
```

[Esempi IMPORTANTI di funzioni: SLIDE 4.43, table-functions: SLIDE 4.44, procedure: SLIDE 4.46]

SOLO LE PROCEDURE!!

FUNZIONI: definite con <SQL statements> "normali"

PROCEDURE: " " <" "> che implementano COSTRUTTI PROCEDURALI:

- WHILE: [while (P) do <Statement> end while]

```
es: while (n<10) do
        set n=n+1
    end while
```

```
es: repeat
        set n=n+1
    until n=10
    end repeat
```

- REPEAT: [repeat <Statement> until (P) end repeat]
- FOR: [for r as <SQL Statement che restituisce una relazione> do <Statement> end for]

```
es: for r as
        select balance from account
        where branch_name = "Verona"
    do
        set n=n+r.balance
    end for
```

- IF-THEN-ELSE: [if P_1 then <statement>, elseif P_2 Then <Statement>, else <statement>, end if]
  - ↳ (CASE: disponibile da SQL:1999)

EXCEPTION HANDLING (GESTIONE ECCEZIONI):

```
es: declare out_of_stock condition
    declare exit handler for out_of_stock
    begin
        <Statement>
        signal out_of_stock
    end
```

Da SQL:1999 è possibile usare funzioni e procedure scritte in altri linguaggi!
↳ EXTERNAL LANGUAGE FUNCTIONS & PROCEDURES -> va dichiarato però il language
↳ [Esempio IMPORTANTE SLIDE 4.50]
↳ usare il linguaggio esterno rende alcune operazioni più efficienti, ma porta dei rischi di complessità del DB (rischio di corruzione accidentale del DB, di sicurezza, accesso a dati non autorizzati, ecc...) -> ci sono delle ALTERNATIVE.

Sono usati i linguaggi esterni quando l'efficienza è più importante della sicurezza. Se no si possono utilizzare delle SANDBOX o dei programmi che dall'ESTERNO svolgono le FUNZIONI / PROCEDURE (> sicurezza, < efficienza)

8. Ricorsione in SQL.

SQL:1999 permette la definizione di VIEW RICORSIVE -> rendono possibili query che non possono essere scritte senza la ricorsione o l'iterazione (ad esempio ricavo la CHIUSURA TRANSITIVA di una relazione.

Le VIEW RICORSIVE DEVONO ESSERE MONOTONE (se ????

9. Funzioni Avanzate

LIKE: Permette di creare tabelle con LO STESSO SCHEMA di altre già esistenti:

[create table <nome> like <relazione>]

[BOH ...]

LATERAL: Permette alle subquery nelle "from" di accedere agli attributi delle altre relazioni contenute nel "from":

```
es.: from r1
        lateral (select (*)
                 from r2
                 where r2.A1 = r1.A2)
     (...)
```

MERGE: (...)

Cap 4: Advanced SQL (English version)

1 Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** ‘2005-7-27’
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** ‘09:00:30’ **time** ‘09:00:30.75’
- **timestamp**: date plus time of day
  - Example: **timestamp** ‘2005-7-27 09:00:30.75’
- **interval:** period of time
  - Example: interval ‘1’ day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

- Can extract values of individual fields from date/time/timestamp
  - Example: **extract** (**year from** r.starttime)

- Can cast string types to date/time/timestamp
  - Example: **cast** <string-valued-expression> **as date**
  - Example: **cast** <string-valued-expression> **as time**

1.1 User-Defined Types

- **create type** construct in SQL creates user-defined type
  create type *Dollars* **as numeric (12,2) final**
- **create domain** construct in SQL-92 creates user-defined domain types
  create domain *person_name* **char**(20) **not null**
- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.

2 Constraints

2.1 Domain Constraints

- **Domain constraints** are the most elementary form of integrity constraint.
  They test values inserted in the database, and test queries to ensure that the comparisons make sense.
- New domains can be created from existing data types
  - Example: **create domain** *Dollars* **numeric**(12, 2)
    create domain *Pounds* **numeric**(12,2)
- We cannot assign or compare a value of type Dollars to a value of type Pounds.
  - However, we can convert type as below
    (**cast** *r.A* **as** *Pounds*)
    (Should also multiply by the dollar-to-pound conversion-rate)

2.2 Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object -- object is a large collection of character data
  - When a query returns a large object, a pointer is returned rather than the large object itself.

2.3 Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $4.00 an hour
  - A customer must have a (non-null) phone number

2.4 Constraints on a Single Relation:

- **not null**
- **primary key**
- **unique**
- **check** (*P*), where *P* is a predicate

2.4.1 Not Null Constraint

- Declare *branch_name* for *branch* is **not null**

  branch_name **char**(15) **not null**
- Declare the domain *Dollars* to be **not null**

  create domain *Dollars* **numeric**(12,2) **not null**

2.4.2 The Unique Constraint

- **unique** (A_1, A_2, …, A_m)
- The unique specification states that the attributes

  *A*1, *A*2, … *Am*

  form a candidate key.
- Candidate keys are permitted to be null (in contrast to primary keys).

2.4.3 The check clause

- **check** (*P*), where *P* is a predicate
- Example: Declare *branch_name* as the primary key for *branch* and ensure that the values of *assets* are non-negative.

```
create table branch
        (branch_name    char(15),
         branch_city    char(30),
         assets         integer,
         primary key (branch_name),
         check (assets >= 0))
```

- The **check** clause in SQL-92 permits domains to be restricted:
  - Use **check** clause to ensure that an hourly_wage domain allows only values greater than a specified value.

```
create domain hourly_wage numeric(5,2)
        constraint value_test check(value > = 4.00)
```

  - The domain has a constraint that ensures that the hourly_wage is greater than 4.00
  - The clause **constraint** *value_test* is optional; useful to indicate which constraint an update violated.

2.5 Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If "Perryridge" is a branch name appearing in one of the tuples in the *account* relation, then there exists a tuple in the *branch* relation for branch "Perryridge".
- Primary and candidate keys and foreign keys can be specified as part of the SQL **create table** statement:
  - The primary key clause lists attributes that comprise the primary key.
  - The unique key clause lists attributes that comprise a candidate key.
  - The foreign key clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key. By default, a foreign key references the primary key attributes of the referenced table.

```
create table customer
    (customer_name    char(20),
     customer_street  char(30),
     customer_city    char(30),
     primary key (customer_name ))

create table account
    (account_number char(10),
     branch_name    char(15),
     balance        integer,
     primary key (account_number),
     foreign key (branch_name) references branch )

create table branch
    (branch_name   char(15),
     branch_city   char(30),
     assets        numeric(12,2),
     primary key (branch_name ))

create table depositor
    (customer_name  char(20),
     account_number  char(10),
     primary key (customer_name, account_number),
     foreign key (account_number ) references account,
     foreign key (customer_name ) references customer )
```

2.6 Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- An assertion in SQL takes the form

  create assertion <assertion-name> **check** <predicate>
- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
  - This testing may introduce a significant amount of overhead; hence assertions should be used with great care.
- Asserting

  for all *X*, *P*(*X*)

  is achieved in a round-about fashion using

  not exists *X* such that not *P*(*X*)

- Every loan has at least one borrower who maintains an account with a minimum balance or $1000.00

```
create assertion balance_constraint check
   (not exists (
      select *
      from loan
      where not exists (
         select *
         from borrower, depositor, account
         where loan.loan_number = borrower.loan_number
            and borrower.customer_name = depositor.customer_name
            and depositor.account_number = account.account_number
            and account.balance >= 1000)))
```

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

```
create assertion sum_constraint check
    (not exists (select *
                 from branch
                 where (select sum(amount )
                        from loan
                         where loan.branch_name =
                               branch.branch_name )
                    >= (select sum (amount )
                        from account
                         where loan.branch_name =
                               branch.branch_name )))
```

3 Data Control Language (DCL)

3.1 Authorization

Forms of authorization on parts of the database:

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

Forms of authorization to modify the database schema (covered in Chapter 8):

- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.

3.1.1 Authorization Specification in SQL

- The **grant** statement is used to confer authorization

    grant <privilege list>

    on <relation name or view name> **to** <user list>
- <user list> is:
    - a user-id
    - **public**, which allows all valid users the privilege granted
    - A role (more on this in Chapter 8)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

3.2 Privileges in SQL

- **select:** allows read access to relation,or the ability to query using the view
    - Example: grant users U_1, U_2, and U_3 **select** authorization on the *branch* relation:

        grant select on *branch* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.

- **all privileges**: used as a short form for all the allowable privileges
- more in Chapter 8

3.2.1 Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke <privilege list>

  on <relation name or view name> **from** <user list>
- Example:

  revoke select on *branch* **from** U_1, U_2, U_3
- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

4 EMBEDDED SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol.
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL.
- The basic form of these languages follows that of the System R embedding of SQL into PL/I.
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

  EXEC SQL <embedded SQL statement > END_EXEC

  Note: this varies by language (for example, the Java embedding uses
  # SQL { …. };)

- From within a host language, find the names and cities of customers with more than the variable amount dollars in some account
- Specify the query in SQL and declare a *cursor* for it

```
EXEC SQL
  declare c cursor for
  select depositor.customer_name, customer_city
  from depositor, customer, account
  where depositor.customer_name = customer.customer_name
      and depositor account_number = account.account_number
      and account.balance > :amount
END_EXEC
```

- The **open** statement causes the query to be evaluated

  EXEC SQL **open** *c* END_EXEC
- The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.

  EXEC SQL **fetch** *c* **into** *:cn, :cc* END_EXEC

  Repeated calls to **fetch** get successive tuples in the query result
- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available
- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

  EXEC SQL **close** *c* END_EXEC

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.

4.1 UPDATES THROUGH CURSORS

- Can update tuples fetched by cursor by declaring that the cursor is for update

  - **declare** *c* **cursor for**
    select *
    from *account*
    where *branch_name* = 'Perryridge'
    for update

- To update tuple at the current location of cursor *c*

  - **update** *account*
    set *balance* = *balance* + 100
    where current of *c*

4.2 DYNAMIC SQL

- Allows programs to construct and submit SQL queries at run time.
- Example of the use of dynamic SQL from within a C program.

```
char *  sqlprog = "update account
                   set balance = balance * 1.05
                    where account_number = ?"
EXEC SQL prepare dynprog  from :sqlprog;
char account [10] = "A-101";
EXEC SQL execute dynprog using :account;
```

- The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.

5 ODBC

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables
- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
- JDBC (Java Database Connectivity) works with Java

- Open DataBase Connectivity(ODBC) standard
  - standard for application program to communicate with a database server.
  - application program interface (API) to
    - open a connection with a database,
    - send queries and updates,
    - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC

- Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.
- ODBC program first allocates an SQL environment, then a database connection handle.

- Opens database connection using SQLConnect(). Parameters for SQLConnect:
  - connection handle,
  - the server to which to connect
  - the user identifier,
  - password
- Must also specify types of arguments:
  - SQL_NTS denotes previous argument is a null-terminated string.
- **int ODBCexample()**

```
{
  RETCODE error;
  HENV    env;     /* environment */
  HDBC    conn;  /* database connection */
  SQLAllocEnv(&env);
  SQLAllocConnect(env, &conn);
  SQLConnect(conn, "aura.bell-labs.com", SQL_NTS, "avi", SQL_NTS, "avipasswd", SQL_NTS);
  { .... Do actual work ... }

  SQLDisconnect(conn);
  SQLFreeConnect(conn);
  SQLFreeEnv(env);
}
```

- Program sends SQL commands to the database by using SQLExecDirect
- Result tuples are fetched using SQLFetch()
- SQLBindCol() binds C language variables to attributes of the query result
  - When a tuple is fetched, its attribute values are automatically stored in corresponding C variables.
  - Arguments to SQLBindCol()
    - ODBC stmt variable, attribute position in query result
    - The type conversion from SQL to C.
    - The address of the variable.
    - For variable-length types like character arrays,
      - The maximum length of the variable
      - Location to store actual length when a tuple is fetched.
      - Note: A negative value returned for the length field indicates null value
- Good programming requires checking results of every function call for errors; we have omitted most checks for brevity.

- Main body of program

```
char branchname[80];
float  balance;
int  lenOut1, lenOut2;
HSTMT   stmt;
SQLAllocStmt(conn, &stmt);
char * sqlquery = "select branch_name, sum (balance)
                   from account
                   group by branch_name";
error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
if (error == SQL_SUCCESS) {
    SQLBindCol(stmt, 1, SQL_C_CHAR,   branchname , 80, &lenOut1);
    SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance,        0 , &lenOut2);
    while (SQLFetch(stmt) >= SQL_SUCCESS) {
        printf (" %s  %g\n", branchname, balance);
    }
}
SQLFreeStmt(stmt, SQL_DROP);
```

5.1 ODBC More features

- **Prepared Statement**
  - SQL statement prepared: compiled at the database
  - Can have placeholders: E.g. insert into account values(?,?,?)
  - Repeatedly executed with actual values for the placeholders
- **Metadata features**
  - finding all the relations in the database and
  - finding the names and types of columns of a query result or a relation in the database.
- By default, each SQL statement is treated as a separate transaction that is committed automatically.
  - Can turn off automatic commit on a connection
    - SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)}
  - transactions must then be committed or rolled back explicitly by
    - SQLTransact(conn, SQL_COMMIT) or
    - SQLTransact(conn, SQL_ROLLBACK)

5.2 ODBC Conformance Levels

- Conformance levels specify subsets of the functionality defined by the standard.
  - Core
  - Level 1 requires support for metadata querying
  - Level 2 requires ability to send and retrieve arrays of parameter values and more detailed catalog information.
- SQL Call Level Interface (CLI) standard similar to ODBC interface, but with some minor differences.

6 JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL
- JDBC supports a variety of features for querying and updating data, and for retrieving query results
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes
- Model for communicating with the database:
  - Open a connection
  - Create a "statement" object
  - Execute queries using the Statement object to send queries and fetch results
  - Exception mechanism to handle errors

JDBC Code example:

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@aura.bell-labs.com:2000:bankdb", userid, passwd);
        Statement stmt = conn.createStatement();
            … Do Actual Work ….
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

- Update to database

```
try {
    stmt.executeUpdate( "insert into account values
                        ('A-9732', 'Perryridge', 1200)");
} catch (SQLException sqle) {
    System.out.println("Could not insert tuple. " + sqle);
}
```

- Execute query and fetch and print results

```
ResultSet rset = stmt.executeQuery( "select branch_name, avg(balance)
                                    from account
                                    group by branch_name");
while (rset.next()) {
    System.out.println(
            rset.getString("branch_name") + "  " + rset.getFloat(2));
}
```

- Getting result fields:
  - **rs.getString("branchname") and rs.getString(1) equivalent if branchname is the first argument of select result.**
- Dealing with Null values

```
int a = rs.getInt("a");
if (rs.wasNull()) Systems.out.println("Got null value");
```

7 Procedural Extensions and Stored Procedures

- SQL provides a **module** language
  - Permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.
  - more in Chapter 9
- Stored Procedures
  - Can store procedures in the database
  - then execute them using the **call** statement
  - permit external applications to operate on the database without knowing about internal details
- These features are covered in Chapter 9 (Object Relational Databases)

7.1 Functions and Procedures

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language
  - Functions are particularly useful with specialized data types such as images and geometric objects
    - Example: functions to check if polygons overlap, or to compare images for similarity
  - Some database systems support **table-valued functions**, which can return a relation as a result
- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999

7.2 SQL Functions

- Define a function that, given the name of a customer, returns the count of the number of accounts owned by the customer.

```
create function account_count (customer_name varchar(20))
returns integer
begin
    declare a_count integer;
    select count (* ) into a_count
    from depositor
    where depositor.customer_name = customer_name
    return a_count;
end
```

- Find the name and address of each customer that has more than one account.

```
select customer_name, customer_street, customer_city
from customer
where account_count (customer_name ) > 1
```

7.3 TABLE FUNCTIONS

- SQL:2003 added functions that return a relation as a result
- Example: Return all accounts owned by a given customer

```
create function accounts_of (customer_name char(20)
        returns table (    account_number char(10),
                           branch_name char(15)
                           balance numeric(12,2))
return table
        (select account_number, branch_name, balance
         from account A
         where exists (
            select *
            from depositor D
            where D.customer_name = accounts_of.customer_name
                  and D.account_number = A.account_number ))
```

- Usage

```
        select *
        from table (accounts_of ('Smith'))
```

7.4 SQL PROCEDURES

- The *author_count* function could instead be written as procedure:

```
create procedure account_count_proc (in title varchar(20),
                                     out a_count integer)
begin
  select count(author) into a_count
  from depositor
  where depositor.customer_name = account_count_proc.customer_name
end
```

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

```
        declare a_count integer;
        call account_count_proc( 'Smith', a_count);
```

  Procedures and functions can be invoked also from dynamic SQL

- SQL:1999 allows more than one function/procedure of the same name (called name **overloading**), as long as the number of arguments differ, or at least the types of the arguments differ

- Compound statement: **begin … end**,
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements

- **While** and **repeat** statements:

```
        declare n integer default 0;
        while n < 10 do
           set n = n + 1
        end while
```

```
repeat
   set n = n – 1
until n = 0
end repeat
```

- **For** loop
  - Permits iteration over all results of a query
  - Example: find total of all balances at the Perryridge branch

```
declare n integer default 0;
for r as
    select balance from account
     where branch_name = 'Perryridge'
do
    set n = n + r.balance
end for
```

- Conditional statements (**if-then-else**)
  E.g. To find sum of balances for each of three categories of accounts (with balance <1000, >=1000 and <5000, >= 5000)

```
if r.balance < 1000
    then set l = l + r.balance
elseif r.balance < 5000
    then set m = m + r.balance
else set h = h + r.balance
end if
```

- SQL:1999 also supports a **case** statement similar to C case statement
- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_stock condition
declare exit handler for out_of_stock
begin
…
.. signal out-of-stock
end
```

  - The handler here is **exit** -- causes enclosing **begin..end** to be exited
  - Other actions possible on exception

7.5 External Language Functions/Procedures

- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions

```
create procedure account_count_proc(in customer_name varchar(20),
                                    out count integer)
language C
external name ' /usr/avi/bin/account_count_proc'

create function account_count(customer_name varchar(20))
returns integer
language C
external name '/usr/avi/bin/author_count'
```

- Benefits of external language functions/procedures:
  - more efficient for many operations, and more expressive power

- Drawbacks
  - Code to implement function may need to be loaded into database system and executed in the database system's address space
    - risk of accidental corruption of database structures
    - security risk, allowing users access to unauthorized data
  - There are alternatives, which give good security at the cost of potentially worse performance
  - Direct execution in the database system's space is used when efficiency is more important than security

7.6 Security with External Language Routines

- To deal with security problems
  - Use **sandbox** techniques
    - that is use a safe language like Java, which cannot be used to access/damage other parts of the database code
  - Or, run external language functions/procedures in a separate process, with no access to the database process' memory
    - Parameters and results communicated via inter-process communication
- Both have performance overheads
- Many database systems support both above approaches as well as direct executing in database system address space

7.7 Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find all employee-manager pairs, where the employee reports to the manager directly or indirectly (that is manager's manager, manager's manager's manager, etc.)

```
with recursive empl (employee_name, manager_name ) as (
        select employee_name, manager_name
        from   manager
    union
        select manager.employee_name, empl.manager_name
        from   manager, empl
        where manager.manager_name = empl.employe_name)
select *
from   empl
```

This example view, *empl,* is called the *transitive closure* of the *manager* relation

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *manager* with itself
    - This can give only a fixed number of levels of managers
    - Given a program we can construct a database with a greater number of levels of managers on which the program will not work

- Computing transitive closure
  - The next slide shows a *manager* relation
  - Each step of the iterative process constructs an extended version of *empl* from its recursive definition.
  - The final result is called the *fixed point* of the recursive view definition.
- Recursive views are required to be *monotonic.* That is, if we add tuples to *manger* the view contains all of the tuples it contained before, plus possibly more

7.7.1 Example of Fixed-Point Computation

employee_name	*manager_name*
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Iteration number	*Tuples in empl*
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)

8 Advanced Features

- Create a table with the same schema as an existing table:

  create table *temp_account* **like** *account*
- SQL:2003 allows subqueries to occur *anywhere* a value is required provided the subquery returns only one value. This applies to updates as well
- SQL:2003 allows subqueries in the **from** clause to access attributes of other relations in the **from** clause using the **lateral** construct:

```
select C.customer_name, num_accounts
from customer C,
     lateral (select count(*)
              from account A
              where A.customer_name = C.customer_name )
          as this_customer (num_accounts )
```

- Merge construct allows batch processing of updates.
- Example: relation *funds_received* (*account_number, amount*) has batch of deposits to be added to the proper account in the *account* relation

```
merge into account as A
    using (select *
           from funds_received as F )
    on (A.account_number = F.account_number )
    when matched then
        update set balance = balance + F.amount
```

Cap 6: Modello E-R

1 ENTITÀ, ATTRIBUTI, RELAZIONI

Un DB può essere modellizzato come una **collezione di entità** e delle **relazioni tra entità**.

Entità e (*entity*)≔ oggetto la cui esistenza è distinta dagli altri oggetti (es: una persona specifica, un evento…)

Ogni entità ha degli **attributi** (*attributes*) (es: nome della persona specifica, suo indirizzo di casa ecc…)

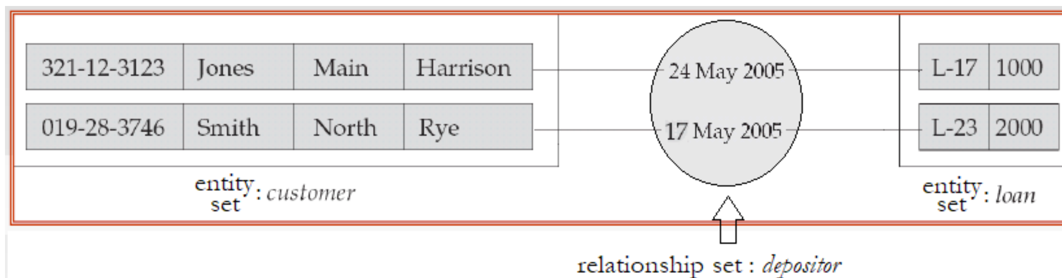
Insieme di entità E (*entity set*) ≔ insieme di entità che condividono le stesse proprietà (es: insieme delle persone che lavorano per una determinata azienda, …)

Relazione (*relationship*) ≔ associazione tra diverse entità $(e_1, \dots, e_n)$

Insieme di relazioni (*relationship set*) ≔ relazione matematica tra $n \geq 2$ entità, ciascuna presa da *entity sets*. Ovvero $\{(e_1, \dots, e_n)\ \ t.c.\ \ e_1 \in E_1, \dots, e_n \in E_n\}$

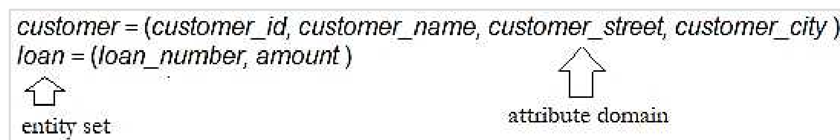
Un attributo può anche essere proprietà di un relationship set (es: il relatioship set *depositor* tra i due entity set *customer* e *loan*[1] può avere l'attributo "data di accesso")

Vediamo un esempio:

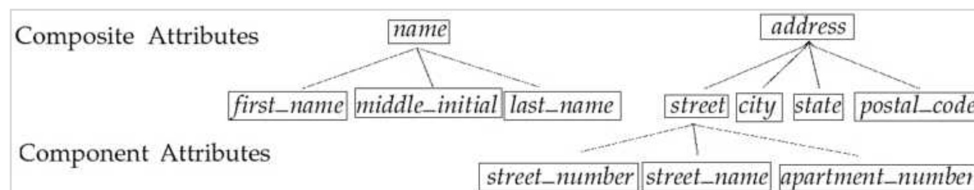


Grado (∂eg) di un Relationship Set ≔ è il numero di entity sets che occorrono nel relationship set (se $\partial eg = 2$ allora la relazione è detta *binaria* → la maggior parte delle relazioni sono così)

Un'entità è rappresentata da un insieme di attributi, descrittivo delle proprietà possedute da tutti i membri dell'entity set → il **dominio dell'attributo** è l'insieme dei valori che l'attributo può assumere. Ad es.:



Tipi di attributi: possono essere singoli o compositi, mono- o multi-valore (es: un attributo *phone_numbers*), primitivi o derivati, possono essere calcolati da altri attributi (es: l'età tramite la data di nascita). Ad esempio:



[1] In teoria quando la relazione è *depositor* dovrebbero esserci *customer* (cliente) ed *account* (conto), ma sulle slides c'era un errore negli schemo, quindi si utilizza *loan* (mutuo/prestito) anche per la relazione *depositor*, e non solo per quella *borrower*.

2 CARDINALITÀ E CHIAVI

I **vincoli di cardinalità** (*cardinality constraints*) esprimono il numero di entità alle quali un'altra entità può essere associate con un relatioship set. Sono molto utili per descrivere i relatioship set binari ($\partial eg = 2$→ per questi sets abbiamo, come tipi vincoli di cardinalità: uno-a-uno, uno-a-molti, molti-a-uno e molti-a-molti).

Chiave (*super key*) **di un entity set** ≔ insieme di uno o più attributi che la identificano univocamente.

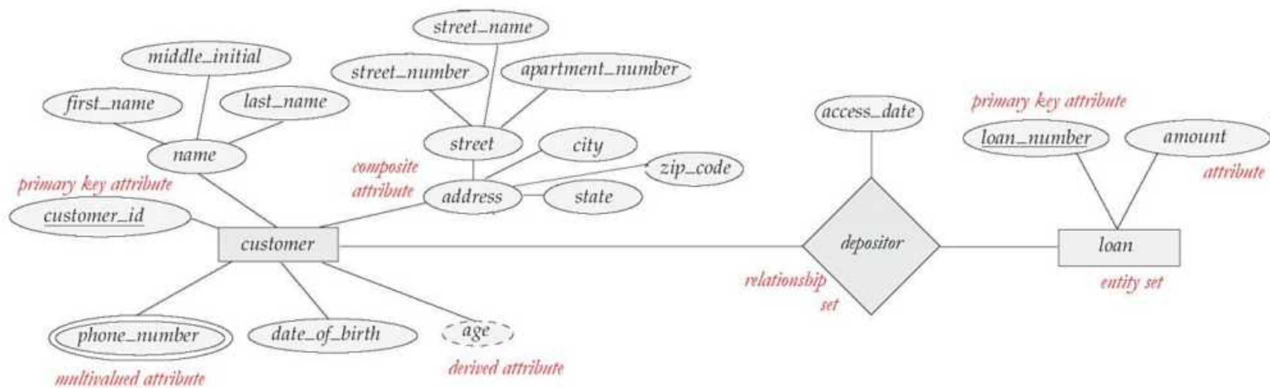
Chiave candidata (*candidate key*) di un entity set ≔ una *super key* **minimale** (→ovvero una super key che non contiene al suo interno nessun'altra super key, ad esempio *Customer_id* per l'entity set *customer*) → Una sola delle chiavi candidate viene scelta **chiave primaria** (*primary key*) dell'entity set.

La combinazione delle primary key degli entity set di un relationship set forma la **chiave** (*super key*) **del relationship set** (ad es: (*customer_id*, *account_number*) è chiave di *depositor*).

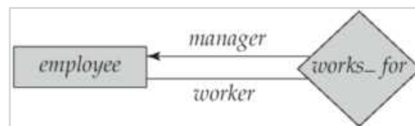
Bisogna tenere presente la cardinalità delle entità nel relationship set quando si decide quali sono le sue **chiavi candidate**. Inoltre bisogna anche aver presente la **semantica** del relationship set quando si valuta quale di quelle si deve scegliere come **chiave primaria**.

3 DIAGRAMMI E-R

Modo grafico per rappresentare i rapporti tra entity sets e relationship sets:

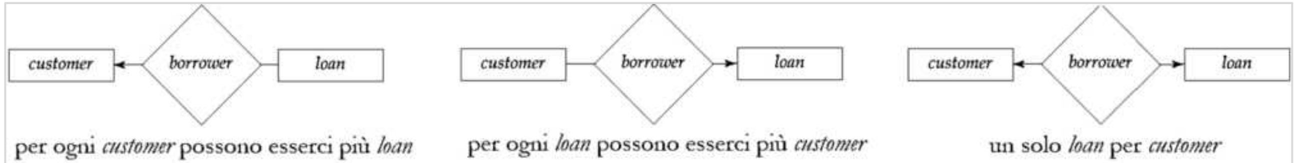


Ruoli: gli entity sets di una relazione non devono necessariamente essere diversi! Affidare delle "etichette" (nell'esempio *manager* e *worker*) aiuta a specificare come i due ruoli dello stesso entity set interagiscano tra loro (le etichette di ruolo sono opzionali, ma aiutano a chiarire la semantica della relazione):



Nell'esempio di qui sopra è stata usata una freccia. Le frecce nei diagrammi E-R indicano la **cardinalità** ("→" significa "uno" e "–" significa "molti"):

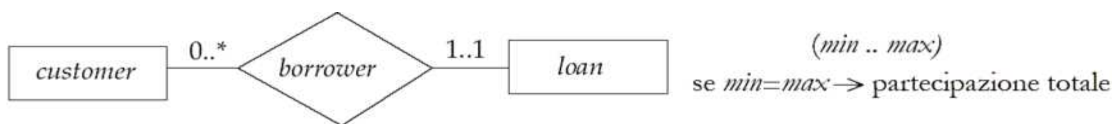
Ad esempio:



Partecipazione totale (*total participation*): è indicata con una <u>doppia linea</u>, vuole dire che ogni entità nell'entity set partecipa in almeno una relazione del relationship set (es: la partecipazione di *loan* in *borrower* è totale, non ci sono prestiti senza clienti)

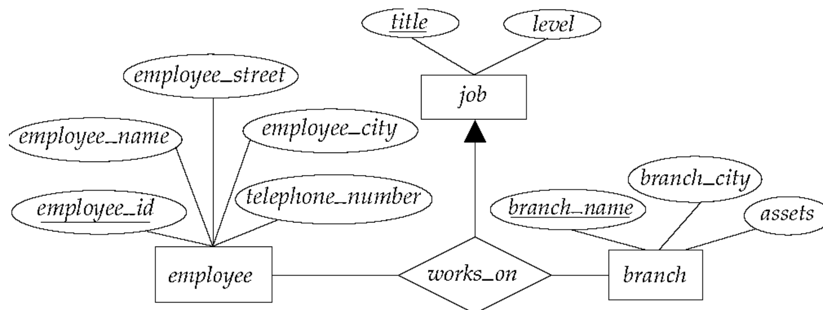
Partecipazione totale (*total participation*): alcune entità possono non partecipare a nessuna delle relazioni nell'entity set (es: la partecipazione di *customer* in *borrower* è parzial, ci sono clienti che non sono*borrowers*, ma magari *depositors*).

Con una <u>notazione alternativa</u> si possono esprimere sia cardinalità che partecipazione:



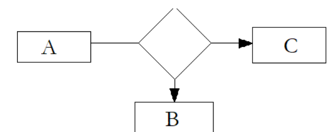
3.1 Cardinalità in relazioni a grado > 2

Nel caso di <u>relazioni *ternarie*</u> (o di grado maggiore), ammettiamo comunque una sola freccia per indicare la cardinalità (nell'esempio sottostante la frecca da *works_on* a *job* indica che ogni *employee* lavora almeno a un *job* in ogni *branch*):



Se ci sono più frecce, ci sono tre modi per definirne il significato (scelta che eviteremo per non fare confusione):

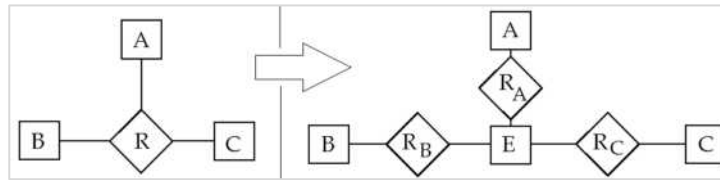
1. Ogni entità A è associata a un'unica entità di B e C
2. Ogni coppia di entità da (A,B) è associata ad un'unica entità di C, ed ogni coppia di (A,C) è associata ad un'unica di B



4 Problemi e dubbi comuni (1)

1. **Uso di entity sets o attributi?** La scelta dipende dalla struttura che deve essere modellata dal DB e dalla semantica associata all'attributo in questione.
2. **Uso di entity sets o relationship sets?** Una possibile linea guida è di costruire un relationship set per descrivere un'azione che accade tra due entità.
3. **Uso di relationship sets binari o *n*-ari?** Sebbene sia possibile sostituire ogni relationship set *n*-ario con un numero di relationship sets binari distinti, gli n-ari mostrano con maggiore chiarezza che diverse entità partecipano ad una singola relazione → vedi dopo.
4. **Inserimento di attributi nei relationship sets? →** vedi dopo.

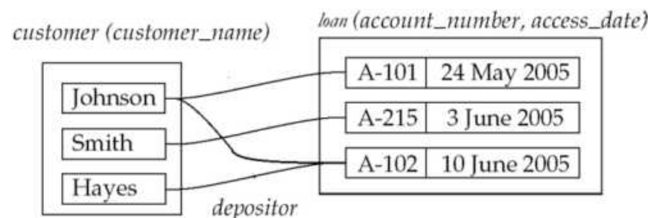
Per quanto riguarda il **punto 3**, in generale, ogni relazione n-aria R può essere rappresentato usando relazioni binarie R_i creando un entity set "artificiale" E che abbia tutti gli attributi di R:



∀ relazione $(a_i, b_i, c_i) \in R$ bisogna creare una nuova entità $e_i \in E$ e aggiungere (e_i, a_i) a R_A, (e_i, b_i) a R_B e (e_i, c_i) a R_C.

Bisogna però anche "tradurre" eventuali vincoli, e non sempre è possibile!

Per quanto riguarda il **punto 4**, in riferimento all'esempio sviluppato nei paragrafi precedenti, è possibile a volte, in base ai vincoli di cardinalità, tradurre un attributo di una relationship set (in questo caso *access_date*) in un attributo di un entity set (in questo caso *loan* → attenzione che ciò è possibile farlo solo se la relazione $\boxed{customer \leftarrow \diamondsuit - loan}$ è una relazione uno-a-molti **!!!**)



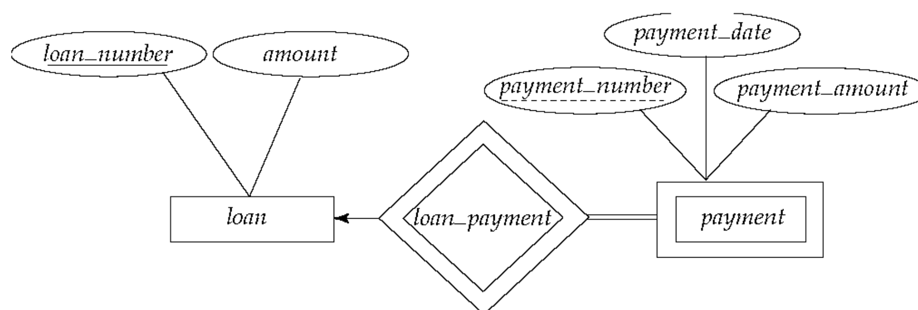
5 ENTITÀ DEBOLI

Entity set debole (*weak*) ≔ entity set che non ha una primary key. La sua esistenza dipende dall'esistenza di **entity set identificatore** (*identifying entity set*) che identifica univocamente, "dall'esterno", l'entità debole. Nel diagramma E-R è identificato con un doppio rettangolo.

L' entity set debole deve essere collegato all'entity set identificatore mediante una **relazione identificatrice** (*identifying relationship*) uno-a-molti (**un** solo identificatore per uno o più entity set deboli). La relazione identificatrice è rappresentata nel diagramma E-R da un doppio rombo.

Il **discriminante**, o **chiave parziale** (*discriminator*, o *partial key*) di un entity set debole è l'insieme di attributi che identificano univocamente le singole entità all'interno dell'entity set debole → La **primary key** di un entity set debole è formata dalla primary key dell'entity set identificatore più il discriminante dell'entity set debole. La partial key è l'attributo sottolineato con il tratteggio.

Vediamo un esempio:

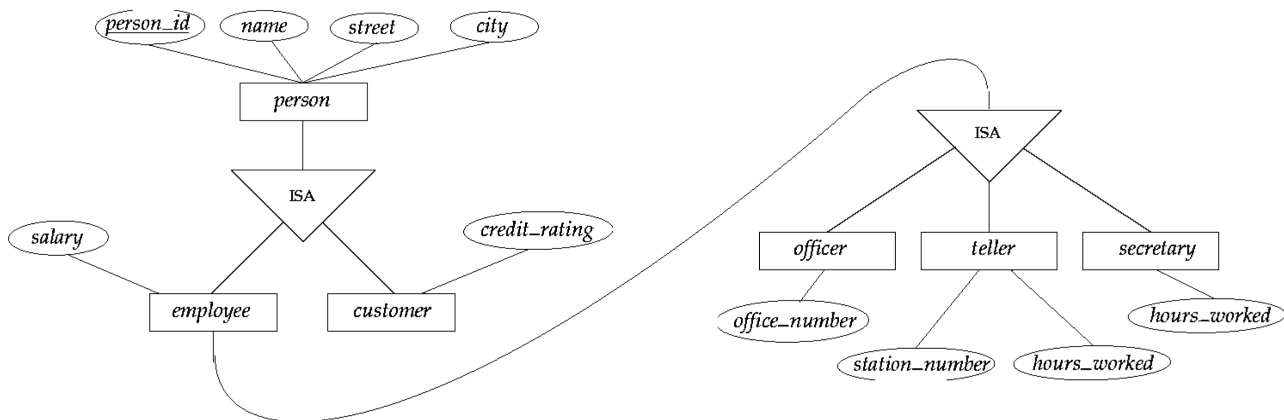


Nota Bene: la primary key dell'entity set identificatore non è esplicitamente memorizzata con l'entity set debole, dato che è implicita nella relazione identificatrice (es: se *loan_number* fosse esplicitamente memorizzato in *payment*, *payment* potrebbe essere un entity set "forte", ma allora la relazione *loan_payment* tra *payment* e *loan*

sarebbe duplicata da una relazione implicita definita dall'attributo *loan_number* presente sia a *payment* che a *loan*!).

6 SPECIALIZZAZIONE E GENERALIZZAZIONE

Specializzazione: **Progettazione top-down**: si progettano sottogruppi di entità in un entity set che sono distinti dalle altre entità del set. Questi sottogruppi diventeranno entity sets di livello inferiore che avranno attributi o parteciperanno a relazioni che non si applicano agli entity sets di livello superiore. Questo processo è indicato con un **triangolo IS-A**[2] (es: *customer* "is a" *person*):



↳ **ereditarietà degli attributi** (*attribute inheritance*): un entity set di livello inferiore eredita tutti gli attributi e la partecipazione alle relazioni degli entity set di livello superiore ai quali è collegato.

Generalizzazione: **Progettazione bottom-up**: si combinano un numero di entity sets che condividono le stesse caratteristiche in un entity set di livello superiore

→ Specializzazione e generalizzazione sono una l'inverso dell'altra, sono rappresentati nel diagramma E-R allo stesso modo (triangolo IS-A) → i termini sono intercambiabili.

Si possono avere specializzazioni molteplici di un entity set basate sulle sue differenti caratteristiche (es: *permanent_employee* vs. *temporary_employee*, in aggiunta alla specializzazione *officer* vs. *secretary* vs. *teller*)

6.1 DEFINIZIONE DEI VINCOLI NELLE SPECIALIZZAZIONI/GENERALIZZAZIONI

Vincoli su quali entità possano essere membri di un dato entity set a livello inferiore e quali no. Possono essere:

- Definiti dalle condizioni, (*condition-defined*) → es: tutti i *customers* con più di 65 anni sono membri del *senior-citizen* entity set; e *senior-citizen* ISA *person*.
- Definiti dagli utenti (*user-defined*)

Vincoli su quali entità, all'interno di una singola generalizzazione, appartengano a più di un entity set di livello inferiore e quali no:

- **Disjoint** (disgiunzione): un'entità può appartenere solo ad un solo entity set di livello inferiore (nel diagramma E-R si scrive "*disjoint*" vicino al triangolo ISA).
- **Overlapping** (sovrapposizione): un'entità può appartenere a più di un entity set di livello inferiore.

Vincoli di completezza: vincoli su quali entità in un entity set di livello superiore, all'interno di una generalizzazione, debbano appartenere ad almeno un entity set a livello inferiore e quali no:

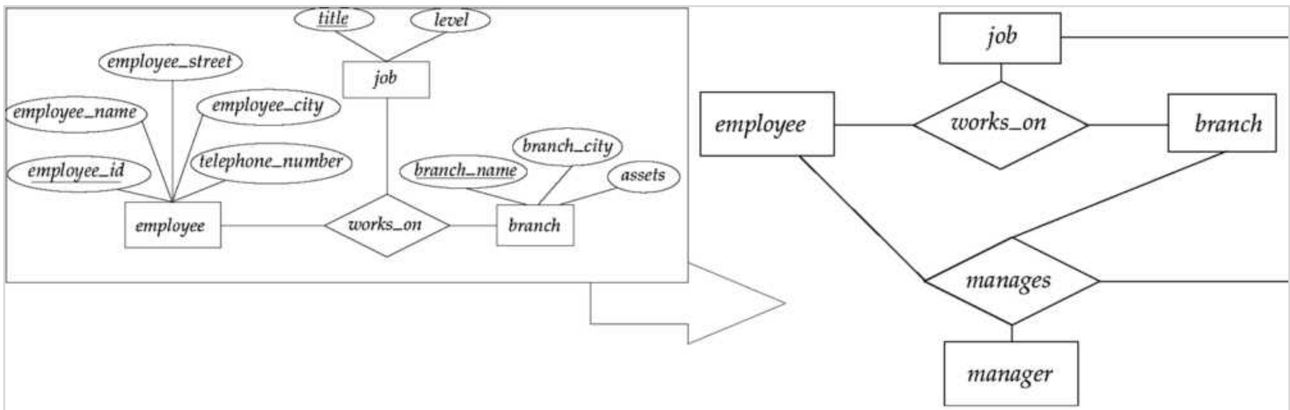
- **Completezza totale**: un'entità deve appartenere ad uno degli entity set di livello inferiore

[2] La **relazione ISA** è spesso detta relazione superclasse-sottoclasse (*superclass-subclass relationship*)

- **Completezza parziale**: un'entità non deve necessariamente appartenere ad uno degli entity set di livello inferiore

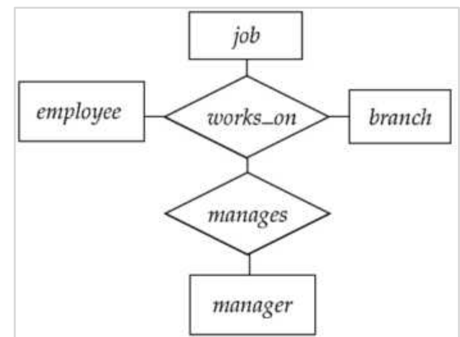
7 AGGREGAZIONE

Consideriamo la relazione ternaria *works_on* già vista. Supponiamo che si vogliano registrare i *manager* per i lavori eseguiti da un *employee* in un *branch*:



works_on e *manages* rappresentano dell'informazione sovrapposta (*overlapping*). Infatti ogni relazione di *manages* corrisponde a una di *works_on*. Comunque, qualche relazione di *works_on* potrebbe non corrispondere a nessuna relazione di *manages* → non si scarta *works_on*! → eliminiamo la ridondanza con l'**aggregazione**:

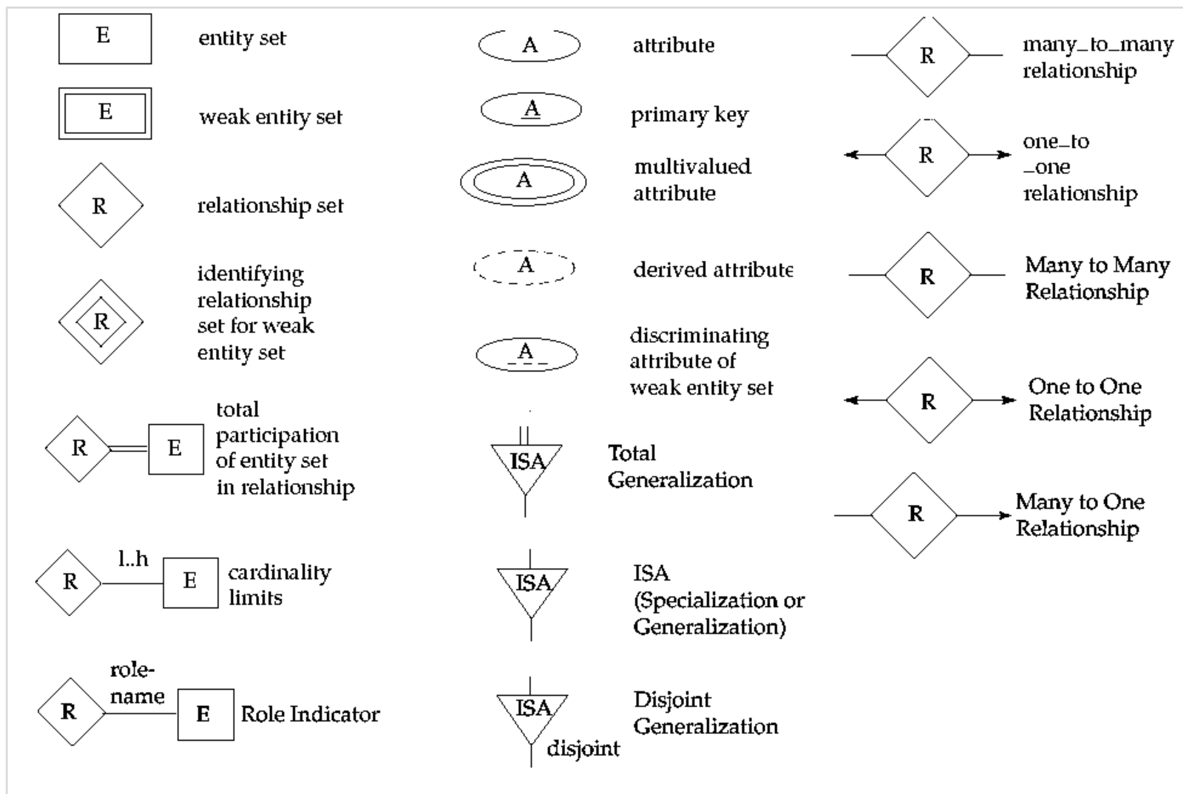
- Trattiamo la relazione come un'entità astratta, permettiamo perciò relazioni tra relazioni.
- Senza avere ridondanza quindi, il nuovo diagramma rappresenta con immediatezza il fatto che ogni *employee* lavori su (*works_on*) un particolare *job* in un determinato *branch*.
- Vediamo quindi come una combinazione (*employee, branch, job*), ovvero una n-upla, e perciò la relazione *works_on* possa essere associata mediante la relazione *manages* ad una entità di *manager*.



8 PROBLEMI E DUBBI COMUNI (2)

- **Uso di attributo o di entity set** per rappresentare un oggetto.
- Se un concetto del mondo reale sia meglio espresso da **entity set o relationship set**.
- **Uso di entity set "forti" o deboli**.
- **Uso di specializzazione/generalizzazione** per contribuire rendere la progettazione modulare
- **Uso di aggregazione** per trattare entity set come singole unità, senza considerare i dettagli della loro struttura interna.

9 SIMBOLI DEI DIAGRAMMI E-R



10 RIDUZIONE ALLO MODELLO RELAZIONALE

Le *primary key* permettono agli entity sets e relationship sets di essere espressi uniformemente come un **modello relazionale** che rappresenti il contenuto del DB. Un DB rappresentabile con un diagramma E-R infatti può anche essere rappresentato come una <u>collezione di tabelle (*schemas*)</u>.

↳ per ogni entity set e relationship set esiste una tabella (*schema*) unica con il nome dell'entity set o relationship set corrispondente. Ogni *schema* ha un numero di colonne (generalmente <u>colonne = attributi</u> dell'entity set) che hanno un nome unico.

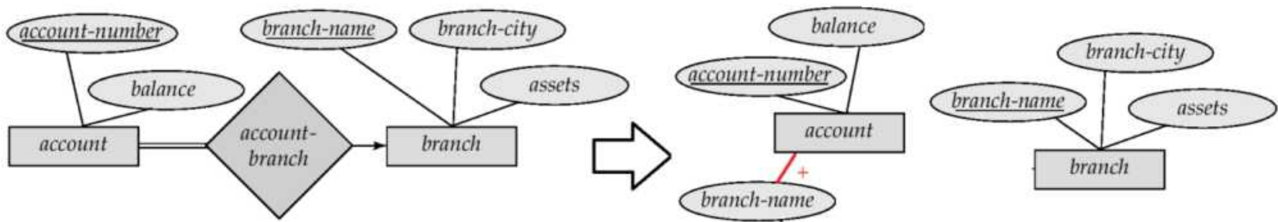
Un **entity set** "forte" è riducibile ad una tabella con i suoi stessi attributi.

Un **entity set debole** è riducibile ad una tabella con i suoi stessi attributi che include una colonna per la *primary key* dell'entity set "forte" che fa da identificatore (es: payment = (<u>loan_number</u>, <u>payment_number</u>, payment_date, payment_amount).

Un **relationship set molti-a-molti** è riducibile ad una tabella con attributi per le *primary keys* dei due entity set partecipanti alla relazione e ogni altro attributo descrittivo del relationship set (es: borrower = (<u>customer_id</u>, <u>loan_number</u>, access_date)

10.1 ELIMINAZIONE DELLE RIDONDANZE

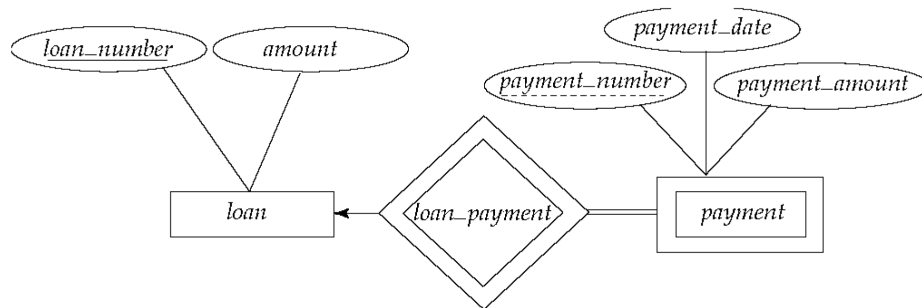
I **relationship set molti-a-uno** e **uno-a-molti** che sono **<u>totali</u>** dalla parte dei "molti" possono essere ridotti aggiungendo un attributo extra dal lato "molti" contenente la *primary key* del lato "uno" (es: invece di creare una tabella per il relationship set *account_branch*, aggiungere un attributo *branch_name* alla tabella preso dall'entity set *account* ⟶ vedi immagine successiva).



Per i **relationship set uno-a-uno** che sono **totali** è possibile scegliere uno dei due lati e considerarlo come un lato "molti". Così gli attributi extra possono essere aggiunti a ciascuna delle due tabelle corrispondenti ai due entity set.

Se la partecipazione è parziale dal lato "many", durante la riduzione della relazione alla una tabella, nella colonna dell'attributo extra compariranno dei valori *null* corrispondenti alle relazioni non partecipate.

La tabella del **relationship set indicatori** di entity set deboli (ovvero la relazione che collega l'entità debole al suo entity set "forte" identificatore) è ridondante (es: la tabella *payment* contiene già gli attributi che possono apparire nella tabella *loan_payment*, ovvero *loan_number* e *payment_number*, poichè per costruire le tabelle di entity set deboli, come abbiamo visto, dobbiamo aggiungere una colonna per la primary key esterna, in questo caso *loan_number*)



10.2 Attributi composti e multi-valore

Gli attributi composti sono "appiattiti" creando attributi singoli per ogni componente dell'attributo originale.

Gli attributi multivalore M di un'entity set E sono rappresentati da una tabella (*schema*) EM distinta:

- La tabella EM ha attributi corrispondenti alla *primary key* di E ed un attributo corrispondente all'attributo multi-valore M
  - es: l'attributo multi-valore *dependent_names* di *employee* è rappresentato dallo schema: *employee_dependent_names* = (*employee_id*, *dname*)
- Ogni valore di M è mappato in una tupla separata della tabella EM
  - es: se un entità *employee* con *primary key* "123-45-6789" ha due dipendenti Jack e Jane, il valore di M, ovvero "123-45-6789", è mappato da due tuple: (123-45-6789, Jack) e (123-45-6789 , Jane)

10.3 Riduzione delle specializzazioni

Metodo 1: "stile Topdown"

1. Fare una tabella dell'entity set di livello massimo
2. Fare una tabella per ogni entity set di livello inferiore, includendo *primary key* dell'entity set di livello superiore e gli attributi locali. Ad esempio:

schema	attributes
person	*name, street, city*
customer	*name, credit_rating*
employee	*name, salary*

3. Inconveniente: per ricavare informazioni su, ad esempio, *employee* bisogna accedere a due relazioni: una corrispondente alla tabella di livello inferiore ed una corrispondente alla tabella di livello superiore.

Metodo 2: "stile Bottomup"

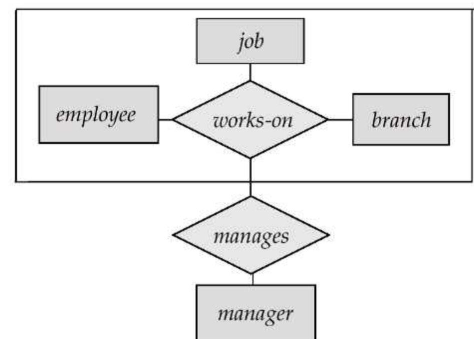
1. Fare una tabella per ogni entity con tutti gli attributi locali ed ereditati

schema	attributes
person	*name, street, city*
customer	*name, street, city, credit_rating*
employee	*name, street, city, salary*

2. Se la specializzazione è totale, lo schema per l'entity set generalizzato (in questo caso *person*) non richiede di essere memorizzato
   a. Può essere definite come una *view* relation contenente l'unione dei relatioship set specializzati
   b. Ma la tabella esplicita potrebbe essere comunque necessaria per i vincoli di chiave esterna (*foreign key constraints*)
3. Inconveniente: *street* e *city* potrebbero essere memorizzati ridondantemente per *people* che sono sia *customers* sia *employees*

10.4 Ridurre le aggregazioni

Per rappresentare un'aggregazione bisogna creare una tabella contenente la *primary key* del relationship set aggregato, la *primary key* dell'entity set associato e ogni attributo descrittivo.



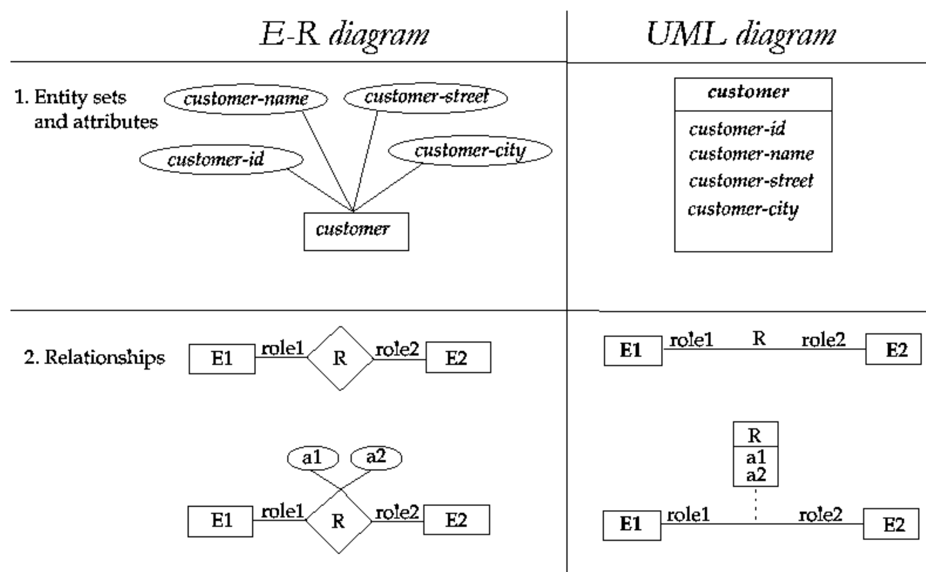
Es: per rappresentare l'aggregazione tra il relationship set *works_on* e l'entity set *manager*, si crea una tabella:

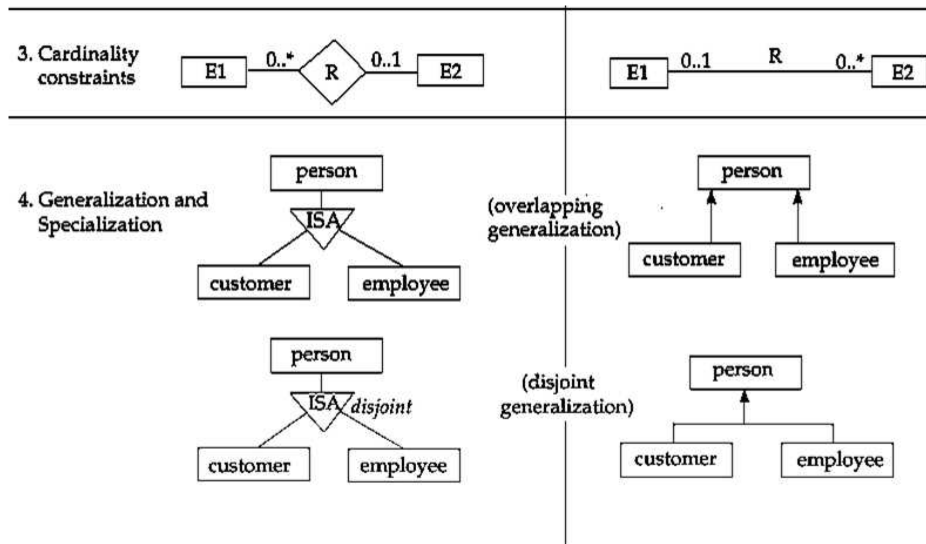
manages(employee_id, branch_name,title, manager_name)

In questo caso la tabella *works_on* è ridondante a condizione che si sia disposti a memorizzare valori *null* per l'attributo *manager_name* nella tabella di *manages*.

11 UML

UML sta per **Unified Modeling Language**. UML ha molti component per modellizzare graficamente I differenti aspetti di un sistema software. Gli **UML Class Diagrams** corrispondono ai Diagrammi E-R, ma con alcune differenze:





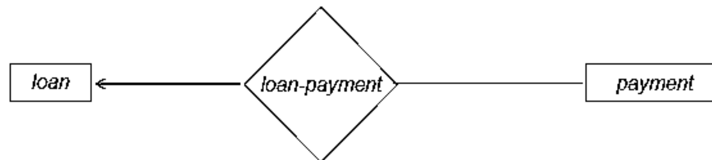
Le relazioni non binarie in UML sono però disegnate usando dei rombi, come nel diagramma E-R!

Posizione inversa dei vincoli di cardinalità tra UML e E-R!!!

12 Dipendenze di esistenza

Se l'esistenza dell'entita x dipende dall'esistenza dell'entità di y, allor x è detta di essere **dipendente dall'esistenza** (*existence dependent*) di y.

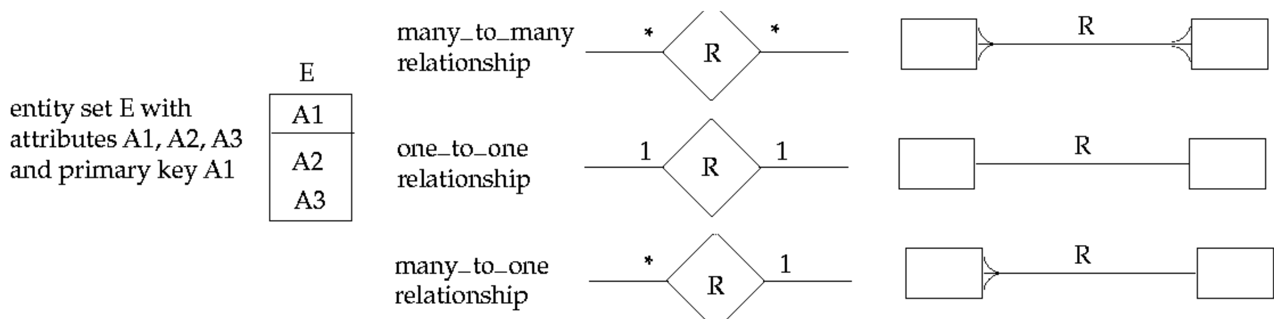
Ad esempio: y è l'entity set dominante (es: *loan*) ed x è l'entity set subordinato (es: *payment*):



Se *loan* è eliminato, allora tutte gli entity set associate (tra cui *payment*) devono anch'essi essere eliminati.

13 Notazione modello E-R alternativa

Esiste una notazione alternativa per indicare alcuni elementi del modello E-R:



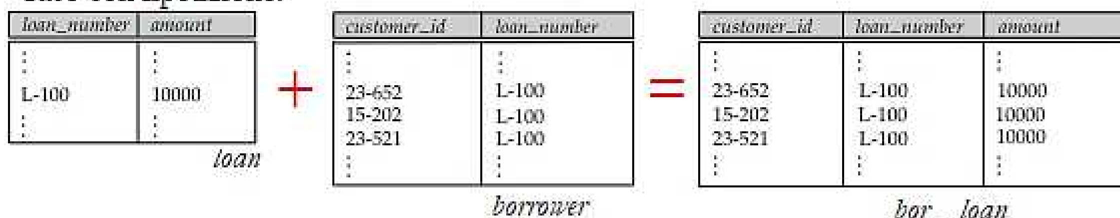
Cap 7: Progettazione database relazionali

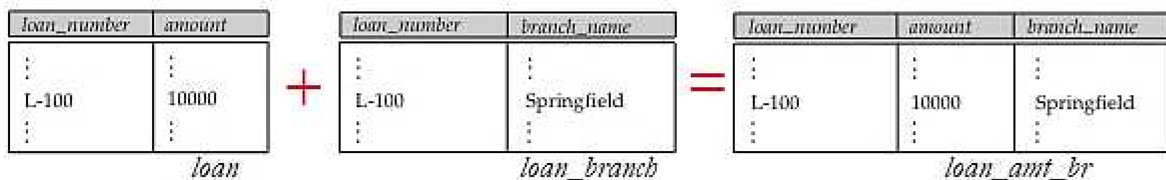
1 DECOMPOSIZIONE E PRIMA FORMA NORMALE

Prendiamo a esempio il DB di una banca mutuato dagli esempi precedenti del Cap. 6:

- branch = (branch_name, branch_city, assets)
- customer = (customer_id, customer_name, customer_street, customer_city)
- loan = (loan_number, amount)
- account = (account_number, balance)
- employee = (employee_id, employee_name, telephone_number, start_date)
- dependent_name = (employee_id, dname)
- account_branch = (account_number, branch_name)
- loan_branch = (loan_number, branch_name)
- borrower = (customer_id, loan_number)
- depositor = (customer_id, account_number)
- cust_banker = (customer_id, employee_id, type)
- works_for = (worker_employee_id, manager_employee_id)
- payment = (loan_number, payment_number, payment_date, payment_amount)
- savings_account = (account_number, interest_rate)
- checking_account = (account_number, overdraft_amount)

Supponiamo di voler combinare *borrower* e *loan* per avere *bor_loan* = (*customer_id, loan_number, amount*) ed anche *loan* e *loan_branch* per avere *loan_amt_br* = (*loan_number, amount, branch_name*). Potrebbe verificarsi una **ripetizione** di alcune infonazioni (ad esempio di "L-100" vedendo sotto)



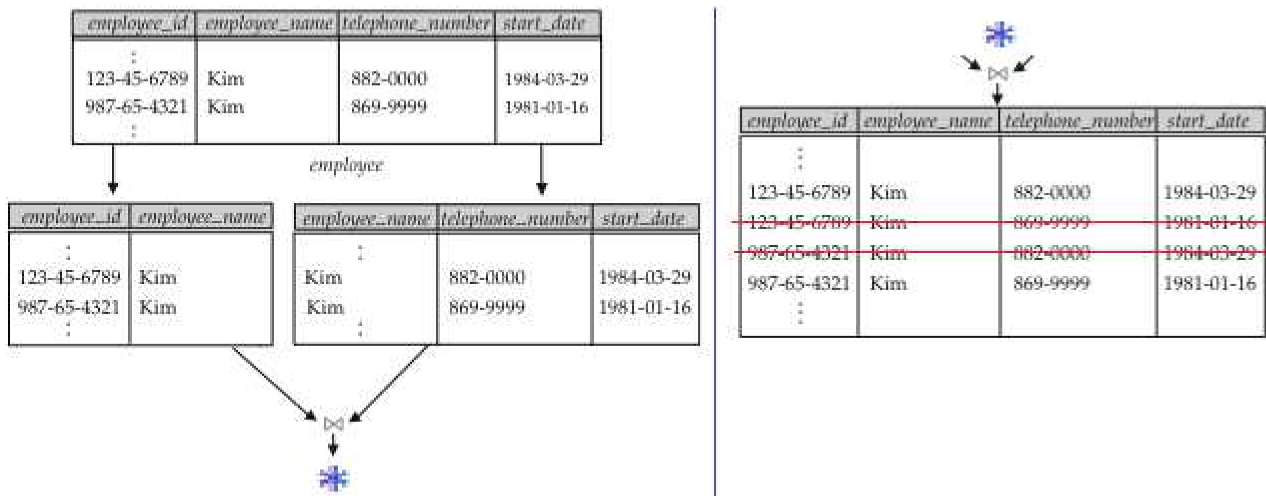


Se avessimo invece iniziato con la tabella *bor_loan*, come avremmo potuto sapere di poterla decomporre in *borrower* e *loan*? Possiamo identificare a tal proposito una **"regola":** se c'è una tabella (*loan_number, amount*), allora *loan_number* è una candidate key.

In maniera formale ciò si definisce **dipendenza funzionale** *loan_number* → *amount* e si indica appunto con la freccia (lo vedremo meglio il paragrafo dopo).

In *bor_loan*, abbiamo sempre un legame "semantico" tra *loan_number* e *amount*, ma *loan_number* può essere ripetuto ⇒ non è una candidate key! Ciò indica la necessita di decomporre *bor_loan*.

Non tutte le decomposizioni sono buone! Per esempio vediamo un caso in cui vi è perdita di informazione (si tratta di una *lossy decomposition*):



Abbiamo ottenuto due righe in più di informazione "sbagliata".

Un **dominio** è **atomico** se i suoi elementi sono considerati unità indivisibili (es. di domini non atomici: l'insieme dei nomi, gli attributi compositi, numeri identificativi come CS101 che possono essere divisi in parti, ...). In realtà però l'**atomicità** di un dominio dipende da come gli elementi del dominio sono usati! (es: una stringa di caratteri normalmente è considerata indivisibile) Avere domini atomici evita di avere informazioni codificate all'interno di essere che devono poi essere decodificate dall'applicazione che legge il database (usando domini atomici l'informazione è invece già esplicitamente memorizzata nel DB).

Uno *schema* relazionale R è in **prima forma normale (1NF)** se i domini di tutti i suoi attributi sono atomici.

↳ i valori non atomici complicano la memorizzazione ed aiutano a memorizzare dati ridondanti (es: insieme di *account* memorizzati per ogni *customer*, e insiemi di *owners* memorizzati per ogni *account* o viceversa)

Assumeremo che tutte le relazioni siano in prima forma normale (e torneremo su questo punto nel Cap. 9).

Decidiamo quindi che nel caso una relazione R sia non in 1NF è il caso di decomporla in un insieme di schemi relazionali $\{R_1, R_2, \dots, R_n\}$ tali che ogni schema R_i sia in 1NF (→ **normalizzazione** alla 1NF) e che la decomposizione sia *loseless-join* (non si perde informazioni se poi viene ricombinata con un *join*). Al fine di fare ciò sviluppiamo quindi una teoria basata sulle dipendenze funzionali e multi-valore.

2 DIPENDENZE FUNZIONALI

Le **dipendenze funzionali** sono **vincoli** sull'insieme di relazioni permesse, e richiedono che il valore di un certo insieme di attributi determini univocamente il valore per un altro insieme di attributi → è una generalizzazione del concetto di chiave.

Si ha dipendenza funzionale tra attributi quando il valore di uno o più attributi α determina univocamente il valore di un attributo β e si indica con $\boldsymbol{\alpha \to \beta}$. Diciamo quindi che α "determina funzionalmente" β, oppure che β "dipende funzionalmente da" α, oppure che α "è un determinante per" β.

Detto R uno *schema* di una relazione e presi due attributi $\alpha \subseteq R, \beta \subseteq R$, la dipendenza funzionale $\alpha \to \beta$ è **valida su $\boldsymbol{R}$** (*holds on* R) ⇔ ∀ relazione permessa $r(R)$ abbiamo:

$$t_1[\alpha] = t_2[\alpha] \;\Rightarrow\; t_1[\beta] = t_2[\beta] \quad \forall \text{ tupla } t_1, t_2$$ [1]

r(A,B)

	A	B
t1	1	4
t2	1	5
t3	3	7

In questo caso:

A→B non è su (A,B)=R
B→A è su/*holds on* (A,B)

[1] $t_1[\alpha]$ sta a significare il valore dell'attributo α nella tupla t_1.

K è una **chiave** (*superkey*) per lo schema relazionale $R \Leftrightarrow [\ K \rightarrow R\ ]$

K è una **chiave candidata** per lo schema relazionale $R \Leftrightarrow [\ K \rightarrow R \land$ è minimale $(\ \nexists \alpha \subset K \quad t.c. \quad \alpha \rightarrow R\ )\ ]$

Le dipendenze funzionali permettono di esprimere vincoli che non possono essere espressi con le chiavi (*superkey*). Consideriamo per esempio lo schema *bor_loan* = (*customer_id, loan_number,* *amount*). Ci aspettiamo una dipendenza di questo tipo *loan_number* ⟶ *amount*, ma non la seguente *amount* ⟶ *customer_id*.

Usiamo le dipendenze funzionali per:

- **Testare relazioni** vedendo se sono permesse sotto un dato insieme di dipendenze funzionali (se una relazione è permessa sotto un insieme F di dipendenze funzionali, diciamo che $\boldsymbol{r}$ **soddisfa** $\boldsymbol{F}$).
- **Specificare i vincoli su**ll'insieme delle **relazioni** permesse (diciamo che $\boldsymbol{F}$ **è valida su**/*holds on* **R** se tutte le relazioni permesse su R soddisfano l'insieme di dipendenze funzionali F).

N.B.: una specifica istanza nello schema di una relazione può soddisfare una dipendenza funzionale anche se la dipendenza funzionale non è valida su (*doesn't hold on*) tutte le istanze permesse (es: una specifica istanza di *loan* può, per caso, soddisfare la dipendenza funzionale *amount* ⟶ *customer_name*).

Una dipendenza funzionale è detta **banale** (*trivial*) se è soddisfatta da tutte le istanze di una relazione.

- (es: *customer_name, loan_number* ⟶ *customer_name* oppure *customer_name* ⟶ *customer_name*)
- in generale $\alpha \rightarrow \beta$ è banale se $\beta \subseteq \alpha$

2.1 CHIUSURA E BCNF

Chiusura $\boldsymbol{F^+}$ di $\boldsymbol{F}$ ($F =$ insieme di dipendenze funzionali): insieme di tutte le dipendenze funzionali logicamente implicate da F (es: $A \rightarrow B \land B \rightarrow C \Rightarrow A \rightarrow C$). Ovviamente $F^+ \subseteq F$.

Lo schema $\boldsymbol{R}$ di una relazione **è in BCNF** (*Boyce-Codd Normal Form*) **rispetto a F** ($F =$ insieme di dip. funz.) se $\forall$ "$\alpha \rightarrow \beta$" $\in F^+$ (con $\alpha, \beta \subseteq R$), è valida almeno una delle due seguenti affermazioni:

- $\alpha \rightarrow \beta$ è banale (cioè $\beta \subseteq \alpha$)
- α è chiave (*superkey*) per R

Uno schema non in BCNF è, ad esempio, *bor_loan* = (*customer_id, loan_number, amount*) perché *loan_number* ⟶ *amount* è valida su (*holds on*) *bor_loan*, ma *loan_number* non è una superkey.

Supponiamo di avere uno schema R e che una dipendenza non banale (*non-trivial*) $\alpha \rightarrow \beta$ causi una violazione della BCNF. Per risolvere ciò possiamo decomporre R in $(\alpha \cup \beta)$ ed $[R - (\alpha \cup \beta)]$

↳ in riferimento all'esempio di prima, possiamo risolverlo così: $(\alpha \cup \beta) =$ (*loan_number, amount*) ed $[R - (\alpha \cup \beta)] =$ (*customer_id, loan_number*)

2.2 BCNF, LA CONSERVAZIONE DELLE DIPENDENZE E LA 3NF

I vincoli, incluse le dipendenze funzionali, sono costantemente da controllare nella pratica a meno che non riguardino una sola relazione.

Lo schema $\boldsymbol{R}$ è **conservativo delle dipendenze** (*dependency preserving*) ⇔ per assicurare che tutte le dip. funz. siano valide su (*holds on*) R è sufficiente testare le dip. solo su ogni singola relazione della decomposizione.

Poiché non è sempre possibile avere sia BCNF che conservazione delle dipendenze, definiamo infine una forma normale "più debole", conosciuta come **terza forma normale** (**3NF**):

- uno schema relazionale R è in 3NF se $\forall \alpha \rightarrow \beta \in F^+$ vale almeno una delle tre seguenti affermazioni:
  - $\alpha \rightarrow \beta$ è banale (cioè $\beta \subseteq \alpha$)
  - α è chiave (*superkey*) per R
  - ogni attributo $A \in (\beta - \alpha)$ è contenuto in una *candidate key* di R (N.B.: ogni attributo può essere in una candidate key differente)

R in BCNF ⇒ R in 3NF (essendo le due condizioni BCNF le prime due di 3NF) ⟶ la terza condizione è un "rilassamento" della BCNF per assicurare anche la *dependency preservation* (vedremo dopo come mai).

2.3 Obiettivi della normalizzazione

Preso uno schema relazionale R con un insieme F di dipendenze funzionali, l'**obiettivo della normalizzazione** è stabilire se R sia in una forma "buona". Nel caso non fosse così, l'obiettivo è di decomporlo in un insieme di schemi relazionali $\{R_1, R_2, \dots, R_n\}$ tali che:

- Ogni schema relazionale R_i sia in forma "buona".
- La decomposizione sia una *lossless-join decomposition*.
- Preferibilmente, la decomposizione dovrebbe essere *dependency preserving*.

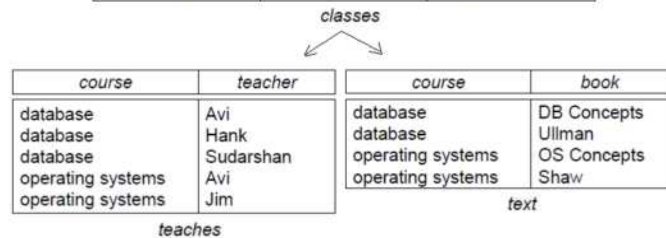
2.4 Forme normali ulteriori

Ci sono DB in BCNF che non sembrano essere sufficientemente normalizzati:

- ad esempio: *classes* (*course, teacher, book*) tale che la tupla rappresentante la singola *class* (*c, t, b*) significhi che t sia qualificato per insegnare c, e b sia un libro richiesto per c
- il DB si suppone elenchi per ogni corso l'insieme di *teachers* che possono potenzialmente essere gli insegnanti del corso, e l'insieme di *books* che possono potenzialmente servire al corso (non importa chi insegni veramente!)
- non ci sono dipendenze funzionali non banali (*non-trivial*) e perciò la relazione è in BCNF. Si manifestano però delle anomalie nell'inserimento, ad esempio, di una nuova insegnante: dovremmo inserire sia (*database, newteach_name, DB Concepts*) che (*database, newteach_name, Ullman*).
- perciò è meglio decomporre *classes* in due sotto-tabelle *teaches* e *text*.

course	teacher	book
database	Avi	DB Concepts
database	Avi	Ullman
database	Hank	DB Concepts
database	Hank	Ullman
database	Sudarshan	DB Concepts
database	Sudarshan	Ullman
operating systems	Avi	OS Concepts
operating systems	Avi	Stallings
operating systems	Pete	OS Concepts
operating systems	Pete	Stallings

classes



course	teacher
database	Avi
database	Hank
database	Sudarshan
operating systems	Avi
operating systems	Jim

teaches

course	book
database	DB Concepts
database	Ullman
operating systems	OS Concepts
operating systems	Shaw

text

L'esempio suggerisce che ci sia la necessità di **forme normali ulteriori** (in quel caso la 4NF), che però vedremo successivamente.

3 Teoria formale delle dipendenze funzionali

Consideriamo adesso la teoria formale delle dipendenze funzionali: essa ci dice che le dip. funz. sono implicate logicamente da un dato insieme di dip. funz. Sviluppiamo allora algoritmi per generare *lossless decompositions* in BCNF e 3NF per poi sviluppare algoritmi per testare se la decomposizione sia *dependency-preserving*.

3.1 Chiusura delle dipendenze funzionali

Possiamo trovare la chiusura F^+ di un insieme di dipendenze funzionali F mediante gli **Assiomi di Armstrong**:

- **Riflessività:** $\beta \subseteq \alpha \Rightarrow (\alpha \to \beta)$
- **Aumento:** $(\alpha \to \beta) \Rightarrow (\gamma\alpha \to \gamma\beta)$
- **Transitività:** $(\alpha \to \beta) \wedge (\beta \to \gamma) \Rightarrow (\alpha \to \gamma)$

Queste regole sono:

- Solide (*sound*): generano solo dip. funz. che realmente sono valide ("*that actually hold*")
- Complete: generano tutte dip. funz. che sono valide ("*that hold*")

Vediamo un primo algoritmo per calcolare la chiusura di F:

```
F+ = F
repeat
    for each functional dependency f in F+
        apply reflexivity and augmentation rules on f
        add the resulting functional dependencies to F+
    for each pair of functional dependencies f1 and f2 in F+
        if f1 and f2 can be combined using transitivity
            then add the resulting functional dependency to F+
until F+ does not change any further
```

Vedremo successivamente anche un metodo alternativo di computazione.

Possiamo ulteriormente semplificare il calcolo di F^+ con le seguenti regole, ricavabili dagli assiomi di Armstrong:

- **Unione:** $(\alpha \rightarrow \beta$ *holds* $\wedge\, \alpha \rightarrow \gamma$ *holds* $) \Rightarrow (\alpha \rightarrow \beta\gamma$ *holds* $)$
- **Decomposizione:** $(\alpha \rightarrow \beta\gamma$ *holds*$) \Rightarrow (\alpha \rightarrow \beta$ *holds* $\wedge\, \alpha \rightarrow \gamma$ *holds* $)$
- **Pseudo-transitività:** $(\alpha \rightarrow \beta$ *holds* $\wedge\, \gamma\beta \rightarrow \delta$ *holds* $) \Rightarrow (\gamma \rightarrow \delta$ *holds* $)$

3.2 CHIUSURA DEGLI ATTRIBUTI

Dato un insieme di attributi α definiamo $\boldsymbol{\alpha^+}$, ovvero **chiusura di $\boldsymbol{\alpha}$ sotto $\boldsymbol{F}$**, l'insieme di attributi che sono funzionalmente determinati da α sotto F. Vediamo l'algoritmo per calcolare α^+:

```
result := α;
while (changes to result) do
    for each β → γ in F do
      begin
        if β ⊆ result then result := result ∪ γ
      end
```

Ci sono diversi usi di questo algoritmo:

- Per **testare *superkeys***: per capire se α è una chiave di R, vediamo se α^+ contiene tutti gli attributi di R
- Per **testare le dipendenze funzionali**: per capire se $\alpha \rightarrow \beta$ *holds* (ovvero se è contenuta in F^+), basta controllare che $\beta \subseteq \alpha^+$ (è un test semplice e molto utile)
- Per **calcolare $\boldsymbol{F^+}$**: $\forall \gamma \subseteq R$, troviamo la chiusura γ^+ e $\forall S \subseteq \gamma^+$, restituiamo in output una dipendenza funzionale $\gamma \rightarrow S$.

3.3 COPERTURA CANONICA

Gli insiemi di dipendenze funzionali possono avere dipendenze ridondanti che possono essere derivate dalle altre (es: A→C in {A→B, B→C}) oppure parti di dipendenze ridondanti (es: {A→B, B→C, A→CD} oppure {A→B, B→C, AC→D} possono essere semplificati in {A→B, B→C, A→D})

Intuitivamente, una **copertura canonica di $\boldsymbol{F}$** sarà un insieme di dipendenze funzionali minimale equivalente ad F, ma che non ha dipendenze o parti di dipendenze ridondanti.

Definiamo ora un concetto che ci sarà utile. Preso un insieme F di dipendenze funzionali tra le quali $\alpha \rightarrow \beta$: [2]

- L'attr. **A è estraneo in $\boldsymbol{\alpha}$** se $A \in \alpha \;\wedge\; F$ implica logicamente $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ [3]
- L'attr. **A è estraneo in $\boldsymbol{\beta}$** se $A \in \beta \;\wedge\; F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ implica logicamente F

[2] **N.B.**: L'implicazione in direzione opposta è banale in ognuno dei casi seguenti, poiché una dipendenza funzionale "più forte" implica sempre quella più debole

[3] L'insieme F' è F, ma con la dipendenza funzionale $\alpha \rightarrow \beta$ privata di A! Ad esempio se $F = \{\dots, BV \rightarrow CD\}$ se $D \in \beta$ è estraneo a β allora è possibile eliminarlo da $BV \rightarrow CD$, perciò $F' = \{\dots, BV \rightarrow C\}$

Ovvero definiamo estraneo quell'attributo A in una dipendenza funzionale $\alpha \to \beta$ che può essere eliminato senza modificare la chiusura!

Vediamo due esempi:

- Dato $F = \{A \to C, AB \to C\}$ vediamo che B è estraneo in $AB \to C$ poiché $\{A \to C, AB \to C\}$ implica logicamente $A \to C$ (ovvero il risultato dell'eliminazione di B da $AB \to C$)
- Dato $F = \{A \to C, AB \to CD\}$ vediamo che C è estraneo in $AB \to CD$ poiché $AB \to C$ può essere derivato anche dopo aver eliminato C, da $\{A \to C, AB \to D\}$

Consideriamo un insieme di dipendenze funzionali F e la dipendenza funzionale $(\alpha \to \beta) \in F$:

- Per **testare** se l'attributo $\boldsymbol{A} \in \alpha$ sia **estraneo in $\boldsymbol{\alpha}$** si può calcolare $(\{\alpha\} - A)^+$ usando le dipendenze in F e poi controllare che $(\{\alpha\} - A)^+$ contenga β. Se è vero, allora A è estraneo in α[4].
- Per **testare** se l'attributo $\boldsymbol{A} \in \beta$ sia **estraneo in $\boldsymbol{\beta}$** si può calcolare α^+ usando le dipendenze in $F' = (F - \{\alpha \to \beta\}) \cup \{\alpha \to (\beta - A)\}$ e poi controllare che α^+ contenga A. Se è vero, allora A è estraneo in β.

Copertura canonica di $\boldsymbol{F}$: insieme di dipendenze funzionali F_C tale che:

- F implichi logicamente ogni dipendenza di F_C e viceversa
- Non esistano attributi estranei nelle dipendenze funzionali di F_C
- Ogni lato sinistro (es: α in $\alpha \to \beta$) sia unico

Per calcolare la copertura canonica di F vediamo il seguente algoritmo:

```
repeat
        Use the union rule to replace any dependencies in F
                α1 → β1 and α1 → β2 with α1 → β1 β2
        Find a functional dependency α → β with an
                extraneous attribute either in α or in β
        If an extraneous attribute is found, delete it from α → β
until F does not change
```

N.B.: La *Union rule* potrebbe essere applicabile solo dopo che alcuni attributi estranei siano stati eliminati, per questo motivo bisogna applicarla nuovamente ogni volta, ed è quindi messa all'interno del ciclo *repeat*.

3.4 LOSSLESS-JOIN DECOMPOSITION

Preso uno schema relazionale $R = (R_1, R_2)$, perché avvenga una *lossless-join decomposition* in R_1 ed R_2, bisogna che per ogni possibile relazione r nello schema R valga la condizione:

$$r = \prod_{R_1}(r) \bowtie \prod_{R_2}(r)$$

Una decomposizione di R in R_1 ed R_2 è *lossless-join* $\Leftrightarrow [(R_1 \cap R_2) \to R_1] \in F^+ \quad \vee \quad [(R_1 \cap R_2) \to R_2] \in F^+$

Esempio: $R = (A, B, C)$ e $F = \{A \to B, B \to C)$ può essere decomposto in due modi:

- $R_1 = (A, B),\ R_2 = (B, C)$
  - è *lossless-join* perché $R_1 \cap R_2 = \{B\}$ e $(B \to BC) \in F^+$
  - è *dependency preserving*.
- $R_1 = (A, B),\ R_2 = (A, C)$
  - è *lossless-join* perché $R_1 \cap R_2 = \{A\}$ e $(A \to AB) \in F^+$
  - **non** è *dependency preserving* perché per assicurare che $B \to C$ sia valida su (*holds on*) R **non** è sufficiente testare la dip. sulle singole R_1 e R_2 della decomposizione, ma bisogna calcolare $R_1 \bowtie R_2$.

---

[4] Questo poiché se A è estraneo ad α la chiusura di α rimane invariata anche se A è rimosso!

3.5 DEPENDENCY PRESERVING DECOMPOSITION

Chiamiato F_i ogni insieme di dipendenze $\subseteq F^+$ che include solo attributi di R_i:

- una decomp. di R in $R_1, R_2, \dots_R_n$ è ***dependency preserving*** se $(F_1 \cup F_2 \cup \dots \cup F_n)^+ = F^+$.

Se una decomposizione non è *dependency preserving*, per controllare che non ci siano stati errori nella decomposizione (ovvero che non sia stata *lossy*) bisognerà calcolare dei join, che è un'operazione computazionalmente costosa.

Vediamo un <u>metodo per verificare che la decomposizione di R in $R_1, R_2, \dots_R_n$ sia *dependancy preserving*:</u>

1. Per testare se una generica dipendenza $\alpha \rightarrow \beta$ sia preservata usiamo il seguente algoritmo (in esso la chiusura è calcolata rispetto a *F*):

```
result = α
while (changes to result) do
  for each Ri in the decomposition
        t = (result ∩ Ri)+ ∩ Ri
        result = result ∪ t
```

2. Dopo averlo svolto, se *result* contiene tutti gli attributi in β allora la dipendenza $\alpha \rightarrow \beta$ è preservata.
3. Se applichiamo i punti 1 e 2 a tutte le dipendenze di F possiamo verificare se è *dependency preserving* o no.
   ↳ questo metodo è vantaggioso perché il tempo richiesto cresce polinomialmente, mentre il tempo per calcolare in maniera "bruta" F^+ ed $(F_1 \cup F_2 \cup \dots \cup F_n)^+$ cresce esponenzialmente.

Esempio: Presa $R = (A, B, C)$ e $F = \{A \rightarrow B, B \rightarrow C\}$ con chiave $\{A\}$, vediamo che R non è in BCNF. Bisognerà allora decomporla. Decomponiamo quindi in $R_1 = (A, B)$ e $R_2 = (B, C)$ che sono in BCNF e vediamo immediatamente che la decomposizione è *lossless-join* e *dependency preserving*.

3.6 BOYCE-CODD NORMAL FORM

$\alpha \rightarrow \beta$ è non banale (*non-trivial*) (e quindi causa una **violazione della BCNF**) se α^+ non include tutti gli attributi per R, ovvero se non è una *superkey* di R.

Vediamo un **metodo semplificato** per il cercare la presenza di eventuali violazioni:

- è sufficiente controllare solo le dipendenze nell'insieme F per una violazione della BCNF, piuttosto che verificare tutte le dipendenze in F^+.
- se nessuna delle dipendenze in F causa la violazione della BCNF, allora nessuna delle dipendenze in F^+ causa la violazione della BCNF

Comunque, usare solo F **non è un metodo corretto** quando si testa una relazione in una decomposizione di R:

- Prendiamo ad esempio $R = (A, B, C, D, E)$, con $F = \{ A \rightarrow B, BC \rightarrow D\}$ e decomponiamo R in $R_1 = (A, B)$ e $R_2 = (A, C, D, E)$
- Nessuna delle dipendenze funzionali in F contiene solamente attributi da (A, C, D, E), allora potremmo essere indotti a pensare che R_2 soddisfi BCNF. <u>**Ma**</u> invece la dipendenza $AC \rightarrow D$ in F^+ mostra che R_2 non è in BCNF.

Per verificare se una relazione $\boldsymbol{R_i}$ **di una decomposizione** di R sia **in BCNF** possiamo:

- Testare R_i per la BCNF rispetto ad una <u>restrizione</u> di F a R_i (ovvero tutte le dipendenze funzionali in F^+ che contengono solo attributi da R_i)
- Usare l'insieme di dipendenze originali F che è valida su (*holds on*) R e, per ogni insieme di attributi $\alpha \in R_i$ verificare che α^+ o non includa attributi di $R_i - \alpha$, oppure che includa tutti gli attributi di R_i.

↳ infatti se la condizione è violata da qualche dipendenza $(\alpha \rightarrow \beta) \in F$, si può dimostrare che la dipendenza $\alpha \rightarrow (\alpha^{+} - \alpha) \cap R_i$ è valida su (*holds on*) R_i, ed allora R_i viola la BCNF. Useremo quindi $\alpha \rightarrow \beta$ per decomporre R_i ↓

Vediamo ora l'algoritmo per la **normalizzazione alla BCNF**, ovvero decomporre R in R_i tutti in BCNF, con decomposizioni *lossless-join:*

```
result := {R };
done := false;
compute F +;
while (not done) do
   if (there is a schema Ri in result  that is not in BCNF)
      then begin
            let α → β be a nontrivial functional dependency that holds on Ri
                        such that α → Ri is not in F +,
                        and α ∩ β = ∅;
             result := (result – Ri ) ∪ (Ri – β) ∪ (α, β );
            end
      else done := true;
```

Ad esempio: data $R = (A, B, C)$ e $F = \{A \rightarrow B, B \rightarrow C\}$ con chiave $\{A\}$, non è in BCNF ($B \rightarrow C$ ma B non è *superkey*). La decomposizione è $R_1 = (B, C)$ e $R_2 = (A, B)$.

<u>Non è sempre possibile decomporre in BCNF e con decomposizione *dependency preserving*</u>.

Ad esempio: dati $R = (J, K, L)$ e $F = \{JK \rightarrow L, L \rightarrow K\}$ abbiamo due *candidate keys*: JK e JL. R non è in BCNF. Qualsiasi decomposizione di R però non può preservare $JK \rightarrow L$. Ciò vuole dire che per vedere se la decomposizione è *lossless-join* dovremo utilizzare un ⋈ (*join*).

3.7 TERZA FORMA NORMALE (3NF)

Abbiamo quindi che la BCNF non è *dependency preserving*, ma sappiamo che la una verifica efficace delle decomposizioni *lossless* è importante (ovvero è meglio non avere ⋈ da calcolare). La soluzione è quindi definire la già vista **Terza Forma Normale (3NF)**, ovvero una forma normale più "debole".

Uno schema relazionale R è in **3NF** se $\forall \alpha \rightarrow \beta \in F^{+}$vale almeno una delle tre seguenti affermazioni:

- $\alpha \rightarrow \beta$ è banale (cioè $\beta \subseteq \alpha$)
- α è chiave (*superkey*) per R
- ogni attributo $A \in (\beta - \alpha)$ è contenuto in una *candidate key* di R (<u>N.B.</u>: ogni attributo può essere in una *candidate key* differente)

È più "debole" poiché ammette qualche ridondanza (con alcuni problemi che vedremo dopo). Ma le dipendenze funzionali possono essere testate sulle single relazioni decomposte senza calcolare dei ⋈

<u>⇒ **esiste sempre** una decomposizione *lossless-join*, *dependency-preserving* in 3NF!</u>

Ad esempio: preso il caso precedente $R = (J, K, L)$ e $F = \{JK \rightarrow L, L \rightarrow K\}$ abbiamo due *candidate keys*: JK e JL. R è in 3NF, poiché da $JK \rightarrow L$ vediamo che JK è una *superkey* e da $L \rightarrow K$ vediamo che K è contenuto in una *candidate key*.

<u>Esempio di problema di ridondanza in 3NF</u>

$R = (J, K, L)$

$F = \{JK \rightarrow L, L \rightarrow K\}$

J	L	K
j_1	l_1	k_1
j_2	l_1	k_1
j_3	l_1	k_1
null	l_2	k_2

abbiamo:
- ripetizione di informazione (es: l_1, k_1)
- il bisogno di usare *null* (es: per rappresentare la relazione l_2, k_2 dove non c'è nessun corrispondente valore di J

Test: abbiamo bisogno quindi di un metodo per controllare solo le dipendenze funzionali in F, senza dover controllare tutto F^+:

- Usiamo allora la chiusura degli attributi per controllare se, per ogni dipendenza $\alpha \rightarrow \beta$, α sia una *superkey*,
- Se α non è una *superkey*, abbiamo da verificare se ogni attributo in β è contenuto in una *candidate key* di R.

Questo tipo di test è piuttosto pesante in termini computazionali, poiché necessita la ricerca di *candidate keys* (il test per verificare la 3NF è stato dimostrato essere *NP-hard*, ovvero *Non-deterministic Polynomial-time hardness*). La decomposizione in 3NF descritta prima però può essere eseguita in un tempo polinomiale (⟶ vantaggiosa).

Vediamo l'**algoritmo** che implementa la **decomposizione** in **3NF**:

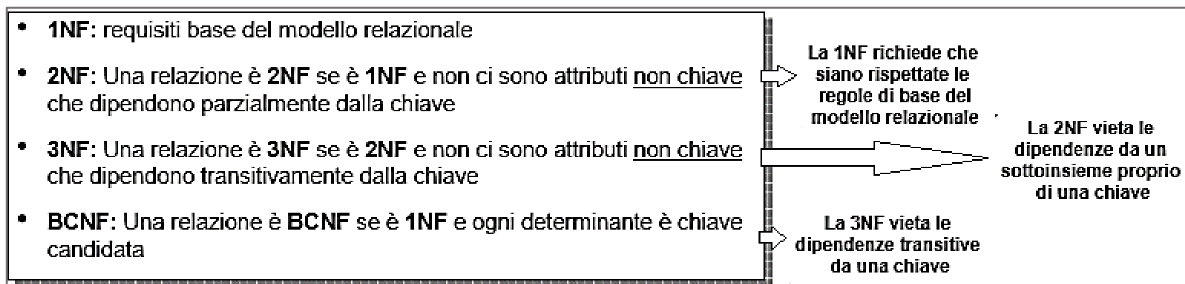
```
Let Fc be a canonical cover for F;
i := 0;
for each functional dependency α → β in Fc do
  if none of the schemas Rj, 1 ≤ j ≤ i contains α β
      then begin
            i := i + 1;
            Ri := α β
         end
if none of the schemas Rj, 1 ≤ j ≤ i contains a candidate key for R
  then begin
         i := i + 1;
         Ri := any candidate key for R;
      end
return (R1, R2, ..., Ri)
```

↳ questo algoritmo assicura che ogni schema relazionale R_i sia in 3NF, che la decomposizione sia *dependency preserving* e *lossless-join* (si può dimostrare).

È sempre possibile decomporre una relazione in un insieme di relazioni che sono in -

- 3NF ed abbiamo che:	- BCNF ed abbiamo che:
la decomposizione è *lossless* la decomposizione è *dependency preserving*	la decomposizione è *lossless* la decomposizione **può non essere** *dependency preserving*

Ricapitolando:



4 OBIETTIVI DELLA PROGETTAZIONE

Gli obiettivi da tenere a mente per costruire un buon DB relazionale devono essere:

- BCNF
- Decomposizioni *Lossless-join*
- *Dependency preservation*

Se non si riescono ad ottenere, si possono avere due compromessi:

- Mancanza di *Dependency preservation*
- Ridondanza per l'utilizzo della 3NF

SQL non fornisce un modo diretto per specificare le dipendenze funzionali diverso da quello delle *superkeys*. Non si possono quindi specificare le dip. funz. con delle *assertions* nel codice, ma sono difficili da testare.

Anche se abbiamo una decomposizione *dependency preserving*, con SQL non siamo in grado di testare efficientemente una dipendenza funzionale la cui parte sinistra (α in $\alpha \rightarrow \beta$) non sia una *superkey*.

5 DIPENDENZE MULTI-VALORE (MVDS)

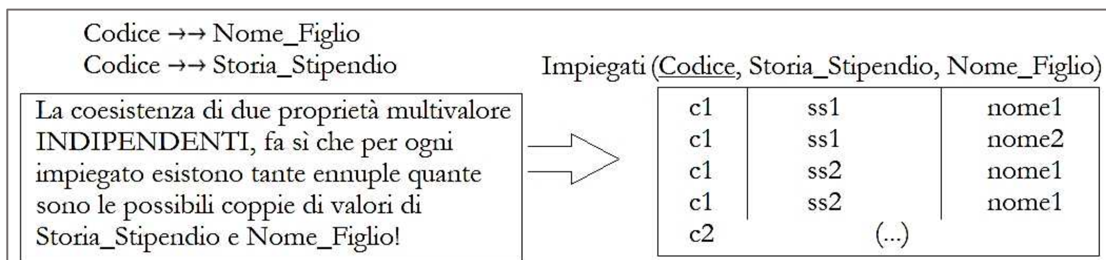
> Remind: si ha dipendenza funzionale tra attributi quando il valore di uno o più attributi α determina univocamente il valore di un attributo β (**identifica un solo valore! Le MVD ne identificheranno più di uno)** e si indica con $\alpha \rightarrow \beta$. Diciamo quindi che α determina funzionalmente β, oppure che β dipende funzionalmente da α, oppure che α è un determinante per β.

Vediamone la definizione formale: preso R uno schema relazionale e $\alpha \subseteq R,\ \beta \subseteq R$ diciamo che la **dipendenza multi-valore $\alpha \rightarrow\rightarrow \beta$ è valida su** (*holds on*) $\boldsymbol{R}$ se in ogni relazione permessa $r(R)$, per ogni coppia di tuple t_1, t_2 in r tale che $t_1[\alpha] = t_2[\alpha]$ esiste una coppia di tuple t_3, t_4 tale che:

- $t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_3[\alpha]$
- $t_3[\beta] = t_1[\beta]$
- $t_3[R-\beta] = t_2[R-\beta]$
- $t_4[\beta] = t_2[\beta]$
- $t_4[R-\beta] = t_1[R-\beta]$

	α	β	$R-\alpha-\beta$
t_1	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
t_2	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
t_3	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
t_4	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

Situazioni di questo tipo si hanno quando un attributo è determinante di un altro che può avere molti valori (es: una persona → i suoi fratelli, uno studente → le sue abilità, un cibo → i vini adatti a quel cibo…), ad esempio:



Prendiamo l'esempio di un po' di paragrafi precedente:

course	*teacher*	*book*
database	Avi	DB Concepts
database	Avi	Ullman
database	Hank	DB Concepts
database	Hank	Ullman
database	Sudarshan	DB Concepts
database	Sudarshan	Ullman
operating systems	Avi	OS Concepts
operating systems	Avi	Stallings
operating systems	Pete	OS Concepts
operating systems	Pete	Stallings

classes

↳ vediamo che $course \rightarrow\rightarrow teacher$, e $course \rightarrow\rightarrow book$. Questa formalizzazione ci serve per formalizzare il concetto già visto che "un particolare valore di A (che in questo caso è $course$) sia associato ad un insieme di valori di B (in questo caso $teacher$) e ad un insieme di valori C (in questo caso $book$), e che questi due insiemi siano in un certo senso indipendenti l'uno dall'altro".

Usiamo le MVD in due modi:

1. Per **testare** le **relazioni** e determinare quando esse siano **permesse** sotto un dato insieme di dipendenze funzionali e multi-valore.
2. Per **specificare i vincoli** su un insieme di relazioni permesse. Dobbiamo quindi occuparci solo delle relazioni che soddisfano un dato insieme di dipendenze funzionali e multi-valore.

Se una relazione r non soddisfa una MVD data, possiamo costruire una relazione r' che soddisfi la MVD aggiungendo tuple a r.

Dalla definizione di MVD possiamo derivare la regola: $\boxed{(\alpha \rightarrow \beta) \Rightarrow (\alpha \rightarrow\rightarrow \beta)}$ (→ ogni dip. funz. è una MVD).

La **chiusura D^+** di D è l'insieme di tutte le dipendenze funzionali e MVDs logicamente implicate da D. Possiamo calcolare D^+ da D utilizzando la definizione formale di dip. funz. e di MVD, ma ciò è realizzabile solo per MVDs davvero semplici. Per dipendenze complesse è meglio usare un *sistema di regole di inferenza* (non le vedremo).

5.1 QUARTA FORMA NORMALE (4NF)

Uno schema relazionale R è in **4NF** rispetto ad un insieme D di dip. funz. e MVD se $\forall(\alpha \rightarrow\rightarrow \beta) \in D^+$, con $\alpha \subseteq R, \beta \subseteq R$, almeno una delle seguenti affermazioni è verificata:

- $\alpha \rightarrow\rightarrow \beta$ è banale (cioè $\beta \subseteq \alpha$ oppure $\alpha \cup \beta = R$)
- α è *superkey* per lo schema R

Se una R è in 4NF, allora è anche in BNCF!

Definiamo, poiché ci servirà, la **D_i ≔ restrizione di D ad R_i** come l'insieme D_i che contiene:

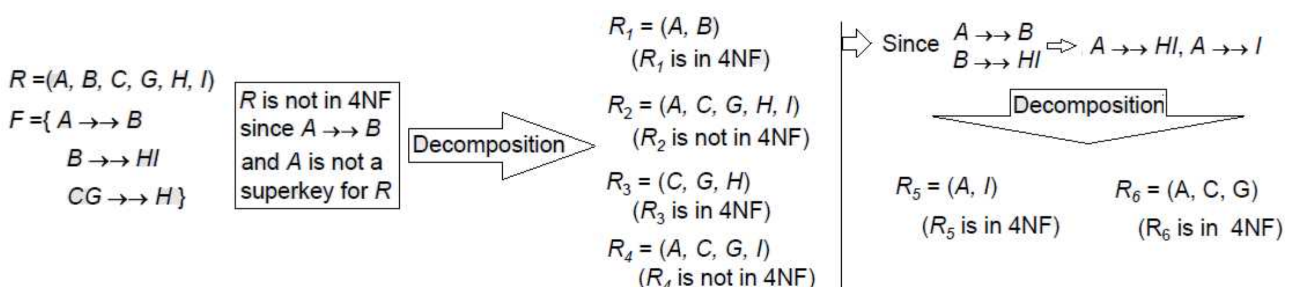
- Tutte le dipendenze funzionali in D^+ che includono solo attributi di R_i
- Tutte le MVDs nella forma $\alpha \rightarrow\rightarrow (\beta \cap R_i)$, con $\alpha \subseteq R_i$ e $(\alpha \rightarrow\rightarrow \beta) \in D^+$

Vediamo quindi adesso l'algoritmo per la decomposizione in 4NF:

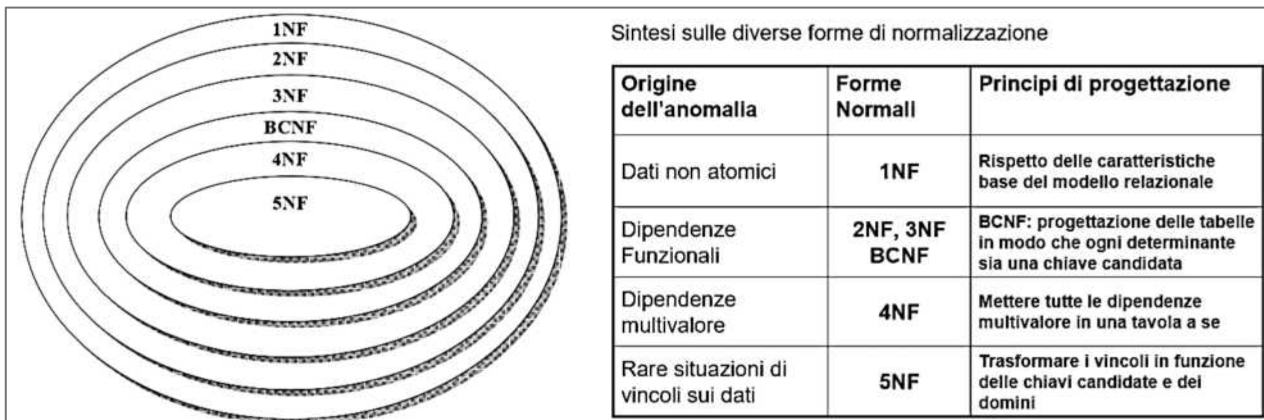
```
result: = {R};
done := false;
compute D+;
Let Di denote the restriction of D+ to Ri
while (not done)
  if (there is a schema Ri in result that is not in 4NF) then
    begin
      let α →→ β be a nontrivial multivalued dependency that holds
        on Ri such that α → Ri is not in Di, and α∩β=φ;
      result := (result - Ri) ∪ (Ri - β) ∪ (α, β);
    end
  else done:= true;
```

N.B.: ogni R_i è in 4NF, e la decomposizione è *lossless-join*.

Vediamo un esempio applicativo dell'algoritmo:



6 FORME NORMALI ULTERIORI



Sintesi sulle diverse forme di normalizzazione

Origine dell'anomalia	Forme Normali	Principi di progettazione
Dati non atomici	1NF	Rispetto delle caratteristiche base del modello relazionale
Dipendenze Funzionali	2NF, 3NF BCNF	BCNF: progettazione delle tabelle in modo che ogni determinante sia una chiave candidata
Dipendenze multivalore	4NF	Mettere tutte le dipendenze multivalore in una tavola a se
Rare situazioni di vincoli sui dati	5NF	Trasformare i vincoli in funzione delle chiavi candidate e dei domini

Le cosiddette **Join dependencies** generalizzano il concetto di MVDs e sono collegate alla **5NF**, detta anche **PJNF** (*Project-Join Normal Form*)

Una classe di vincoli ancora più generali è detta **DKNF** (D*omain-Key Normal Form*)

I problemi con questi vincoli così generalizzato sono la difficoltà computazionale che si portano dietro ed la mancanza di un insieme di regole di inferenza sicuro (→ sono raramente usate).

7 PROGETTAZIONE DI UN DB

Assumendo che un determinato schema R sia dato:

- R può essere stato generato convertendo un diagramma E-R in un insieme di tabelle
- R può essere una singola relazione contenente tutti gli attributi (chiamata ***universal relation***) → la normalizzazione rompe R in relazioni più piccole.
- R può essere il risultato di una qualche progettazione ad hoc di relazioni, che poi convertiremo in forma normale.

Modello E-R e Normalizzazione:

Quando un diagramma E-R è progettato con attenzione, identificando tutte le entità corrette, le tabelle generate dal diagramma E-R non necessitano di una successiva normalizzazione.

Comunque, nelle progettazioni reali possono comparire delle **dipendenze funzionali** da attributi di un'**entità** che non sono chiavi dell'entità ad altri attributi dell'entità (es: un'entità impiegato con attributi *department_number* e *department_address*, ed una dipendenza funzionale *department_number* → *department_address*. Una buona progettazione sarebbe rendere *department* un'entità)

Le **dipendenze funzionali** da attributi di una **relazione** che non sono chiavi della relazione sono possibili, ma rare la maggior parte delle relazioni sono binarie).

Denormalizzazione per Performance:

Può capitare di voler usare *schemas* non normalizzati per migliorare le performance, ad esempio per visualizzare *customer_name* insieme a *account_number* e *balance* richiede un join di *account* con *depositor*). Abbiamo due alternative

1. Usare relazioni non normalizzate contenenti attributi di *account* come di *depositor* aventi tutti gli attributi qui sopra richiesti (ricerca più veloce ma necessita di memoria extra e tempi di esecuzione extra per gli aggiornamenti, ed inoltre di lavoro di programmazione extra per il programmatore, con la possibilità di avere errori nel codice aggiuntivo).
2. Usare una *materialized view* definita come *account* ⋈ *depositor* (vantaggi e inconvenienti come sopra, eccetto il lavoro di programmazione extra).

Altri problemi di progettazione:

Alcuni aspetti della progettazione di DB non sono riassunti nella normalizzazione!

Alcuni esempi di DB scritti male:
- Usare *earnings_2004, earnings_2005, earnings_2006*, ecc… al posto che *earnings (company_id, year,amount)*, nello schema *(company_id, earnings)*. Sono infatti in BCNF, ma rendono le *query* che interrogano il DB su più anni molto piu dificili, ed inoltre ogni anno abbiamo bisogno di una tabella nuova!
- *company_year(company_id, earnings_2004, earnings_2005, earnings_2006)*. Anche questo in BCNF, ma rende anch'esso le *query* che interrogano il DB su più anni molto piu dificili, ed inoltre ogni anno abbiamo bisogno di uno attributo nuovo. E' inoltre un esempio di ***crosstab*** (tabella a campi incrociati), dove i valori per un attributo diventano i nomi delle colonne (usata nei fogli di calcolo e dagli strumenti di analisi dati).

8 MODELLAZIONE DI DATI TEMPORALI

I **dati temporali** hanno un intervallo di **tempo di associazione** (*association time interval*) durante il quale i dati sono validi.

Uno ***snapshot*** ("istantanea") è il valore dei dati in un determinato momento.

Esistono diverse proposte per estendere il modello E-R aggiungendo la validità temporale a:
- Attributi (es.: indirizzo di un cliente in diversi momenti)
- Entità (es.: periodo di tempo in cui esiste un account)
- Relazioni (es.: periodo di tempo per il quale un cliente ha posseduto un account)

(ma non esistono standard accettati).

Aggiungendo la componente temporale si ottengono dipendenze funzionali che non sono valide (*not to hold*) (es: *customer_id → customer_street, customer_city*), perché l'indirizzo alle quali si riferiscono varia nel tempo:

↳ una **dipendenza funzionale temporale** $X \to Y$ vale (*holds*) per lo schema R se la dipendenza funzionale $X \to Y$ è valida su gli *snapshot* per tutte le istanze legali $r(R)$

In pratica, i progettisti di database possono aggiungere **attributi *start time*** e ***end time*** alle relazioni (es: *course (course_id, course_title) → course (course_id, course_title, start, end)*. Vincolo: non esistono due tuple che possono avere sovrapposizioni di tempi validi → difficile da far rispettare in modo efficiente)

Le *references* alle *foreign key* possono essere alla versione corrente dei dati o ai dati in un determinato momento (es: *student transcript* dovrebbe fare riferimento alle informazioni sul corso al momento della frequentazione da parte dello studente)

Cap 8: Progettazione e sviluppo delle *applications*

1 INTERFACCE UTENTE E STRUMENTI: HTML

La maggior parte degli utenti di database non usa un linguaggio di query come SQL. Necessitiamo quindi di strumenti come *forms*, GUI (*graphical user interfaces*), sistemi per la segnalazione (*report generators*), strumenti di analisi dei dati ecc…

Molte interfacce sono basate sul Web. I *Back-end* [1] (Web servers) utilizzano tecnologie come Java Servlet, JSP (Java Server Pages) e ASP (Active Server Pages).

Il **Web** è un sistema informatico distribuito basato sull'ipertesto. La maggior parte dei documenti Web sono documenti ipertestuali formattati tramite **HTML** (HyperText Markup Language). I documenti HTML contengono:

- Un testo con le specifiche dei caratteri ed altre istruzioni di formattazione,
- Dei collegamenti ipertestuali ad altri documenti, che possono essere associati a sezioni del testo,
- *Forms* che consentono agli utenti di immettere dati che possono poi essere inviati al server Web.

Perché interfacciare i database al Web?

(1). I browser Web sono diventati l'interfaccia utente standard in fatto di database:
   a. Il Web permette ad un gran numero di utenti di accedere ai database da qualsiasi luogo
   b. Evita di scaricare/installare un programma dedicato, fornendo al tempo stesso una buona interfaccia grafica (esempi: prenotazioni di banche, compagnie aeree e noleggio auto, registrazione e valutazione dei corsi universitari, ecc…)

(2). Permettono la generazione dinamica di documenti:
   a. Esistono delle limitazioni nelle pagine HTML statiche (non è possibile personalizzare le pagine per i singoli utenti, è problematico aggiornare i documenti Web, specialmente se più documenti Web utilizzano gli stessi dati, e quindi li replicano)
   b. È possibile però risolvere questi problemi generando dinamicamente i documenti Web, basandosi sui dati memorizzati nel database (ad esempio è possibile adattare il display in base alle informazioni sull'utente memorizzate nel database, come annunci e previsioni meteo personalizzate, le ultime notizie locali ecc… o anche fare in modo che le informazioni visualizzate siano sempre aggiornate, a differenza delle pagine Web statiche, utile ad esempio per le informazioni sul mercato azionario ecc…)

1.1 UNIFORM RESOURCE LOCATORS (URLS)

Nel Web, la funzionalità dei puntatori viene implementata dagli **URLs** (ad es: http://www.bell-labs.com/topics/book/db-book) La prima parte di un URL indica come accedere al documento ("http" indica che è necessario accedere al documento utilizzando l'*Hyper Text Transfer Protocol).* La seconda parte fornisce il nome univoco di una macchina su Internet. Il resto dell'URL identifica il documento all'interno della macchina (→ identificatore locale).

L'identificatore locale può essere:

- Il nome del percorso di un file sulla macchina
- Un identificatore (nome del percorso) di un programma, più argomenti da passare al programma (ad es: http://www.google.com/search?q=silberschatz)

---

[1] I termini **front end** (FE) e **back end** (BE) denotano, rispettivamente, la parte visibile all'utente e con cui egli può interagire (interfaccia utente) e la parte che permette l'effettivo funzionamento di queste interazioni. Il FE, nella sua accezione più generale, è responsabile dell'acquisizione dei dati di ingresso e della loro elaborazione con modalità conformi a specifiche predefinite e invarianti, tali da renderli utilizzabili dal BE. Il collegamento del FE al BE è un caso particolare di interfaccia.

1.2 HTML: Gestione degli input

HTML fornisce formattazione, collegamenti ipertestuali e funzionalità per la visualizzazione delle immagini. Fornisce anche alcune **funzionalità di input** (ad esempio la seleziona da una serie di opzioni mediante *pop-up menus, radio buttons, check lists* o anche l'immissione di dati mediante caselle di testo). L'input viene restituito al server, ed è utilizzabile dagli eseguibili presenti su di esso

HTTP (HyperText Transfer Protocol) è il protocollo utilizzato per le comunicazioni con il server Web.

Sample HTML Source Text

```
<html> <body>
<table border cols = 3>
      <tr> <td> A-101 </td> <td> Downtown </td> <td> 500 </td> </tr>
      …
</table>
<center> The <i>account</i> relation </center>

<form action="BankQuery" method=get>
 Select account/loan and enter number <br>
 <select name="type">
    <option value="account" selected> Account
    <option> value="Loan">             Loan
 </select>
 <input type=text size=5 name="number">
 <input type=submit value="submit">
</form>
</body> </html>
```

Display of Sample HTML Source

A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900

The *account* relation

Select account/loan and enter number

Account | | submit

1.3 Scripting "lato client" e applets

I browser possono recuperare (*fetch*) determinati script[2] (***client-side scripts***) o programmi, assieme ai documenti, ed eseguirli in "modalità provvisoria" (*safe mode*) sul sito del cliente. Esempi sono Javascript, Macromedia Flash e Shockwave per animazioni/giochi, VRML e varii applets[3]

Gli scripts ed i programmi *client-side* consentono ai documenti di essere "attivi", dinamici, ad esempio mediante animazioni (eseguendo programmi sul sito locale), assicurando che i valori inseriti dagli utenti soddisfino alcuni vincoli di correttezza, permettendo l'interazione flessibile con l'utente (mediante l'esecuzione di programmi sul sito del cliente si accelera l'interazione, evitando molti *server roundtrip*[4]) ecc…

1.4 Scripting "lato client" e sicurezza

Servono dei meccanismi di sicurezza necessari per garantire che gli script dannosi non causino danni al computer del client. Sono facili da implementare per linguaggi di scripting a capacità limitata, è più difficile per linguaggi di programmazione generici come Java.

Per fare un esempio, il sistema di sicurezza Java garantisce che il codice applet Java non effettui chiamate di sistema direttamente → non permette azioni pericolose come le scritture di file e notifica all'utente le azioni potenzialmente pericolose consentendo di interrompere il programma o di continuarne l'esecuzione.

---

[2] Gli ***Script*** sono un insieme di strumenti/linguaggi per la programmazione, più facili da utilizzare rispetto ai tradizionali linguaggi. In questo caso si intende "porzioni di codice", ad esempio *scripts* scritte in JavaScript.

[3] ***Applet*** (*application+let*) indica un programma che viene eseguito come "ospite" nel contesto di un altro programma, detto per questo *container*, su un computer client (elaborazione lato client). In altre parole, un applet è un programma progettato per essere eseguito all'interno di un programma-container; ne consegue che l'applet non può essere eseguito indipendentemente da un altro programma.

[4] Un ***server roundtrip*** è ciò che avviene ogniqualvolta si inviano dati al server e lui risponde.

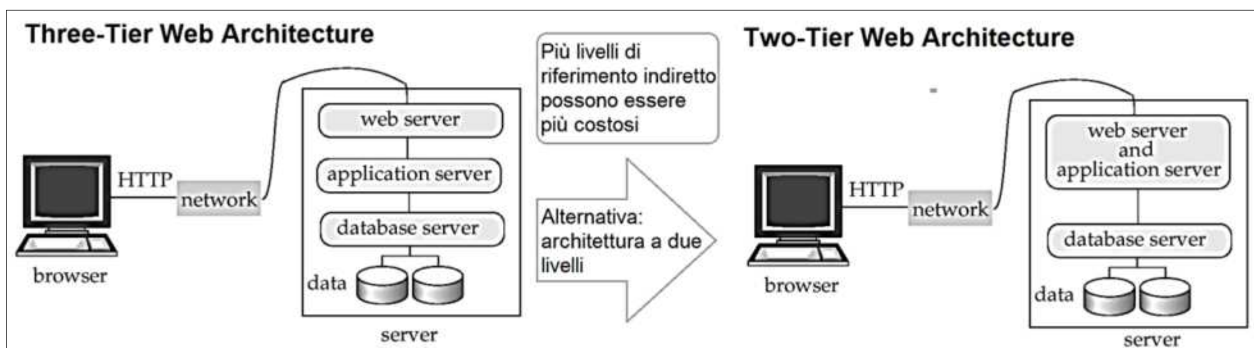
1.5 SERVER WEB

Un server Web può facilmente essere utilizzato come *front-end* per vari di servizi di informazione.

Il nome del documento in un URL può identificare un programma eseguibile che, una volta eseguito, genera un documento HTML. Quando un web server HTTP riceve una richiesta per tale documento, esegue il programma e restituisce il documento HTML che viene generato. Il client Web può passare argomenti aggiuntivi con il nome del documento.

Per installare un nuovo servizio sul Web, è sufficiente creare e installare un eseguibile che fornisca quel servizio. Il browser Web fornisce una GUI al servizio di informazioni.

Il Common Gateway Interface (**CGI**) è l'interfaccia standard tra *web server* e *application server*.



1.6 SESSIONI

Il protocollo HTTP è *connection-less*. Perciò, dopo che il server risponde a una richiesta, chiude la connessione con il client e dimentica tutto ciò che riguarda la richiesta. Al contrario, gli accessi Unix e le connessioni JDBC/ODBC rimangono connessi fino a quando il client non si disconnette (→ conservano l'autenticazione dell'utente e altre informazioni).

HTTP è *connection-less* per ridurre il carico sul server (→ i sistemi operativi hanno stretti limiti sul numero di possibli connessioni aperte su di una sola macchina). Ma i **servizi informativi** necessitano di informazioni sulla sessione (es: l'autenticazione dell'utente deve essere eseguita solo una volta per sessione) Come fare? Si possono utilizzare i ***cookie***[5].

---

[5] Un ***magic cookie***, o semplicemente ***cookie***, è un token digitale, ovvero un breve pacchetto di dati scambiato tra programmi in comunicazione fra loro, dai contenuti solitamente opachi, ovvero non significativi per il programma destinatario. Il dato è infatti tipicamente interpretato solo quando, in un secondo momento, il destinatario restituisce il cookie al mittente originario o ad un altro programma. Molto spesso il cookie è usato in maniera simile a un ticket, cioè un *magic number* generato da un server per un client e in grado di identificare quest'ultimo in maniera univoca. Il ticket, essendo difficilmente falsificabile, può essere utilizzato come prova di autenticazione o di autorizzazione.

I ***cookie HTTP*** (più comunemente denominati ***cookie web***, o per antonomasia *cookie*) sono un tipo particolare di *magic cookie*, una sorta di gettone identificativo, usato dai server web per poter riconoscere i browser durante comunicazioni con il protocollo HTTP usato per la navigazione web.

Tale riconoscimento permette di realizzare meccanismi di autenticazione, usati ad esempio per i login; di memorizzare dati utili alla sessione di navigazione, come le preferenze sull'aspetto grafico o linguistico del sito; di associare dati memorizzati dal server, ad esempio il contenuto del carrello di un negozio elettronico; di tracciare la navigazione dell'utente, ad esempio per fini statistici o pubblicitari.

Date le implicazioni per la riservatezza dei naviganti del web, l'uso dei cookie è categorizzato e disciplinato negli ordinamenti giuridici di numerosi paesi, tra cui quelli europei, inclusa l'Italia. La sicurezza di un cookie di autenticazione dipende generalmente dalla sicurezza del sito che lo emette, dal browser web dell'utente, e dipende dal fatto che il cookie sia criptato o meno. Le vulnerabilità di sicurezza possono permettere agli hacker di leggere i dati del cookie, che potrebbe essere usato per ottenere l'accesso ai dati degli utenti, o per ottenere l'accesso (con le credenziali dell'utente) al sito web a cui il cookie appartiene (*cross-site scripting* e *cross-site request forgery* ad esempio).

I cookie, e in particolare i cookie di terza parte, sono comunemente usati per memorizzare le ricerche di navigazione degli utenti; questi dati sensibili, possono essere una potenziale minaccia alla privacy degli utenti; proprio questo ha indotto le

Un ***cookie*** è una piccola porzione di testo che contiene informazioni identificative. Come funziona:

(1). È inviato dal server al browser alla prima interazione.
(2). È poi inviato dal browser al server che ha creato il cookie con ulteriori interazioni (⟵ protocollo HTTP)
(3). Il server salva le informazioni sui cookie rilasciati e può utilizzarli quando serve una richiesta

Ad esempio, può contenere informazioni sull'autenticazione e sulle preferenze dell'utente.

I cookie possono essere memorizzati in modo permanente o per un periodo di tempo limitato.

1.7 Servlet

La *Java Servlet specification* definisce un'API[6] per la comunicazione tra il server Web e l'*application program* (ad es.: i metodi per ottenere i valori di un parametro o per inviare testo HTML al client).

L'*application program* (chiamato anche ***servlet***) viene caricato nel server Web (→ modello a due livelli, *two-tier*, in cui ogni richiesta genera un nuovo *thread*[7] nel server Web, il quale *thread* viene chiuso una volta che la richiesta è esaudita)

L'API Servlet fornisce il metodo `getSession()` che:

- imposta un cookie durante la prima interazione con il browser e lo utilizza per identificare la sessione nelle interazioni successive
- fornisce metodi per memorizzare e ottenere le informazioni sulla sessione (es: nome utente, preferenze…)

```
Example Servlet Code:
Public class BankQuery(Servlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse result)
            throws ServletException, IOException {
            String type = request.getParameter("type");
            String number = request.getParameter("number");
              …code to find the loan amount/account balance …
              …using JDBC to communicate with the database..
              …we assume the value is stored in the variable balance
            result.setContentType("text/html");
            PrintWriter out = result.getWriter( );
            out.println("<HEAD><TITLE>Query Result</TITLE></HEAD>");
            out.println("<BODY>");
            out.println("Balance on " + type + number + "=" + balance);
            out.println("</BODY>");
            out.close ( );
    }
}
```

1.8 Scripting "lato server"

Lo *scripting* lato server semplifica l'attività di connessione di un database al Web. Vediamo i passi necessari:

(1). Definire un documento HTML contentente un *executable code* (codice eseguibile) oppure delle query SQL incorporati (*embedded*).

---

autorità europee e degli Stati Uniti a regolamentarne l'uso mediante una legge nel 2011. Infatti, la legislazione europea impone a tutti i siti degli stati membri, di informare gli utenti che il sito utilizza certe tipologie di cookie.

[6] Con ***application programming interface*** (in acronimo ***API***, in italiano "interfaccia di programmazione di una applicazione") si indica ogni insieme di procedure disponibili al programmatore, di solito raggruppate a formare un set di strumenti specifici per l'espletamento di un determinato compito all'interno di un certo programma. Spesso con tale termine si intendono le librerie software disponibili in un certo linguaggio di programmazione.

[7] Un ***thread*** o ***thread di esecuzione***, in informatica, è una suddivisione di un processo in due o più filoni o sottoprocessi che vengono eseguiti concorrentemente da un sistema di elaborazione monoprocessore (*multithreading*) o multiprocessore o multicore.

(2). I valori di input provenienti dai *forms* HTML possono essere utilizzati direttamente nel codice embedded o nelle query SQL embedded.
(3). Quando il documento viene richiesto, il server Web esegue il codice/la query SQL per generare il documento HTML effettivo.

Esistono numerosi linguaggi di scripting lato server (JSP, Javascript lato server, ColdFusion Markup (cfml), PHP, Jscript; ma anche linguaggi di scripting per scopi generali come VBScript, Perl, Python).

1.9 MIGLIORAMENTO DELLE PRESTAZIONI DEL SERVER WEB

Le performance sono un problema per i siti Web più popolari ed utilizzati. Per essi infatti deve essere possibile l'accesso da parte milioni di utenti ogni giorno, ed i web server devono saper risolvere migliaia di richieste al secondo nei momenti di picco.

Esistono perciò **tecniche di *caching***[8] utilizzate per ridurre i costi di pubblicazione delle pagine sfruttando le caratteristiche comuni tra le richieste:

(A). A lato server:
1. Caching delle connessioni JDBC tra le richieste servlet
2. Caching dei risultati delle query del database → i risultati della cache devono essere aggiornati se il database sottostante cambia
3. Caching del codice HTML generato

(B). A lato client:
1. Caching delle pagine tramite proxy Web

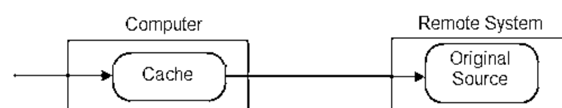
2 TRIGGERS

Un ***trigger*** è un'istruzione che viene eseguita automaticamente dal sistema come "effetto collaterale" di una modifica al database. Per progettare un meccanismo di *trigger* (innesco), dobbiamo:
- Specificare le condizioni alle quali deve essere eseguito il trigger.
- Specificare le azioni da intraprendere quando viene eseguito il trigger.

I trigger sono stati introdotti nello standard SQL in SQL 1999, ma erano supportati anche prima, utilizzando la sintassi non standard, dalla maggior parte dei database.

---

[8] Il ***caching*** è una tecnica usata per accelerare le ricerche di dati (perciò la lettura dei dati). Invece di leggere i dati direttamente dalla fonte, che potrebbe essere un database o un altro sistema remoto, i dati vengono letti direttamente da una cache sul computer che ha però bisogno i dati. Ecco un'illustrazione del principio di memorizzazione nella cache:



Una cache è un'area di memoria più vicina all'entità che ne ha bisogno rispetto alla sorgente originale. L'accesso a questa cache è in genere più veloce dell'accesso ai dati dalla sua fonte originale. Una cache viene in genere archiviata in memoria o su disco. Una memoria cache è in genere più veloce da leggere rispetto a una cache del disco, ma una memoria cache in genere non sopravvive al riavvio del sistema.

Vediamo un esempio:

supponiamo che invece di consentire saldi negativi dei conti, una banca tratti gli scoperti impostando il saldo del conto a zero, creando un prestito per l'importo dello scoperto e dia a questo prestito un numero di prestito identico al numero di conto del conto scoperto.

↳ La condizione per l'esecuzione del trigger è l'aggiornamento della relazione dell'account che determina un valore di bilancio negativo.

Trigger Example in SQL:1999

```
create trigger overdraft-trigger after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
        insert into borrower
           (select customer-name, account-number
            from depositor
            where nrow.account-number =
                      depositor.account-number);
        insert into loan values
           (n.row.account-number, nrow.branch-name,
                                        – nrow.balance);
        update account set balance = 0
         where account.account-number = nrow.account-number
end
```

2.1 Attivare eventi ed azioni in SQL mediante i trigger

- Il *trigger event* può essere inserito, eliminato o aggiornato (`insert, delete, update`) in SQL.
- I trigger sensibili agli aggiornamenti possono essere ristretti ad attributi specifici (es: `create trigger overdraft-trigger after update of balance on account`).
- I valori degli attributi prima e dopo un aggiornamento possono essere referenziati:
  - si utilizza `referencing old row as` per le eliminazioni e gli aggiornamenti
  - si utilizza `referencing new row as` per gli inserimenti e gli aggiornamenti
- I trigger possono essere attivati prima di un evento, che può fungere da vincolo extra. Ad esempio, se volessimo convertire gli spazi vuoti in *null* potremmo fare così:

```
create trigger setnull-trigger before update on r
referencing new row as nrow
for each row
   when nrow.phone-number = ' '
   set nrow.phone-number = null
```

2.2 Statement Level Triggers

Invece di eseguire un'azione separata per ogni riga interessata, è possibile eseguire un'unica azione per tutte le righe interessate da una transazione. Si può fare:

- Usando il `for each statement` al posto del `for each row`
- Usando il `referencing old table` oppure `referencing new table` per riferirsi a tabelle temporanee (chiamate "**tabelle di transizione**") contenenti le righe interessate

Tutto ciò può essere più efficiente quando si ha a che fare con istruzioni SQL che aggiornano un numero elevato di righe.

2.3 External World Actions

A volte richiediamo azioni "nel mondo esterno" attivate (*triggered*) dopi un aggiornamento del database (es: riordinare un oggetto la cui quantità in un magazzino è diventata piccola o far partire un segnale luminoso che segnali questa situazione)

I trigger non possono essere utilizzati per implementare direttamente azioni nel mondo esterno, ma possono essere utilizzati per registrare, in una tabella separata, le azioni da intraprendere → un processo esterno scansiona ripetutamente la tabella, esegue le azioni a livello di "mondo esterno" ed le elimina dalla tabella

Facciamo un esempio: supponiamo che un magazzino abbia le seguenti tabelle:

- *inventory (item, level)*: "quanto" di ogni articolo è nel magazzino
- *minlevel (item, level)*: qual'è il livello minimo desiderato per ciascun elemento
- *reorder (item, amount)*: quale quantità è da riordinare
- *orders (item, amount)*: ordini da piazzare (dato che viene letto dal processo esterno)

```
create trigger reorder-trigger after update of amount on inventory
referencing old row as orow, new row as nrow
for each row
        when nrow.level < = (select level
                                    from minlevel
                                    where minlevel.item = orow.item)
                and orow.level > (select level
                                        from minlevel
                                        where minlevel.item = orow.item)
  begin
            insert into orders
                (select item, amount
                 from reorder
                 where reorder.item = orow.item)
  end
```

Triggers in MS-SQLServer Syntax

```
create trigger overdraft-trigger on account
for update
as
if  inserted.balance < 0
begin
   insert into borrower
     (select customer-name,account-number
      from depositor, inserted
      where inserted.account-number =
                     depositor.account-number)
   insert into loan values
     (inserted.account-number, inserted.branch-name,
                   – inserted.balance)
   update account set balance = 0
     from account, inserted
     where account.account-number = inserted.account-number
end
```

2.4 QUANDO NON USARE I TRIGGER

Abbiamo visto l'uso dei trigger per compiti come:

- Mantenere i dati di riepilogo (***summary data***), come ad es. "lo stipendio totale di ciascun "dipartimento".
- Replicare database registrando le modifiche a speciali relazioni (chiamate ***change relations*** or ***delta relations***) con anche un processo separato che applica le modifiche alla replica.

Ci sono modi migliori per farlo ora:

- I database oggi forniscono *materialized view* per mantenere i *summary data*
- I database forniscono supporto per la replicazione (*change/delta relations*)

In molti casi è possibile utilizzare le funzioni di *encapsulation*, al posto dei trigger, per:

- Definire metodi di aggiornamento dei campi
- Eseguire azioni come parte dei metodi di aggiornamento anziché tramite un trigger

3 AUTORIZZAZIONI IN SQL[9]

3.1 TIPI DI AUTORIZZAZIONI

Tipi di autorizzazione su parti del database:

- Read authorization: consente la lettura, ma non la modifica dei dati.
- Insert authorization: consente l'inserimento di nuovi dati, ma non la modifica di dati esistenti.
- Update authorization: consente la modifica, ma non la cancellazione dei dati.
- Delete authorization: consente la cancellazione dei dati

Tipi di autorizzazione per modificare lo schema del database:

- Index authorization: consente la creazione e la cancellazione di indici.
- Resources authorization: consente la creazione di nuove relazioni.
- Alteration authorization: consente l'aggiunta o la cancellazione di attributi in una relazione.
- Drop authorization: consente la cancellazione delle relazioni

3.2 AUTORIZZAZIONI E VIEWS

Autorizzazioni e views:

- Gli utenti possono ricevere autorizzazioni sulle views senza ricevere alcuna autorizzazione sulle relazioni utilizzate nella definizione della view
- La capacità delle views di "nascondere" i dati serve sia a semplificare l'utilizzo del sistema sia a migliorare la sicurezza consentendo agli utenti di accedere solo ai dati di cui hanno bisogno per il loro lavoro
- Combinazioni di *relational-level security* e *view-level security* possono essere utilizzate per limitare l'accesso di un utente ai soli dati di cui ha bisogno.

↳ Vediamo un esempio con una *view:* supponiamo che un impiegato della banca abbia bisogno di conoscere i nomi dei clienti di ciascuna filiale, ma non sia autorizzato a visualizzare informazioni specifiche sul prestito →approccio: negare l'accesso diretto alla relazione *loan*, ma dare accesso alla view *cust-loan*, che consiste solo nei nomi dei clienti e delle filiali presso cui i hanno un prestito. La *cust-loan view* è definita in SQL così:

```
create view cust-loan as
  select branchname, customer-name
  from borrower, loan
  where borrower.loan-number = loan.loan-number
```

L'impiegato è autorizzato a vedere il risultato della query: `select * | from cust-loan`. Quando il *query processor* traduce il risultato in una query sulle relazioni del database, otteniamo una query su *borrower* e *loan*. Le autorizzazioni del commesso per la query devono però essere controllate prima che l'elaborazione di essa sostituisca la view con la definizione della view.

Autorizzazioni sulle views:

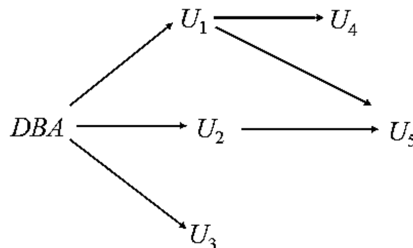
- La creazione della *view* non richiede la *resources authorization* poiché non viene creata alcuna relazione reale
- Il creatore di una *view* ottiene solo quei privilegi che non forniscono alcuna autorizzazione aggiuntiva oltre a quella che già aveva.
  ↳ ad esempio, se il creatore della wiev *cust-loan* avesse avuto solo la *read authorization* su *borrower* e *loan*, avrebbe ottenuto la *read authorization* solo su *cust-loan*.

[9] (vedi anche Sezione 4.3)

3.3 CONCESSIONE DEI PRIVILEGI

Il passaggio dell'autorizzazione da un utente a un altro può essere rappresentato da un ***authorization graph***. I nodi di questo grafico sono gli utenti. La radice del grafico è l'amministratore del database.

Consideriamo il grafico per l'*update authorization* in *loan*. La freccia $U_i \rightarrow U_j$ indica che l'utente U_i ha concesso (***granted***) l'*update authorization* in *loan* a U_j:



Requisiti del grafico: tutte le frecce di un *authorization graph* devono far parte di un percorso che ha origine nell'amministratore del database (DBA). Infatti:

- se DBA revoca la sovvenzione da U_1:
  - la concessione (*grant*) deve essere revocata da U_4 poiché U_1 non la ha più
  - non deve però essere revocata a U_5 poiché ha un altro percorso di autorizzazione tramite U_2
- non devono accadere autorizzazioni cicliche senza un percorso dalla radice:
  - DBA concede (*grant*) l'autorizzazione a U_7, U_7 concede l'autorizzazione a U_8, U_8 concede l'autorizzazione a U_7, DBA revoca la concessione a U_7, revocate le concessioni da U_7 a U_8 e da U_8 a U_7 poiché non vi è più alcun percorso da DBA a U_7 o U_8.

3.4 SPECIFICHE DI SICUREZZA IN SQL

Il **`grant`** *statement* è utilizzato per conferire l'autorizzazione:

```
grant <privilege list>
on <relation name or view name> to <user list>
```

`<user list>` può essere:

- un user-id
- la keyword `public`, che concede il privilegio a tutti gli utenti validi
- un ruolo (vedremo questo più tardi)

La concessione di un privilegio su una view non implica la concessione di alcun privilegio sulle relazioni sottostanti ad essa. Il concedente del privilegio deve già detenere il privilegio sull'elemento specificato (o essere l'amministratore del database), come abbiamo visto dai requisiti dell'*authorization graph*.

`<privilege list>` in SQL può contenere:

- `select`: consente il *read access* alla relazione o la possibilità di eseguire query utilizzando la view
  ↳ es.: per concedere agli utenti U_1, U_2 e U_3 la *select authorization* sulla relazione *branch*:
  `grant select on branch to` U_1, U_2, U_3
- `insert`: capacità di inserire tuple
- `update`: capacità di aggiornare le tuple utilizzando l'istruzione `update` di SQL
- `delete`: capacità di cancellare tuple
- `references`: capacità di dichiarare *foreign key* quando si creano relazioni
- `usage`: in SQL-92 autorizza un utente a utilizzare un dominio specificato
- `all privileges`: usato come forma abbreviata per concedere tutti i privilegi consentiti

Lo statement **`with grant option`** consente a un utente a cui viene concesso un privilegio di passare il privilegio ad altri utenti. Ad esempio, per concedere il `select` sul *branch* a U_1 con la `grant option`:

grant select on branch **to** U_1 **with grant option**

↳ concede ad U_1 i privilegi di `select` sul *branch* e consente a U_1 di concedere questo privilegio ad altri.

3.5 Ruoli

I ruoli consentono di specificare i privilegi comuni ad una classe di utenti una volta per tutte, creando un ***role*** corrispondente. I privilegi possono essere concessi o revocati dai ruoli, proprio come per l'utente. I ruoli possono essere assegnati agli utenti o anche ad altri ruoli (SQL 1999 supporta i ruoli).

Vediamo un esempio:

```
create role teller
create role manager

grant select on branch to teller
grant update (balance) on account to teller
grant all privileges on account to manager

grant teller to manager

grant teller to alice, bob
grant manager to avi
```

3.6 Revoca delle Autorizzazioni in SQL

Lo statement **`revoke`** viene utilizzato per revocare le autorizzazioni:

```
revoke <privilege list>
on <relation name or view name> from <revokee-list> [restrict|cascade]
```

Ad esempio: **`revoke select on`** `branch` **`from`** `U1, U2, U3` **`cascade`**

La revoca di un privilegio da un utente può causare anche la perdita di tale privilegio da parte di altri utenti. Questo fenomeno è detto ***cascading of the revoke***, Se vogliamo prevenire il *cascading* possiamo specificarlo mediante l'istruzione **`restrict`**:

```
revoke select on branch from U1, U2, U3 restrict
```

↳ se usiamo **`restrict`**, il comando **`revoke`** fallisce nel caso fossero necessari **`revoke`** a cascata!

Vediamo alcune caratteristiche:

- `<privilege list>` potrebbe essere `all to` per revocare tutti i privilegi che il revocatore può detenere.
- Se `<revokee-list>` include `public`, tutti gli utenti perdono il privilegio ad eccezione di quelli che lo hanno avuto in concessione esplicitamente.
- Se lo stesso privilegio è stato concesso due volte allo stesso utente da diversi beneficiari, l'utente può mantenere il privilegio dopo la revoca.
- Tutti i privilegi che dipendono dalle autorizzazioni concesse sono anch'essi revocati.

3.7 Limitazioni delle Autorizzazioni SQL

SQL non supporta l'autorizzazione a un livello di tuple (non possiamo ad esempio limitare gli studenti per vedere solo le tuple che memorizzano i loro voti).

Con l'aumento dell'accesso mediante Web ai database, gli accessi provengono principalmente dai server delle applicazioni. Perciò gli utenti finali non hanno un ID utente del database ognuno, sono tutti mappati sullo

stesso ID utente. Tutti gli utenti finali di un'applicazione (come un'applicazione web) possono perciò essere associati a un singolo utente del database

Il compito di autorizzazione nei casi appena mostrati è tutto da realizzarsi quindi nell'*application program*, senza supporto da parte di SQL:

- Benefici: le autorizzazioni a maggior dettagli, come quelle riguardanti le singole tuple, possono essere implementate dall'*application*.
- Inconvenienti: le autorizzazioni devono essere eseguite nel codice dell'*application* e possono quindi essere "disperse" in tutta l'*application* → la verifica l'assenza di lacune nell'autorizzazione diventa quindi molto difficile in quanto richiede la lettura di grandi quantità di codice dell'applicazione

4 Audit Trails

Un ***audit trail*** è un ***log*** (registro) di tutte le modifiche (inserimenti/eliminazioni/aggiornamenti) avvenuti nel database che contiene informazioni quali "l'utente che ha eseguito la modifica" e "quando è stata eseguita la modifica". E' utilizzato per tenere traccia di aggiornamenti errati/fraudolenti e può essere implementato utilizzando i trigger, ma molti sistemi di database forniscono supporto diretto per questo tipo di log.

5 Sicurezza nell'Application

5.1 Crittografia

I dati possono essere ***encrypted*** (crittografati) quando le *authorization provisions* (disposizioni di autorizzazione) del database non offrono una protezione sufficiente. Vediamo quindi alcune caratteristiche necessarie per una buona crittografia:

- Deve essere relativamente semplice per gli utenti autorizzati crittografare e de-crittografare i dati
- Lo schema di crittografia non deve dipendere dalla segretezza dell'algoritmo ma dalla segretezza di un parametro dell'algoritmo chiamato *encryption key* (chiave di crittografia)
- Deve essere estremamente difficile per un intruso determinare la *encryption key*

Data Encryption Standard **(DES)**: sostituisce i caratteri e riorganizza il loro ordine sulla base di una *encryption key* fornita agli utenti autorizzati tramite un meccanismo sicuro. La sicurezza del sistema non maggiore di quella del meccanismo di trasmissione della chiave, poiché la chiave deve essere condivisa.

Advanced Encryption Standard **(AES)**: è un nuovo standard che sostituisce DES e si basa sull'algoritmo di Rijndael, ma dipende anche dalle chiavi segrete condivise

Public-key encryption: ciascun utente possiede due chiavi:

- Una *public key*: chiave pubblica pubblicamente utilizzata per crittografare i dati, che non può essere utilizzata per de-crittografare i dati
- Una *private key*: chiave nota solo al singolo utente e utilizzata per de-crittografare i dati. Non è necessario che sia trasmessa al sito.

↳ lo schema di crittografia è tale che è impossibile o estremamente difficile de-crittografare i dati essendo forniti della sola chiave pubblica.

Lo schema di crittografia a chiave pubblica RSA si basa sulla difficolta computazionale della fattorizzazione di un numero molto elevato (100 di cifre) nei suoi fattori principali.

5.2 Autenticazione

L'autenticazione basata su password è ampiamente utilizzata, ma può essere identificata con uno ***sniffing*** della rete.

I **sistemi *challenge-response*** ("di risposta alla concorrenza") evitano per questo motivo la trasmissione di password. Come fare allora?

1. Il DB invia una stringa di verifica (generata a caso) all'utente.
2. L'utente crittografa la stringa ed invia il risultato
3. Il DB verifica l'identità decifrando il risultato.

- È possibile utilizzare il sistema di crittografia a chiave pubblica tramite DB: si invia un messaggio crittografato utilizzando la chiave pubblica dell'utente, e l'utente lo de-crittografa e lo re-invia nuovamente al mittente

Le **firme digitali** vengono utilizzate per verificare l'autenticità dei dati:

- Ad esempio: utilizzando la chiave privata (al contrario) per crittografare i dati, chiunque può verificare l'autenticità utilizzando la chiave pubblica (al contrario) per de-crittografare i dati. Solo il titolare di una chiave privata potrebbe aver creato i dati crittografati → in ciò consiste la firma.
- Le firme digitali aiutano anche a garantire la ***nonrepudiation***: il mittente non può in seguito affermare di non aver creato i dati.

5.3 CERTIFICATI DIGITALI

I certificati digitali sono utilizzati per verificare l'autenticità delle chiavi pubbliche.

Problema: quando comunichi con un sito web, come fai a sapere se stai parlando con il vero sito web o con un impostore? → Soluzione: utilizzare la chiave pubblica del sito web. Ma si presenta un altro problema: come verificare se la chiave pubblica è autentica? Solution:

- Ogni cliente (ad es. Browser) ha le chiavi pubbliche di alcune ***root-level certification authorities***
- Un sito può avere il suo nome/URL e la sua chiave pubblica firmata da un'autorità di certificazione: il documento firmato è chiamato **certificato**
- Il cliente può utilizzare la chiave pubblica dell'autorità di certificazione per verificare il certificato!!!

Possono esistere più livelli di autorità di certificazione. Ogni autorità di certificazione presenta il proprio *public-key certificate* firmato da un'autorità di livello superiore, e utilizza la propria chiave privata per firmare il certificato di altri siti Web/autorità

Cap 9.1: Object-Oriented Databases[1]

1 LA NECESSITÀ DI AVERE *DATA TYPES* COMPLESSI

DB applications tradizionali utilizzano **tipi di dati** concettualmente **semplici** nell'elaborazione dei dati → con relativamente pochi *data types*, vale la **1NF**

I tipi di dati complessi sono diventati più importanti negli ultimi anni:

- Ad es. gli indirizzi possono essere visualizzati come
  - Singola *string*
  - Attributi separati per ogni parte
  - Attributi compositi (***composite attributes***), che non sono in 1NF
- Ad es. è spesso conveniente memorizzare attributi multivalore (***multivalued attributes***) così com'è, senza creare una relazione separata per memorizzare i valori in primo luogo forma normale

Applicazioni possibili di tipi di dati complessi si trovano nell'impiego di computer per le progettazioni *computer-aided* (assistite da computer), nell'ingegneria del software *computer-aided* e nei *document/hypertext databases.*

2 L' OBJECT-ORIENTED DATA MODEL

2.1 CONFRONTO CON IL MODELLO E-R

In senso lato, un **oggetto** (*object*) corrisponde a un'entità nel modello E-R

Il paradigma *Object-Oriented* si basa sull'inclusione (***encapsulating***) del codice e dei dati relativi ad un oggetto in una singola unità.

L' *Object-Oriented Data Model* è un *Logical Data Model* (come il modello E-R), adattamento del paradigma di programmazione orientato agli oggetti (ad es. Smalltalk, C++) ai sistemi di database.

2.2 STRUTTURA DI UN OGGETTO

Ad ogni oggetto sono associati:

- Un insieme (*set*) di ***variables*** (variabili) che contengono i dati dell'oggetto. Il valore di ogni variabile è essa stessa un oggetto.
- Un insieme di ***messagges*** (messaggi) a cui risponde l'oggetto; ogni messaggio può avere 0, 1 o più *parameters* (parametri).
- Un insieme di ***methods*** (metodi), ognuno dei quali è un corpo di codice per implementare un messaggio; un metodo restituisce quindi un valore come *response* (risposta) al messaggio.

La rappresentazione fisica dei dati è visibile solo all'implementatore dell'oggetto.

Messaggi e risposte forniscono l'unica interfaccia esterna ad un oggetto. Il termine "messaggio" non implica necessariamente un messaggio che passa, trasmesso "fisicamente": i messaggi possono essere implementati come *procedure* ***invocations***.

[1] Il testo di riferimento dei capitoli 9.1 e 9.2 è il *Silberschatz, Korth, Sudarshan - Database Systems* **4th** Edition, mentre quello di tutti gli altri capitoli è il *Silberschatz, Korth, Sudarshan - Database Systems* **6th** Edition

2.3 MESSAGGI E METODI

I **metodi** sono programmi scritti in un linguaggio *general-purpose* con le seguenti caratteristiche:

- È possibile referenziare (*referencing*, fare riferimento) direttamente solo le variabili nell'oggetto stesso
- I dati in altri oggetti sono referenziati solo mediante l'invio di messaggi

I metodi possono essere ***read-only*** *methods* o ***update*** *methods* (i metodi "di sola lettura" non modificano, non "aggiornano", il valore dell'oggetto)

Attribute representation: a rigor di termini, ogni attributo di un'entità deve essere rappresentato da una variabile e da due metodi, uno per leggere e l'altro per aggiornare l'attributo. Ad esempio, l'attributo "*address*" è rappresentato da una variabile *address* e da due messaggi *get-address* e *set-address*.

Per comodità, molti *object-oriented data models* consentono l'accesso diretto alle variabili di altri oggetti.

2.4 CLASSI DI OGGETTI

Oggetti simili sono raggruppati in una **classe**; ogni tale oggetto è chiamato un'**istanza** della sua classe. Tutti gli oggetti in una classe hanno:

- Stesse variabili, con gli stessi tipi
- Stessa *messagge interface*
- Stessi metodi

→ Possono differire nei valori assegnati alle variabili

Ad esempio: raggruppamento di oggetti fatti per rappresentare singole persone in una classe *person*.

Le classi sono analoghe agli *entity sets* nel modello E-R.

Esempio - definizione di una classe:

```
class employee {
        /*Variables */
                string name;
                string address;
                date start-date;
                int salary;
        /* Messages */
                int annual-salary();
                string get-name();
                string get-address();
                int set-address(string new-address);
                int employment-length();
};
```

Anche i metodi per leggere e impostare le altre variabili sono necessari con incapsulamento rigoroso (*strict encapsulation*).

I metodi sono definiti separatamente:

```
int employment-length() {
    return today() – start-date;
    }
```

```
int set-address(string new-address) {
    address = new-address;
    }
```

2.5 INHERITANCE

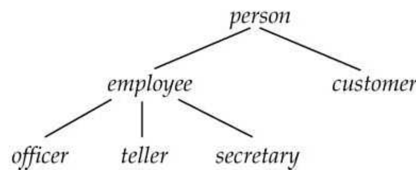
Ad esempio, la classe *bank customer* è simile alla classe *bank employee*, anche se ci sono alcune differenze:

- entrambi condividono alcune variabili e alcuni messaggi (ad esempio: *name* e *address*)
- ma ci sono variabili e messaggi specifici per ogni classe (ad esempio: *salary* per *bank employee* e *credit-rating* per *bank customer*)

Ogni *employee* è una persona; quindi *employee* è una specializzazione di *person*. Allo stesso modo, il *customer* è una specializzazione di *person*. Bisogna quindi creare classi *person*, *employee*, *customer* tenendo conto di:

- variabili/messaggi applicabili a tutte le persone associate alla classe *person*.
- variabili/messaggi specifici per i dipendenti associati alla classe *employee*; allo stesso modo per il *customer*

↳ Si devono inserire le classi in una ***specialization/IS-A hierarchy:*** variabili/messaggi appartenenti alla classe *person* sono ereditati dalle classi *employee* e *customer* → il risultato è una gerarchia di classi (notare l'analogia con la gerarchia ISA nel modello E-R):




```
class person{
string name;
string address:
};
```

```
class customer isa
person {
int credit-rating;
};
```

```
class employee isa person {
date start-date;
int salary;
};
```

```
class officer isa employee {
int office-number,
int expense-account-number,
};
```

Elenco delle variabili di oggetto nella classe *officer*:

- *office-number*, *expense-account-number*: definiti localmente
- *start-date*, *salary*: ereditati (*inherited*) da *employee*
- *name, address*: ereditati da *person*

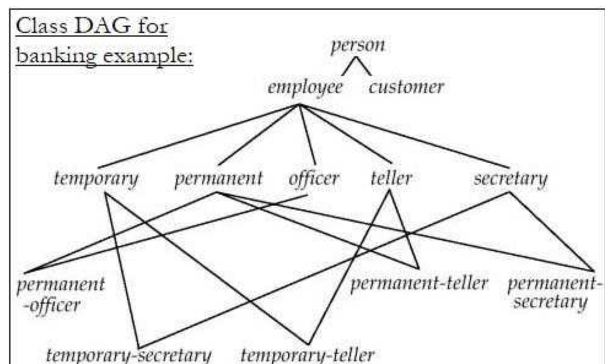
I metodi sono ereditati similmente alle variabili.

Substitutability (sostituibilità): qualsiasi metodo di una classe (per esempio *person*) può essere invocato ugualmente bene con qualsiasi oggetto appartenente a qualsiasi sottoclasse (come ad esempio sottoclasse *officer* di *person*).

Class extent (estensione della classe): insieme (*set*) di tutti gli oggetti nella classe. Vi sono due opzioni:

- L'estensione della classe *employee* include tutti gli oggetti delle classi *officer*, *teller* e *secretary*.
- L' estensione della classe *employee* include solo gli oggetti *employee* che non sono una sottoclasse come *officer*, *teller* e *secretary*
    - Questa seconda scelta è la scelta abituale nei sistemi *Object-Oriented*
    - Si può accedere alle estensioni delle sottoclassi per trovare tutti gli oggetti dei sottotipo di *employee*.

2.6 MULTIPLE INHERITANCE



Con l'ereditarietà multipla una classe può avere più di una superclasse:

- La relazione classe/sottoclasse è rappresentata da un **DAG** (*Directed Acyclic Graph*, "grafo aciclico diretto")
- È utile quando gli oggetti possono essere classificati in più modi, indipendenti l'uno dall'altro:
    - Ad esempio, *temporary*/*permanent* è indipendente da *officer*/*secretary*/*teller*
        - Si deve creare una sottoclasse per ciascuna combinazione di sottoclassi, ma non è necessario creare sottoclassi per le combinazioni che non sono possibili nel database che si sta modellando.

Una classe eredita variabili e metodi da tutte le sue superclassi.

Esiste la possibilità che si verifichino ambiguità quando una variabile/messaggio N con lo stesso nome è ereditata/o da due superclassi A e B. Non vi è nessun problema però se la variabile/messaggio è definita in una superclasse condivisa. Altrimenti, si può proseguire in due modi:

- Contrassegnarlo (*flag*) come un errore
- Rinominare le variabili ($A.N$ e $B.N$)
- Sceglierne una/uno

Concettualmente, un oggetto può appartenere a ciascuna delle diverse sottoclassi. Ad esempio, una *person* può svolgere il ruolo di *student*, *teacher* o *footballPlayer*, o qualsiasi combinazione dei tre. Si può quindi utilizzare l'ereditarietà multipla per modellare i "ruoli" (***roles***) di un oggetto, ovvero, consentire ad un oggetto di assumere uno o più insiemi (*set*) di tipi.

Ma molti sistemi insistono che un oggetto debba avere una classe più specifica rispetto alle altre (***most-specific class***), deve perciò esistere una classe a cui appartiene un oggetto che è sottoclasse di tutte le altre classi a cui l'oggetto appartiene (in riferimento all'esempio precedente, si devono creare sottoclassi come *student-teacher* e *student-teacher-footballPlayer*, per ogni combinazione). Quando molte combinazioni sono possibili, creare le sottoclassi per ogni combinazione può diventare "ingombrante".

2.7 Object identity

Un oggetto conserva la propria identità (*identity*) anche se alcuni o tutti i valori di variabili o definizioni di metodi cambiano nel tempo.

L'identità dell'oggetto è una nozione più forte rispetto a quella di identità rispetto a linguaggi di programmazione o modelli di dati non basati sull'*object orientation*.

Vediamo alcune possibili identità:

- ***Value*** (identità di valore) valore dei dati (ad es. il valore della *primary key* utilizzato in sistemi relazionali).
- ***Name*** (identità di nome) il nome è fornito dall'utente, è utilizzato per le variabili nelle procedure.
- ***Built-in*** (identità integrata) è integrata nel *data model* o nel linguaggio di programmazione
  - non è richiesto alcun *user-supplied identifier* (identificatore fornito dall'utente),
  - è la forma di identità utilizzata nei sistemi *object-oriented*.

Gli ***object identifiers*** (identificatori di oggetti) sono utilizzati per identificare in modo univoco gli oggetti. Gli identificatori di oggetti sono **univoco**:

- non ci sono due oggetti con lo stesso identificatore
- ogni oggetto ha un solo identificatore di oggetto

Ad esempio, il campo (*field*) *spouse* di un oggetto *person* può essere un identificatore di un altro oggetto *person*.

L'identificatore può essere memorizzato come campo di un oggetto, per riferirsi a un altro oggetto.

L'identificatore può essere:

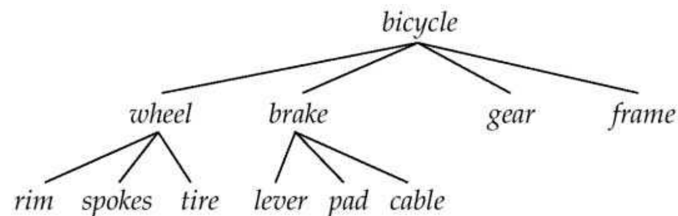
- generato dal sistema (creato dal database)
  - è più facile da usare, ma non possono essere utilizzati tra sistemi di database diversi
  - può essere ridondante se esiste già un identificativo univoco
- esterno (come ad esempio un *social-security number* o un codice fiscale)

2.8 Object Containment

Ogni componente in un *DB design* può contenere altri componenti, può essere quindi modellato come un ***object containment*** ("contenimento di oggetti"), ovvero un oggetto contenente altri oggetti sono chiamati ***composite objects*** (oggetti compositi).

Più livelli di contenimento creano una ***containment hierarchy*** (gerarchia di contenimento), dove i collegamenti sono interpretati come ***is-part-of*** (non *is-a*).

L'*object containment* consente di visualizzare i dati con diverse granularità a diversi utenti.



3 LINGUAGGI *OBJECT-ORIENTED*

Concetti *object-oriented* possono essere utilizzati in diversi modi:

- L'orientamento all'oggetto può essere usato come uno strumento di progettazione ed essere codificato in, ad esempio, un database relazionale:
  - È analogo modellare i dati con un diagramma E-R e poi convertirli in un insieme di relazioni
- Concetti *object-oriented* possono essere incorporati nel linguaggio di programmazione utilizzato per manipolare il database:
  - ***Object-relational systems:*** aggiungono *complex types* e l'*object-orientation* al *relational language*
  - ***Persistent programming languages:*** estendono gli *object-oriented programming languages* per permettere loro di trattare i database, aggiungendo concetti come *persistence* e *collections*.

(Vedremo gli *Object-relational systems* nel capitolo 9.2, nel seguito vedremo i *Persistent programming languages*.)

4 LINGUAGGI DI PROGRAMMAZIONE PERSISTENTI

I linguaggi di programmazione persistenti (*persistent programming languages*) consentono di creare oggetti, memorizzarli in un database e utilizzarli direttamente da un linguaggio di programmazione:

- Permettono che i dati vengano manipolati direttamente dal linguaggio di programmazione (non c'è bisogno di passare attraverso SQL)
- Non c'è nessuna necessità di modificare l'*explicit format* (modificare i *type*)
  - Le modifiche al *format* vengono eseguite in modo trasparente dal sistema
  - Senza un linguaggio di programmazione persistente, la modifica del *format* diventa un peso per il programmatore:
    - Più codice da scrivere
    - Più possibilità di bug
- Consentono agli oggetti di essere manipolati *in-memory* (non è necessario caricare o archiviare esplicitamente nel database)
  - Si risparmia sul codice
  - Si risparmia sugli *overhead* di *loading/storing* di grandi quantità di dati

Svantaggi dei linguaggi persistenti:

- A causa della potenza della maggior parte dei linguaggi di programmazione, è facile che si possano fare errori di programmazione che danneggiano il database.
- La complessità dei linguaggi rende l'ottimizzazione automatica di alto livello più difficile.
- Non supportano query dichiarative come invece fanno i database relazionali

4.1 PERSISTENCE OF OBJECTS

Gli approcci per rendere persistenti gli oggetti transitori includono:

- ***Persistence by Class*** ("persistenza per classe"): dichiarare persistenti tutti gli oggetti di una classe; (→semplice ma inflessibile)

- ***Persistence by Creation*** ("persistenza per creazione"): estendere la sintassi per la creazione di oggetti in modo da specificare quando un oggetto sia persistente.
- ***Persistence by Marking***: un oggetto deve persistere oltre l'esecuzione del programma se è *marked* ("marchiato, segnato") come persistente prima che il programma termini.
- ***Persistence by Reachability*** ("persistenza da raggiungibilità"): dichiarare alcuni oggetti persistenti "radici" (*root*). Gli oggetti sono persistenti se vengono riferiti (*referred*) direttamente o indirettamente ad un oggetto radice (*root*).
  - Più facile per il programmatore, ma più *overhead* per il *DB system*
  - Simile alla *garbage collection* utilizzata ad esempio in Java, che esegue anche test di raggiungibilità

4.2 OBJECT IDENTITY E PUNTATORI

A un oggetto persistente viene assegnato un ***persistent object identifier*** (identificatore di oggetto persistente).

Gradi di permanenza dell'identità:

- ***Intraprocedure:*** l'identità persiste solo durante le esecuzioni di un'unica procedura
- ***Intraprogram:*** l'identità persiste solo durante l'esecuzione di un singolo programma o query.
- ***Interprogram:*** l'identità persiste dall'esecuzione di un programma a un altro, ma ciò potrebbe cambiare se viene modificata la *storage organization* ("l'organizzazione dell'archiviazione").
- ***Persistent:*** l'identità persiste durante le esecuzioni del programma e le riorganizzazioni strutturali di dati → richiesto per *object-oriented systems*

In linguaggi *object-oriented* come C++, un *object identifier* è in realtà un puntatore *in-memory*.

Persistent pointer (puntatore persistente): persiste oltre l'esecuzione del programma. Può essere pensato come un puntatore nel database (Ad esempio, specificare il *file identifier* e l'*offset* nel file). È necessario trattare i problemi dovuti alla riorganizzazione del DB con dei ***forwarding pointers***.

4.3 ARCHIVIAZIONE E ACCESSO DI OGGETTI PERSISTENTI

Come **trovare gli oggetti** nel database:

- Dando **nomi** gli oggetti (come si darebbero ai file)
  - Impossibile per un numero elevato di oggetti.
  - Tipicamente dato solo per *class extents* e altre *collections* di oggetti, ma non per gli oggetti.
- "Esporre" (***expose***) *object identifiers* o *persistent pointers* agli oggetti
  - Possono essere immagazzinati esternamente
  - Tutti gli oggetti hanno *object identifiers*.
- Memorizzare ***collections*** di oggetti e consentire ai programmi di scorrerle per trovarvi gli oggetti richiesti
  - Modelli *collections* di oggetti = ***collection types***
  - ***Class extent***: è la *collection* di tutti gli oggetti appartenenti alla classe; di solito è mantenuta per tutte le classi che possono avere persistente oggetti.

5 SISTEMI C ++ PERSISTENTI

Il linguaggio C++ supporta l'implementazione delle caratteristiche di *persistence* senza dover cambiare linguaggio:

- C++ permette di dichiarare una classe chiamata ***Persistent_Object***, con attributi e metodi per supportare la *persistence*
- ***Overloading:*** è la possibilità di C++ di ridefinire nomi di funzioni standard ed operatori (ad esempio: +, -, l'operatore *pointer deference* ->) quando si tratta di nuovi tipi
- Le ***template classes*** di C++ aiutano a creare un sistema di tipi sicuro (*type-safe type system*) che supporti *collections* e e *persistent types*.

L’implementazione delle caratteristiche di *persistence* senza dover cambiare linguaggio con è relativamente facile, ma è più difficile da usare.

Sono stati creati dei sistemi C++ persistenti (***persistent C++ systems***) che aggiungono funzionalità al linguaggio C++ (come sono stati creati anche sistemi che evitano di cambiare linguaggio).

5.1 ODMG: C ++ Object Definition Language

L’***Object Database Management Group*** è un consorzio volto a standardizzare i database *object-oriented*:

- Si occupa in particolare di linguaggi di programmazione persistenti
- Include gli standard per C++, Smalltalk e Java
- Esistono diversi standard, tra cui ODMG-93, ODMG-2.0 e ODMG-3.0 (che è quello 2.0 con in più le estensioni a Java)
- La nostra trattazione è basata su ODMG-2.0

Gli standard ODMG C++ evitano modifiche al linguaggio C++, ed inoltre forniscono alcune importanti funzionalità mediante *template classes* e *class libraries*.

5.2 ODMG Types

- La *template class* d_Ref < class > è utilizzata per specificare *references* (*persistent pointers*)
- La *template class* d_Ref < class > è utilizzata per definire insiemi (*set*) di oggetti. I metodi includono *insert_element(e)* e *delete_element(e)*
- Sono fornite anche altre *collection classes* come d_Bag (un *set* che consente i duplicati), d_List e d_Varray (*array* di lunghezza variabile)
- Sono fornite anche le versioni d_ di molti *standard types* (ad esempio d_Long e d_string). L' interpretazione di questi *types* è *platform independent* (indipendente dalla piattaforma). I d*ynamically allocated data* (dati allocati dinamicamente) come ad esempio d_string sono allocati nel database, non nella memoria principale.

5.3 ODL: Object Definition Language (ODMG C++)

Vediamo un esempio per di ODL per ODMG C++:

```
class Branch : public d_Object {
....
}

class Person : public d_Object {
    public:
        d_String name;
          // should not use String
        d_String address;
};
```

```
class Account : public d_Object {
    private:
        d_Long balance;
    public:
        d_Long number;
        d_Set <d_Ref<Customer>>
        owners;
        int find_balance();
        int update_balance(int delta);
};
```

```
class Customer : public Person {
    public:
        d_Date member_from;
        d_Long customer_id;
        d_Ref<Branch> home_branch;
        d_Set <d_Ref<Account>> accounts;
};
```

5.4 ODL - Implementare le Relazioni

Le relazioni tra classi sono implementate mediante ***references*** (riferimenti)

Special reference types (tipi riferimento speciali) rafforzano l'integrità aggiungendo/rimuovendo *inverse links* (collegamenti inversi):

- Il tipo d_Rel_Ref <Class, InvRef> è un *reference class*, dove l'attributo InvRef di Class è l’*inverse reference* (riferimento inverso).
- Analogamente, d_Rel_Set<Class, InvRef> è usato per un insieme (*set*) di *references*

L’*Assignment method* “=” (metodo di assegnazione) della classe d_Rel_Ref è *overloaded:*

- Usa la *type definition* per trovare e aggiornare automaticamente l'*inverse link*

- Libera il programmatore dal compito di aggiornare gli *inverse link*
- Elimina la possibilità di *links* (collegamenti) incoerenti/inconsistenti

Allo stesso modo, i metodi *insert_element()* e *delete_element()* di d_Rel_Set usano la *type definition* per trovare e aggiornare automaticamente l'*inverse link*. Ad esempio:

```
extern const char _owners[ ], _accounts[ ];
class Account : public d.Object {
       ….
       d_Rel_Set <Customer, _accounts> owners;
}
  // .. Since strings can't be used in templates …
const char _owners= "owners";
const char _accounts= "accounts";
```

5.5 OML: Object Manipulation Language (ODMG C++)

Utilizza versioni persistenti di operatori C++ come new(db), ad esempio:

```
d_Ref<Account> account = new(bank_db, "Account") Account;
```

new alloca l'oggetto nel database specificato, piuttosto che in memoria. Il secondo argomento ("Account") fornisce il *typename* utilizzato nel database.

Dereference operator → se applicato su una *reference* del tipo d_Ref<Account> carica l'oggetto a cui si riferisce al *reference* in memoria (se non già presente) prima di continuare con la normale dereferenziazione C++.

Class Contructor: è metodo speciale utilizzato per inizializzare gli oggetti quando vengono creati; è chiamato automaticamente ad ogni nuova chiamata.

Class extents gestite automaticamente alla creazione dell'oggetto ed alla sua cancellazione. Ciò avviene solo per le classi per le quali questa funzione è stata specificata (è specificata tramite la *user interface*, non mediante C++). La manutenzione automatica delle *class extents* non è supportata in versioni precedenti di ODMG.

5.6 OML - Funzioni Database e Funzioni Oggetto

La classe d_Database fornisce metodi per:

- aprire un database: open(databasename)
- dare nomi agli oggetti: set_object_name(object, name)
- cercare (*look up*) gli oggetti per nome: lookup_object(name)
- rinominare oggetti: rename_object(oldname, newname)
- chiudere un database: close()

La classe d_Object è ereditata da tutte le classi persistenti:

- fornisce i metodi per allocare ed eliminare gli oggetti
- il metodo mark_modified() deve essere chiamato prima che un oggetto sia aggiornato (viene chiamato automaticamente quando viene creato l'oggetto)[2]

---

[2] Per gli aggiornamenti, è necessario chiamare un metodo speciale, mark_modified(), che indica a OODBMS di bloccare l'oggetto e aggiornare la memoria persistente al momento della transazione. La maggior parte dei prodotti OODBMS fornisce un modo per farlo automaticamente, quindi di solito non bisogna preoccuparsi di questo.

Vediamo un esempio di OML:

```
int create_account_owner(String name, String Address){
   Database bank_db.obj;
   Database * bank_db= & bank_db.obj;
   bank_db =>open("Bank-DB");
   d.Transaction Trans;
   Trans.begin();

   d_Ref<Account> account = new(bank_db) Account;
   d_Ref<Customer> cust = new(bank_db) Customer;
   cust->name - name;
   cust->address = address;
   cust->accounts.insert_element(account);
   ... Code to initialize other fields

   Trans.commit();
}
```

↳ Le *Class extent* gestite automaticamente nel database. Vediamo come:

Per accedere ad una *class extent*: d_Extent<Customer> customerExtent(bank_db);

La classe d_Extent fornisce il metodo d_Iterator<T> create_iterator() per creare un iteratore sulla *class extent*

Fornisce anche il metodo select(pred) che restituisce un iteratore (***iterator***) su oggetti che soddisfano il predicato di selezione pred.

Gli ***iterators*** aiutano a passare attraverso gli oggetti in una *collection* o *class extent*.

Le *collections* (insiemi, liste ecc…) forniscono anche il metodo create_iterator().

Vediamo un esempio per quanto riguarda gli *interators:*

```
int print_customers() {
    Database bank_db_obj;
    Database * bank_db = &bank_db_obj;
    bank_db->open ("Bank-DB");
    d_Transaction Trans; Trans.begin ();

    d_Extent<Customer> all_customers(bank_db);
    d_Iterator<d_Ref<Customer>> iter;
    iter = all_customers->create_iterator();
    d_Ref <Customer> p;
    while{iter.next (p))
          print_cust (p);  // Function assumed to be defined elsewhere
    Trans.commit();
}
```

5.7 ODMG C++: ALTRE CARATTERISTICHE

Il ***Declarative query language*** (***OQL***) è simile a SQL:

- Scrive una query in una stringa e la esegue per ottenere un insieme (*set*) di risultati.[3]

```
d_Set<d_Ref<Account>> result;
d_OQL_Query q1("select a
                from Customer c, c.accounts a
                where c.name='Jones'
                      and a.find_balance() > 100");
d_oql_execute(q1, result);
```

- Fornisce un *error handling mechanism* (meccanismo di gestione degli errori) basato sulle C++ *exceptions*, attraverso la classe d_Error
- Fornisce gli API per l'accesso allo schema di un database.

[3] In realtà non proprio un *set* ma una *bag*, poiché potrebbero essere presenti duplicati.

5.8 RENDERE "TRASPARENTE" LA *POINTER PERSISTENCE*

Rifinitura dell'approccio ODMG C++:

- Con due tipi di puntatori
- Il programmatore deve assicurarsi che mark_modified() sia chiamato, altrimenti il database potrebbe essere danneggiato

Approccio ObjectStore:

- Usa esattamente lo stesso *type* di puntatore per gli oggetti *in-memory* e del database
- La *persistence* è "trasparente" nelle *applications* (tranne quando si creano oggetti)
- Le stesse funzioni possono essere utilizzate sia su oggetti *in-memory* sia su *persistent*, poiché i *types* di puntatore sono uguali
- Implementato da una tecnica chiamata *pointer-swizzling*.
- Non è necessario chiamare mark_modified(), ogni modifica viene rilevata automaticamente.

6 SISTEMI JAVA PERSISTENTI

ODMG-3.0 definisce le estensioni di Java per la persistenza (fatto su slide 8.39-8.40)

Cap 9.2: Object-Relational Databases

Gli *Object-Relational Databases:*

- Si utilizzano per estendere il *relational data model* includendo l'*object orientation* e gli *object-oriented constructs* per gestire gli *added data types* (tipi aggiunti).
- Consentono agli attributi di tuple di avere ***complex types*** (tipi complessi), inclusi valori non atomici come le ***nested relations*** (relazioni annidate).
- Preservano le "*relational foundations*", in particolare l'accesso dichiarativo ai dati (*declarative access to data*), estendendo al contempo il potere di modellazione.
- Sono compatibili "verso l'alto" con i linguaggi relazionali esistenti.

1 RELAZIONI NIDIFICATE

Le motivazioni per cui si introducono le ***nested relations*** sono:

- Permettono domini non atomici (non atomici ≡ non indivisibili) → esempio di dominio non atomico: insieme (*set*) di numeri interi o *set* di tuple
- Consentono una modellazione più intuitiva per applicazioni con dati complessi

Proviamo a darne una definizione intuitiva: permettere una *nested relation* significa permettere le relazioni dove prima permettevamo solamente valori atomici (scalari) → permettiamo relazioni all'interno delle relazioni. Ciò mantiene le basi matematiche del modello relazionale, ma viola la 1NF.

Esempio di relazione annidata: prendiamo un sistema informativo della biblioteca. Ogni libro ha un *title*, un *set* di *authors*, un *Publisher*, e un *set* di *keywords*.

Vediamo una possibile relazione *books* non in 1NF → la chiamiamo *books*:

title	*author-set*	*publisher*	*keyword-set*
		(*name, branch*)	
Compilers	{Smith, Jones}	(McGraw-Hill, New York)	{parsing, analysis}
Networks	{Jones, Frick}	(Oxford, London)	{Internet, Web}

Vediamo una possibile relazione *books* in 1NF → la chiamiamo *flat-books*:

title	*author*	*pub-name*	*pub-branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

1.1 DECOMPOSIZIONE 4NF DI UNA RELAZIONE NIDIFICATA

Per ridurre l'esagerazione vista per la relazione *flat-books* assumiamo che valgano (*holds*) le seguenti *multivalued dependencies*: (*title*) ↠ (*author*); (*title*) ↠ (*keyword*); (*title*) ↠ (*pub-name, pub-branch*)

Si scompone quindi *flat-books* in 4NF usando i seguenti *schemas*: *(title, author); (title, keyword); (title, pub-name, pub-branch)*

Vediamo quindi il risultato:

title	*author*
Compilers	Smith
Compilers	Jones
Networks	Jones
Networks	Frick

authors

title	*keyword*
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

title	*pub-name*	*pub-branch*
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4

1.2 Problemi con lo schema 4NF

La progettazione 4NF richiede che gli utenti utilizzino i *join* nelle loro query → soluzione: si definisce, la *relational view flat-books* in 1NF viene definita mediante join di relazioni 4NF:

- PRO:
  - elimina la necessità per gli utenti di eseguire *join*,
- CONTRO:
  - perde la corrispondenza uno-ad-uno tra tuple e libri.
  - ha un'elevata ridondanza

↳ la rappresentazione mediante *nested relations* è quindi molto più naturale.

2 Tipi complessi e orientamento agli oggetti

2.1 Tipi complessi e SQL 1999

Le estensioni di SQL per supportare *complex types* includono:

- ***Collections*** *types* e ***large object*** *types* (→ le relazioni annidate sono un esempio di *collections types*)
- ***Structured*** *types* (tipi strutturati) (→strutture di record annidate come gli attributi compositi)
- ***Inheritance***
- *Object-orientation* (*object identifiers* e *references* inclusi)

La nostra trattazione si basa principalmente sullo standard SQL 1999:

- Non è completamente implementato in nessun sistema di database al momento, ma alcune funzionalità sono presenti in ciascuno dei principali sistemi di database commerciali (→ leggi il manuale del tuo sistema di database per vedere cosa supporta)
- Quando presentiamo alcune funzionalità che non sono in SQL: 1999, queste saranno indicate esplicitamente

2.2 Collection Types

Set type, tipo di dato che rappresenta gli insiemi (non in SQL: 1999):

```
create table books (
    .....
    keyword-set setof(varchar(20))
    ......
)
```

Gli insiemi *(set)* sono un'istanza dei *collection types*. Altre istanze includono:

- Array (sono supportati in SQL 1999) → Es: author-array **varchar**(20) **array**[10]
  - Può accedere agli elementi dell'array in modo usuale → Es: Ad esempio author-array[1]
- Multiset (non supportati in SQL 1999) → Es: raccolte non ordinate, in cui un elemento può verificarsi più volte

- Le relazioni annidate sono *set* di tuple → Es: SQL: 1999 supporta array di tuple

2.3 Large Object Types

Vi sono a disposizione diverse tipologie di *Large Object Types*:

- ***Clob*** → "*Character large object*". Ad esempio*:* book-review **clob** (10 KB)
- ***Blob*** → "*Binary large object*". Ad esempio*:* image **blob** (10 MB), movie **blob** (2 GB)

JDBC/ODBC fornisce metodi speciali per accedere ai *large object* "a piccoli pezzi":

- Simile all'accesso ai file del sistema operativo
- L'applicazione recupera un ***locator*** ("localizzatore/locatore") per il *large object* e quindi manipola il *large object* dall'*host language* (il linguaggio dell'host).

2.4 Structured and Collection Types

Gli ***structured types*** possono essere dichiarati e usati in SQL:

```
create type Publisher as
    (name varchar(20),
    branch varchar(20))
create type Book as
    (title varchar(20),
    author-array varchar(20) array [10],
    pub-date date,
    publisher Publisher,
    keyword-set setof(varchar(20)))
```

N.B.: la dichiarazione **setof** di *keyword-set* non è supportata da SQL 1999. Inoltre l'utilizzo di un array per memorizzare gli autori ci consente di registrare l'ordine degli autori.

Gli *structured types* possono essere utilizzati per creare tabelle:

```
create table books of Book
```

↳ simile alla relazione annidata *books*, ma con un array di autori anziché un *set*.

Gli ***structured types*** consentono di rappresentare direttamente gli **attributi compositi** dei diagrammi E-R.

↳ *Unnamed row types* possono essere utilizzati anche in SQL 1999 per definire attributi compositi. Ad esempio possiamo omettere la dichiarazione del tipo complesso *Publisher* e utilizzare invece quanto segue nel dichiarare il tipo *Book:*

```
publisher row (  name varchar(20),
                 branch varchar(20) )
```

Allo stesso modo, i ***collection types*** consentono di rappresentare direttamente **attributi multivalore** dei diagrammi E-R.

2.4.1 Tipi strutturati

Possiamo creare tabelle senza creare un tipo intermedio. Ad esempio, la tabella *books* potrebbe anche essere definita come segue:

```
create table books
  (title varchar(20),
  author-array varchar(20) array[10],
  pub-date date,
  publisher Publisher
  keyword-list setof(varchar(20))
```

I metodi possono essere parte della definizione del tipo di un tipo strutturato:

```
create type Employee as (
        name varchar(20),
        salary integer)
method giveraise (percent integer)
```

Scriviamo poi il corpo del metodo separatamente:

```
create method giveraise (percent integer) for Employee
begin
    set self.salary = self.salary + (self.salary * percent) / 100;
end
```

2.5 CREAZIONE DEI VALORI DI TIPI COMPLESSI

I valori dei tipi strutturati vengono creati utilizzando ***constructors*** (ad esempio Publisher('McGraw-Hill', 'New York'))
→ Nota Bene: un valore **non** è un oggetto

Vediamo un esempio in SQL: 1999 di costruttore:

```
create function Publisher (n varchar(20), b varchar(20))
returns Publisher
begin
        set name=n;
        set branch=b;
end
```

Ogni tipo strutturato ha un costruttore predefinito senza argomenti, altri possono essere definiti nel momento in cui è richiesto.

I valori del tipo **row** possono essere costruiti elencando i valori nelle parentesi, ad esempio:

```
row (name varchar(20),
    branch varchar(20) )
```

↳ possiamo assegnare (`McGraw-Hill',`New York') come valore del tipo visto qui sopra.

Vediamo la creazione di valori di tipi complessi in alcuni casi notevoli:

- Per costruire un array: **array** ['Silberschatz',`Korth',`Sudarshan']
- Per impostare (*set*) i valori si attributi: **set**(v1, v2, ..., vn) ← (non supportato in SQL: 1999)
- Per creare una tupla della relazione *books:*

```
('Compilers', array[`Smith',`Jones'],
  Publisher(`McGraw-Hill',`New York'),
        set(`parsing',`analysis'))
```

- Per inserire la tupla precedente nella relazione *books*:

```
insert into books
values
  ( `Compilers', array[`Smith',`Jones'],
    Publisher('McGraw Hill',`New York' ),
    set(`parsing',`analysis'))
```

2.6 INHERITANCE

Supponiamo di avere la seguente *type definition* per *people*:

```
create type Person
    (name varchar(20),
    address varchar(20))
```

Utilizzando l'ereditarietà (*inheritance*) possiamo definire i tipi *student* e *teacher*:

```
create type Student
    under Person
    ( degree varchar(20),
      department varchar(20))

create type Teacher
    under Person
    ( salary integer,
      department varchar(20))
```

I sottotipi possono ridefinire i metodi grazie all'***overriding***, usando **overriding method** invece che **method** nella dichiarazione del metodo.

2.6.1 Multiple Inheritance

SQL: 1999 non supporta l'ereditarietà multipla.

Se il nostro sistema di tipi supporta l'ereditarietà multipla, possiamo definire un tipo *Teaching Assistant* come segue:

```
create type Teaching Assistant
    under Student, Teacher
```

Per evitare un conflitto tra le due occorrenze di department, possiamo rinominarle:

```
create type Teaching Assistant
under
  Student with (department as student-dept),
  Teacher with (department as teacher-dept)
```

2.6.2 Table Inheritance

L'ereditarietà delle tabelle (*table inheritance*) consente a un oggetto di avere più *types* consentendo a un'entità di esistere in più di una tabella alla volta. Ad esempio: **create table** people **of** Person

Possiamo quindi definire le tabelle *students* e *teachers* come **sottotabelle** (*subtables*) di *people:*

```
create table students of Student
    under people
create table teachers of Teacher
    under people
```

Ogni tupla in una sottotabella (ad esempio *students* e *teachers)* è implicitamente presente nelle sue "sopratabelle" (ad es. *people*)

L'ereditarietà multipla è possibile con le tabelle, proprio come è possibile con i types:

```
create table teaching-assistants of Teaching Assistant
    under students, teachers
```

Ricorda: l'ereditarietà multipla non è supportata in SQL 1999.

Table Inheritance - Roles: l'ereditarietà delle tabelle è utile per modellare i **ruoli**, e consente a un valore di avere più *types*, senza avere un più specifico *type* (a differenza della *type inheritance*)

- Ad esempio, un oggetto può essere presente nelle sottotabelle *students* e *teachers* contemporaneamente, senza dover essere in una sottotabella *students-teacher* al di sotto sia di *students* che di *teachers.*
- Un oggetto può guadagnare/perdere ruoli: ciò corrisponde all'inserimento/eliminazione dell'oggetto da un sottotabella

Ricorda: SQL 1999 richiede che i valori abbiano un *most specific type*, perciò quando detto sopra non è applicabile a SQL 1999

Table Inheritance – Consistency requirements: esistono dei requisiti di coerenza per sottotabelle e supertabelle

- Ogni tupla di una supertabelle (ad esempio *people)* può corrispondere al massimo una tupla in ciascuno delle sottotabelle (ad es. *students* e *teachers)*
- Esiste Vincolo aggiuntivo in SQL: 1999: tutte le tuple che corrispondono l'una all'altra (ovvero, con gli stessi valori per gli attributi ereditati) devono essere derivate da una tupla (inserita in una tabella).
  - Cioè, ogni entità deve avere un *most specific type*
  - Non possiamo avere una tupla in *people* corrispondente ad una tupla che sia in *students* che in *teachers*

Table Inheritance – Storage alternatives: esistono due alternative di archiviazione:

1. Memorizzare solo gli attributi locali e la *primary key* della supertabella nella sottotabella
   - Gli attributi ereditati sono derivati per mezzo di una join con la supertabella
2. Ogni tabella memorizza tutti gli attributi, siano essi ereditati o definiti localmente
   - Le supertabelle contengono implicitamente (gli attributi ereditati di) tutte le tuple nelle loro sottotabelle
   - L'accesso a tutti gli attributi di una tupla è più veloce: non è richiesto alcun join
   - Se le entità devono avere un *most specific type*, la tupla viene memorizzata solo in una tabella, quella dove è stata creata. Altrimenti, ci potrebbe essere ridondanza.

2.7 Reference Types

I linguaggi *object-oriented* offrono la possibilità di creare e di fare riferimento agli oggetti. In SQL: 1999 i riferimenti sono alle tuple e devono essere ***scoped*** (vale a dire, possono solo puntare a tuple in una tabella specificata). Studieremo prima come definirli ed in seguito come utilizzarli.

Vediamo un esempio di ***reference declaration*** in SQL: 1999, definendo un type *Department* con un campo *name* e una campo *head*, che è un riferimento al tipo *Person,* avente la tabella *people* come scope:

```
create type Department(
    name varchar(20),
    head ref(Person) scope people)
```

Possiamo quindi creare una tabella *departments* come segue: **create table** *departments* **of** *Department*

Possiamo omettere la dichiarazione **scope** *people* dalla dichiarazione del tipo (**create type** *Department*) e fare invece un'aggiunta all'istruzione **create table**:

```
create table departments of Department
        (head with options scope people)
```

2.7.1 Inizializzazione dei valori di *Reference types*

In Oracle, per creare una tupla con un *reference value*, possiamo prima creare la tupla con un riferimento a *null* e quindi impostare il riferimento separatamente usando la funzione **ref**(p) applicata a una *tuple variable.* Ad esempio, per creare un reparto con nome CS la cui *head* si riferisca alla persona di nome John, usiamo:

```
insert into departments
    values (`CS', null)
update departments
    set head = (select ref(p)
                from people as p
                where name=`John')
where name = `CS'
```

SQL: 1999 non supporta la funzione **ref**(p) ma richiede invece un attributo speciale da dichiarare per memorizzare l'*object identifier.* Il *self-referential attribute* è dichiarato aggiungendo una *clause* **ref is** all'istruzione **create table**:

```
create table people of Person
ref is oid system generated
```

↳ *oid* è un attributo *name*, non una *keyword.*

Per ottenere il riferimento a una tupla, la subquery mostrata in precedenza dovrebbe utilizzare **select** *p.oid* invece che **select ref**(*p*)

2.8 USER GENERATED IDENTIFIERS

SQL: 1999 consente gli identificatori generati dall'utente, ovvero consente agli *object identifier* di essere creati dall'utente. Il *type* dell'*object-identifier* deve essere specificato come parte della *type definition* della tabella referenziata, e la *table definition* deve specificare che il riferimento è generato dall'utente. Ad esempio:

```
create type Person
    (name varchar(20)
     address varchar(20))
  ref using varchar(20)

create table people of Person
  ref is oid user generated
```

Quando si crea una tupla, è necessario fornire un valore univoco per l'identificatore (si presume che sia il primo attributo): **insert into** *people* **values** ('01284567', 'John', `23 Coyote Run')

Possiamo quindi utilizzare il valore dell'identificatore quando si inserisce una tupla in *departments* (ciò evita la necessità di avere una query separata per recuperare l'identificatore). Ad esempio:

```
insert into departments
  values(`CS', `02184567')
```

È anche possibile utilizzare un *primary key value* esistente come *object identifier*, includendo la *clause* **ref from** e dichiarare che il *reference* deve essere derivato:

```
create type Person
      ( name varchar(20) primary key,
        address varchar(20)  )
  ref from(name)
create table people of Person
  ref is oid derived
```

Per inserire una tupla per *departments* possiamo usare: **insert into** *departments* **values**(`CS',`John')

3 QUERY CON TIPI COMPLESSI

3.1 PATH EXPRESSIONS

Per trovare *name* ed *address* dei responsabili di tutti i *departments*:

```
select head−>name, head−>address
from departments
```

Un'espressione come "*head−>name*" è chiamata ***path expression***. Le *path expression* aiutano a evitare join espliciti. Se la *head* di *department* non fosse un *reference*, sarebbe necessario un join tra *departments* e *people* per ottenere *address*. Le *path expressions* rendono l'espressione della query molto più semplice per l'utente.

3.2 QUERY CON I TIPI STRUTTURATI

Per trovare *title* e *name* del *publisher* di ciascun *book*:

```
select title, publisher.name
from books
```

Notare l'uso della ***dot notation*** (*publisher.name*) per accedere ai campi dell'attributo composito (tipo strutturato) *publisher.*

3.3 COLLECTION-VALUE ATTRIBUTES

I *collection-value attributes* possono essere trattati in modo molto simile alle relazioni, utilizzando la parola chiave **unnest**. La relazione *books* ha un attributo ***array-valued*** chiamato *author-array* ed un attributo ***set-valued*** chiamato *keyword-set*.

Ad esempio, per trovare tutti i libri che contengono la parola "database" come una delle loro keywords:

```
select title
from books
where 'database' in (unnest(keyword-set))
```

Nota: la sintassi di qui sopra è valida in SQL 1999, ma l'unico *collection type* supportato da SQL 1999 è l'*array type*

Per ottenere una relazione contenente le coppie del tipo "*title*, *author-name*" per ogni libro e ogni autore del libro:

```
select B.title, A
from books as B, unnest (B.author-array) as A
```

Possiamo accedere ai singoli elementi di un array usando gli indici. Ad esempio, se sappiamo che un particolare libro ha tre autori, potremmo scrivere:

```
select author-array[1], author-array[2], author-array[3]
from books
where title = `Database System Concepts
```

3.4 UNNESTING

La trasformazione di una relazione nidificata (*nested relation*) in una forma con meno (o nessun) *relation-valued attribute* è stata chiamata ***unnesting***. Ad esempio:

```
select title, A as author, publisher.name as pub_name,
        publisher.branch as pub_branch, K as keyword
from books as B, unnest(B.author-array) as A, unnest (B.keywordlist) as K
```

3.5 NESTING

L'**annidamento** (*nesting*) è l'opposto dell'*unnesting*, e si indica con ciò il creare un *collection-valued attribute*. (SQL 1999 non supporta il *nesting*).

Il *nesting* può essere fatto in modo simile all'aggregazione, ma usando la funzione `set ()` al posto di un'operazione di aggregazione, per creare un insieme (*set*).

Vediamo ora due esempi:

1. Per nidificare (*nest*) la relazione *flat-books* sull'attributo *keyword*:

```
select title, author, Publisher(pub_name, pub_branch) as publisher,
set(keyword) as keyword-list
from flat-books
groupby title, author, publisher
```

2. Per nidificare (*nest*) la relazione *flat-books* sia sull'attributo *author* sia su *keyword*:

```
select title, set(author) as author-list,
Publisher(pub_name, pub_branch) as publisher,
set(keyword) as keyword-list
from flat-books
groupby title, publisher
```

Un altro approccio alla creazione di relazioni nidificate consiste nell'utilizzare sottoquery nella clausola *select:*

```
select title,
     ( select author
       from flat-books as M
       where M.title=O.title) as author-set,
     Publisher(pub-name, pub-branch) as publisher,
     (select keyword
       from flat-books as N
       where N.title = O.title) as keyword-set
from flat-books as O
```

È possibile utilizzare la clausola **orderby** nella *nested query* (query nidificata) per ottenere una *collection* ordinata → può quindi creare arrays, diversamente dall'approccio precedente.

4 Funzioni e procedure

SQL 1999 supporta funzioni e procedure:

- Le funzioni/procedure possono essere scritte in SQL stesso o in un linguaggio di programmazione esterno.
- Le funzioni sono particolarmente utili con tipi di dati specializzati come immagini e oggetti geometrici (es: funzioni per verificare se i poligoni si sovrappongono o per confrontare le immagini per similarità)
- Alcuni database supportano ***table-valued functions***, che possono restituire come risultato una relazione

SQL 1999 supporta anche un ricco set di *imperative constructs*, inclusi *loops*, *if-then-else*, *assignment*.

Molti database hanno *proprietary procedural extensions* (estensioni procedurali proprietarie) a SQL che differiscono da SQL 1999.

4.1 Funzioni SQL

Possiamo, per fare un esempio, definire una funzione che, in base al titolo di un libro, restituisce il conteggio del numero di autori (sul 4NF *schema* con le relazioni *books4* e *authors*):

```
create function author-count(name varchar(20))
returns integer
begin
     declare a-count integer;
     select count(author) into a-count
     from authors
     where authors.title=name
     return a=count;
end
```

Altro esempio: trovare i titoli di tutti i libri che hanno più di un autore:

```
select name
from books4
where author-count(title)> 1
```

4.2 Metodi SQL

I metodi possono essere visti come funzioni associate a tipi strutturati. Hanno un primo parametro implicito chiamato ***self*** che è impostato sullo *structured-type value* su cui viene invocato il metodo. Il codice del metodo può fare riferimento agli attributi dello *structured-type value* utilizzando la variabile ***self*** (per esempio **self** .a).

4.3 PROCEDURE IN SQL

La funzione *author-count* potrebbe invece essere scritta come la seguente **procedura**:

```
create procedure author-count-proc (in title varchar(20),
out a-count integer)
begin
select count(author) into a-count
from authors
where authors.title = title
end
```

Le procedure possono essere richiamate sia da una procedura SQL sia da embedded SQL, utilizzando una *call statement* (l'istruzione di chiamata). Ad esempio:

```
declare a-count integer;
call author-count-proc(`Database systems Concepts', a-count);
```

SQL 1999 consente più di una funzione/procedura con lo stesso nome (→ ***name overloading***), a condizione che il numero di argomenti sia diverso, o almeno i *type* degli argomenti differiscano.

4.4 FUNZIONI/PROCEDURE IN LINGUAGGI ESTERNI

SQL: 1999 consente l'uso di funzioni e procedure scritte in altri linguaggi, come C o C++. Vediamo come dichiarare procedure e funzioni in linguaggi esterni:

```
create procedure author-count-proc(in title varchar(20),out count integer)
language C
external name' /usr/avi/bin/author-count-proc'

create function author-count(title varchar(20))
returns integer
language C
external name '/usr/avi/bin/author-count'
```

Ma conviene usare funzioni/procedure scritte in linguaggi esterni?

- Vantaggi:
  - sono più efficienti per molte operazioni ed hanno più "potenza espressiva"
- Svantaggi:
  - Potrebbe essere necessario caricare il codice che implementa la funzione nel *database system* ed eseguirlo nello spazio degli indirizzi del sistema di database
    - rischio di corruzione accidentale delle strutture del database
    - rischio per la sicurezza, poiché si consente agli utenti l'accesso a dati non autorizzati
  - Esistono alternative che offrono una buona sicurezza a costo di prestazioni potenzialmente peggiori
  - L'esecuzione diretta nello spazio del *database system* viene utilizzata quando l'efficienza è più importante della sicurezza

4.5 SICUREZZA NELLE ROUTINES IN LINGUAGGI ESTERNI

Per far fronte a problemi di sicurezza si hanno due possibilità:

1. Usare le tecniche **sandbox**, con un linguaggio sicuro come Java, che non può essere usato per accedere/danneggiare altre parti del codice del database
2. Oppure, eseguire funzioni/procedure del linguaggio esterno in un processo separato, senza accesso alla memoria del processo del database → parametri e risultati comunicati tramite *inter-process communication*
   ↳ Entrambi hanno performance overhead

Molti *database system* supportano entrambi gli approcci di qui sopra, così come l'esecuzione diretta (*direct executing*) nell'*address space* del *database system.*

4.6 COSTRUTTI PROCEDURALI

SQL 1999 supporta una ricca varietà di costrutti procedurali.

- ***Compound*** *statement:*
    - È della forma **begin ... end**,
    - Può contenere più istruzioni SQL tra **begin** ed **end**.
    - Le variabili locali possono essere dichiarate all'interno di un *compound statement*
- ***While*** *and* ***repeat*** *statements:*

```
declare n integer default 0;
while n < 10 do
    set n = n+1
end while

repeat
    set n = n – 1
until n = 0
end repeat
```

- ***For*** *loop:*
    - Consente l'iterazione su tutti i risultati di una query
    - Ad esempio, per trovare il totale di tutti i *balances* sul *Perryridge branch:*

```
declare n integer default 0;
for r as
    select balance from account
    where branch-name = 'Perryridge'
do
    set n = n + r.balance
end for
```

- *Conditional statement (**if-then-else**):*
    - Ad esempio, per trovare la somma dei *balance* per ciascuna delle tre categorie di *account* (con *balance* <1000,> = 1000 e <5000,> = 5000):

```
if r.balance < 1000
then set l = l + r.balance
elseif r.balance < 5000
then set m = m + r.balance
else set h = h + r.balance
end if
```

- SQL 1999 supporta anche un'istruzione **case** simile all'istruzione del linguaggio C
- Segnalazione ***exception conditions*** e la dichiarazione degli ***exception handlers****:*

```
declare out_of_stock condition
declare exit handler for out_of_stock
begin
…
.. signal out-of-stock
end
```

↳ L'*handler* qui è **exit** - causa l' uscita dal corpo di codice compreso tra **begin** ...**end** se viene attivato il **signal** out-of-stock. →(Sono possibili altre azioni in seguito al segnale di eccezione).

5 CONFRONTO TRA *OBJECT-ORIENTED* E *OBJECT-RELATED*

Facciamo un breve riepilogo dei punti di forza di vari sistemi di database:

- ***Relational systems:*** tipi di dati semplici, potenti linguaggi query, alta protezione.

- ***Persistent-programming-language-based OODBs:*** tipi di dati complessi, query con linguaggi di programmazione, alte prestazioni.
- ***Object-relational systems:*** tipi di dati complessi, potenti linguaggi query, alta protezione.

Nota bene: molti sistemi reali non rispettano questi limiti, ma stanno "nel mezzo", o hanno confini "sfocati". Ad esempio, il linguaggio di programmazione persistente costruito come "*wrapper*" su un database relazionale offre i primi due vantaggi, ma può avere prestazioni scadenti.

5.1 TROVARE TUTTI I DIPENDENTI DI UN MANAGER

Vediamo per ultima la procedura per trovare tutti i *employee* che lavorano direttamente o indirettamente per un *mgr*.

La relazione *manager*(*empname, mgrname*) specifica chi lavora direttamente per chi.

Il risultato è memorizzato in *empl* (*name*).

```
create procedure findEmp(in mgr char(10))
begin
    create temporary table newemp(name char(10));
    create temporary table temp(name char(10));
    insert into newemp -- store all direct employees of mgr in newemp
        select empname
        from manager
        where mgrname = mgr
    repeat
        insert into empl -- add all new employees found to empl
            select name
            from newemp;
        insert into temp -- find all employees of people already found
            (select manager.empname
              from newemp, manager
              where newemp.empname = manager.mgrname;
              )
          except ( -- but remove those who were found earlier
              select empname
              from empl
              );
        delete from newemp; -- replace contents of newemp by contents of temp
        insert into newemp
              select *
              from temp;
        delete from temp;
    until not exists(select* from newemp) -- stop when no new employees are found
    end repeat;
end
```

Cap 9: Object-Based Databases

1 Object-Relational Data Models

Vogliamo estendere il *relational data model* includendo l'*object orientation* ed i costrutti per gestire i tipi di dati aggiunti.

- Questi *data models* consentono agli attributi delle tuple di avere tipi complessi, inclusi valori non atomici come le relazioni annidate (*nested relations*).
- Preservano inoltre i fondamenti dei DB relazionali, in particolare l'accesso dichiarativo ai dati, aumentando al contempo le possibilità e potenzialità di modellazione.
- Sono anche compatibili ai linguaggi relazionali esistenti

Perché necessitiamo di **tipi di dato complessi** (*complex data types*)?

- Per poter utilizzare domini non atomici (atomici = indivisibili), ad esempio: un insieme di numeri interi, o un set[1] di tuple
- Consentono una modellazione più intuitiva per *applications* che trattano dati complessi

Cerchiamo di darne una definizione intuitiva:

- Vogliamo poter utilizzare relazioni laddove utilizziamo valori atomici (scalari): vogliamo perciò relazioni all'interno delle relazioni (**relazioni annidiate**)
- Vogliamo mantenere i fondamenti matematici del modello relazionale
- Tutto ciò però vìola prima forma normale (1NF).

2 Relazioni Annidiate

Vediamo un esempio di *nested* relation (relazione annidata): trattiamo un sistema informativo di una biblioteca. Ogni libro ha un titolo, un set di autori, un editore ed un set di parole chiave. Vediamo quindi la relazione *books* (non è in 1NF !!!):

title	*author-set*	*publisher*	*keyword-set*
		(*name, branch*)	
Compilers	{Smith, Jones}	(McGraw-Hill, New York)	{parsing, analysis}
Networks	{Jones, Frick}	(Oxford, London)	{Internet, Web}

Decomposizione 4NF di relazione nidificata:

Bisogna rimuovere l'ambiguità di *books* assumendo che le seguenti dipendenze multi-valore valgano (*hold*):

- $title \twoheadrightarrow author$ $\quad$ $title \twoheadrightarrow keyword$ $\quad$ $title \twoheadrightarrow pub\text{-}name, pub\text{-}branch$

Si può quindi scomporre *books* in 4NF usando i seguenti *schemas*:

- (*title, author*) $\quad$ (*title, keyword*) $\quad$ (*title, pub-name, pub-branch*)

[1] Ricorda: il **set** è, in informatica, un tipo di dato astratto consistente in una collezione di valori disposti in ordine casuale e senza valori ripetuti. Corrisponde al concetto matematico di insieme, ma con la restrizione che deve essere finito.
Eccezion fatta per la sequenza e per il fatto che non ci sono valori ripetuti, il set è uguale alla lista. Il set può essere concepito come un vettore associativo (mappatura parziale) in cui il valore di ogni coppia di valori chiave viene ignorato.

title	author
Compilers	Smith
Compilers	Jones
Networks	Jones
Networks	Frick

authors

title	keyword
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

title	pub-name	pub-branch
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4

Si evidenziano però alcuni problemi con lo schema 4NF:

- La progettazione 4NF richiede che gli utenti utilizzino i *join* nelle loro query.
- La *relational view* in 1NF *flat-books* è definita mediante *join* di relazioni 4NF → elimina la necessità per gli utenti di eseguire *join*, ma perde la corrispondenza uno-a-uno tra tuple e libri ed presenta molte ridondanze

La rappresentazione delle relazioni annidate è molto più "naturale" quindi, rispetto a quella in 4NF!

3 Tipi complessi e SQL 1999

Le estensioni di SQL per supportare tipi complessi includono:

- *Collection types* e *large object types*
    - Le relazioni nidificate sono un esempio di *collection type*
- ***Structured types*** (tipi strutturati)
    - Strutture annidate come attributi compositi
- **Ereditarietà** (inheritance)
- *Object orientation* (orientamento all'oggetto)
    - Inclusi gli identificatori di oggetti ed i riferimenti a oggetto

La nostra descrizione si basa principalmente sullo standard SQL 1999:

- Non completamente implementato in nessun sistema di database al momento
- Ma alcune funzionalità sono presenti in ciascuno dei principali sistemi di database commerciali
    ↳ Leggi il manuale del tuo sistema di database per vedere cosa supporta

4 Structured types in SQL

•• I tipi strutturati possono essere dichiarati e utilizzati in SQL (`final` e `not final` indicano se sia possibile creare sottotipi (*subtypes*):

```
create type Name as
   (firstname    varchar(20),
    lastname     varchar(20))
    final

create type Address as
   (street    varchar(20),
    city      varchar(20),
    zipcode   varchar(20))
    not final
```

•• I tipi strutturati possono essere utilizzati per creare tabelle con attributi compositi

```
create table customer (
   name        Name,
   address     Address,
   dateOfBirth date)
```

↳ la notazione con il punto (.) è utilizzata per riferirsi ai singoli componenti (es: `nome.firstname`)

•• I tipi strutturati possono essere utilizzati per creare *row types* definiti dall'utente:

```
create type CustomerType as (
    name Name,
    address Address,
    dateOfBirth date)
    not final
```

•• Si può quindi volendo creare una tabella le cui righe sono di un tipo definito dall'utente:

```
create table customer of CustomerType
```

4.1 STRUCTURED TYPES E METODI

Si possono aggiungere delle *method declaration* con i tipi strutturati:

```
method ageOnDate (onDate date)
    returns interval year
```

In alternativa il corpo del metodo può essere scritto separatamente:

```
create instance method ageOnDate (onDate date)
      returns interval year
      for CustomerType
begin
      return onDate - self.dateOfBirth;
end
```

Grazie al metodo ora possiamo facilmente trovare l'età di ogni cliente:

```
select name.lastname, ageOnDate (current_date)
from customer
```

4.2 EREDITARIETÀ

Supponiamo di avere la seguente definizione di tipo per le persone:

```
create type Person
    (name varchar(20),
     address varchar(20))
```

Utilizziamo l'ereditarietà per definire i tipi di studenti e insegnanti a partire dal type *person*:

```
create type Student
  under Person
  (degree       varchar(20),
   department  varchar(20))
create type Teacher
  under Person
  (salary       integer,
   department  varchar(20))
```

I sottotipi possono ridefinire i metodi utilizzando **overriding method** al posto che la sola keyword **method** nella dichiarazione del metodo.

4.3 EREDITARIETÀ MULTIPLA

SQL 1999 e SQL 2003 non supportano l'ereditarietà multipla. Se il nostro sistema di *types* supporta però l'ereditarietà multipla, possiamo definire, per continuare con il nostro esempio, un *type* per l'assistente all'insegnamento nella maniera che segue:

```
create type Teaching Assistant
    under Student, Teacher
```

Per evitare un conflitto tra le due occorrenze di *department*, possiamo rinominare:

```
create type Teaching Assistant
    under
        Student with (department as student_dept),
        Teacher with (department as teacher_dept)
```

4.4 Requisiti di coerenza per le sotto-tabelle

Requisiti di coerenza per sotto-tabelle e super-tabelle (*subtables* & *supertables*).

- Ogni tupla della *supertable* (ad esempio *person*) può corrispondere al massimo a una tupla in ciascuna delle *subtables* (ad esempio *student* e *teacher*)
- Esiste un vincolo aggiuntivo in SQL 1999: Tutte le tuple che corrispondono l'una all'altra (vale a dire, con gli stessi valori per gli attributi ereditati) devono essere derivate da una tupla (inserita in una tabella).
  ↳ Cioè, ogni entità deve avere un tipo più specifico, non possiamo avere una tupla in *person* che corrisponde a una tupla sia in *student* sia in *teacher*.

5 Array e Multiset Types in SQL

Vediamo un esempio di dichiarazione di *array* e *multiset* [2]:

```
create type Publisher as
    (   name        varchar(20),
        branch      varchar(20)     )

create type Book as
    (   title           varchar(20),
        author-array    varchar(20) array [10],
        pub-date        date,
        publisher       Publisher,
        keyword-set     varchar(20) multiset    )

create table books of Book
```

È simile alla nested relation *books* vista precedentemente, ma con un array di autori invece di un set.

5.1 Creazione di Collection Values

Un array quindi si costruisce in questo modo:

```
array ['Silberschatz',`Korth',`Sudarshan']
```

Un multiset si costruisce in questo modo:

```
multisetset ['computer', 'database', 'SQL']
```

Per creare una tupla di un type definito dalla relazione *books*:

```
('Compilers',  array[`Smith',`Jones'],
        Publisher  (`McGraw-Hill',`New York'),
        multiset   [`parsing',`analysis' ]   )
```

Per inserire questa tupla nella relazione *books*:

---

[2] A **multiset** (aka *bag* or *mset*) is a generalization of the concept of a *set* that, unlike a set, allows multiple instances of the multiset's elements. For example, $\{a, a, b\}$ and $\{a, b\}$ are different multisets although they are the same set. However, order does not matter, so $\{a, a, b\}$ and $\{a, b, a\}$ are the same multiset.

```
insert into books
values
    ('Compilers', array['Smith','Jones'],
        Publisher ('McGraw-Hill','New York'),
        multiset ['parsing','analysis' ])
```

5.2 Query sui Collection-Valued Attributes

Vediamo come si può scrivere una query per trovare tutti i libri che contengono la parola "database" come parola chiave:

```
select title
from books
where 'database' in (unnest(keyword-set ))
```

Possiamo accedere ai singoli elementi di un array usando gli indici, ad esempio se sappiamo che un particolare libro ha tre autori, potremmo scrivere:

```
select author-array[1], author-array[2], author-array[3]
from books
where title = `Database System Concepts'
```

Per ottenere una relazione contenente coppie nella forma "*title*, *author-name*" per ogni libro e ogni autore del libro:

```
select B.title, A.author
from books as B, unnest (B.author-array) as A (author )
```

Per conservare l'ordine delle informazioni, utilizziamo **`with ordinality`**:

```
select B.title, A.author, A.position
from books as B, unnest (B.author-array) with ordinality as
        A (author, position )
```

6 Annidamento (nesting)

6.1 Unnesting

La trasformazione di una relazione nidificata in una forma con meno (o nessuno) attributi *relational-valued* è chiamata ***un-nesting***. Ad esempio:

```
select title, A as author, publisher.name as pub_name,
        publisher.branch as pub_branch, K.keyword
from books as B, unnest(B.author_array ) as A (author ),
     unnest (B.keyword_set ) as K (keyword )
```

6.2 Nesting

Il ***nesting*** (annidamento) è l'opposto dell'un-nesting, ovvero è la creazione di un attributo *collection-valued* (SQL 1999 non supporta il *nesting*!).

Il *nesting* può essere fatto in modo simile all'*aggregation*, ma usando la funzione **`colect()`** al posto di un'aggregazione, per creare un *multiset*. Vediamo un esempio: annidare la relazione *flat-book* sull'attributo *keyword*:

```
select title, author, Publisher (pub_name, pub_branch ) as publisher,
        collect (keyword) as keyword_set
from flat-books
groupby title, author, publisher
```

Per annidare sia sugli autori sia sulle *keyword*:

```
select title, collect (author ) as author_set,
        Publisher (pub_name, pub_branch) as publisher,
                collect  (keyword ) as keyword_set
from   flat-books
group by title, publisher
```

Vediamo qui la versione 1NF (*flat-book*) della relazione nidificata *book*:

title	*author*	*pub-name*	*pub-branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

Un altro approccio alla creazione di relazioni nidificate consiste nell'utilizzare sottoquery nella `select`.

```
select title,
        array ( select author
          from authors as A
          where A.title = B.title
          order by A.position) as author_array,
        Publisher (pub-name, pub-branch) as publisher,
        multiset (select keyword
          from keywords as K
          where K.title = B.title) as keyword_set
from books4 as B
```

7 Object-Identity & Reference Types

Definiamo un tipo *departement* con un campo *name* e un campo *head* che è un **riferimento** al tipo *Person*, con la *people* come *scope*:

```
create type Department (
        name varchar (20),
        head ref (Person) scope people)
```

Possiamo quindi creare la tabella *departments* come segue:

```
create table departments of Department
```

Possiamo omettere "**`scope`** `people`" dalla dichiarazione del type e invece fare una aggiunta allo statement **`create table`**:

```
create table departments of Department
    (head with options scope people)
```

7.1 Inizializzare i Reference-Typed Values

Per creare una tupla con un ***reference*** *value*, possiamo prima creare la tupla con un riferimento a *null* e quindi impostarlo separatamente, successivamente:

```
insert into departments
    values (`CS', null)
update departments
   set head = (select p.person_id
               from people as p
                 where name = `John')
   where name = `CS'
```

7.2 USER-GENERATED IDENTIFIERS

Il type dell'*object identifier* deve essere specificato nella definizione del *type* della *reference table*. Inoltre la definizione della tabella deve specificare che il riferimento è generato dall'utente (***user-generated reference***):

```
create type Person
   (name varchar(20)
    address varchar(20))
  ref using varchar(20)
create table people of Person
  ref is person_id user generated
```

Quando si crea una tupla, è necessario fornire un valore univoco per l'identificatore:

```
insert into people (person_id, name, address ) values
  ('01284567', 'John', `23 Coyote Run')
```

È quindi possibile utilizzare il valore dell'identificatore quando si inserisce una tupla in *departments*. Bisogna evitare query separate per ottenere l'identificatore:

```
insert into departments
values(`CS', `02184567')
```

È possibile utilizzare una *primary key* esistente come identificatore:

```
create type Person
     (name varchar (20) primary key,
      address varchar(20))
   ref from (name)
create table people of Person
   ref is person_id derived
```

Quando si inserisce una tupla in *departments*, possiamo quindi usarla

```
insert into departments
   values(`CS',`John')
```

8 PATH EXPRESSIONS

Iniziamo con un esempio: "trova i nomi e gli indirizzi dei responsabili di tutti i reparti", lo possiamo scrivere con la seguente notazione:

```
select head –>name, head –>address
from departments
```

Un'espressione come "`head -> name`" è chiamata ***path expression***. Le *path expression* aiutano a evitare join espliciti:

- Se il capo di dipartimento (*department head*) non fosse una *reference*, sarebbe necessario un join tra *departments* e *people*
- Le *path expression* rendono l'espressione della query molto più semplice per l'utente

9 IMPLEMENTAZIONE DI FUNZIONALITÀ O-R

Le funzionalità O-R (Object-Relation) sono mappate su *relation schemas* similmente alle funzionalità E-R (Entity-Relation)

Implementazione delle sotto-tabelle: Ogni *subtabel* memorizza la *primary key* e gli definiti in quella tabella oppure ogni *subtabel* memorizza sia attributi definiti localmente che attributi ereditati dalle super-tabelle.

10 LINGUAGGI DI PROGRAMMAZIONE PERSISTENTI

Esistono linguaggi estesi con costrutti adatti a gestire dati persistenti. Il programmatore può manipolare direttamente i dati persistenti e così non è necessario recuperarli in memoria e memorizzarli su disco (a differenza dell'embedded SQL)

Si definiscono oggetti persistenti:

- **per classe** - vi è una dichiarazione esplicita di persistenza
- **per creazione** - esiste sintassi speciale per creare oggetti persistenti
- **per *marking*** - rende gli oggetti persistenti dopo la creazione
- **per raggiungibilità** - l'oggetto è persistente se dichiarato esplicitamente o se raggiungibile da un oggetto persistente

11 OBJECT-IDENTITY & PUNTATORI

Esistono diversi gradi di permanenza dell'*object identity*:

- **Intra-procedurale**: solo durante l'esecuzione di una singola procedura
- **Intra-programma**: solo durante l'esecuzione di un singolo programma o query
- **Inter-programma**: attraverso le esecuzioni del programma, ma non se il formato di archiviazione dei dati sul disco cambia
- **Persistente**: interprogramma, oltre a persistente per le riorganizzazioni dei dati

Sono state implementate versioni "persistenti" di C++ e Java: ODMG C++, ObjectStore, Java Database Objects (JDO) ecc…

12 CONFRONTO DEI DATABASE O-O E O-R

Vediamo quindi un veloce confronto tra DB *Object-Oriented* e *Object-Related*:

- ***Relational DB systems***: tipi di dati semplici, linguaggi di query potenti, protezione elevata.
- ***Persistent-programming-language-based OODBs***: tipi di dati complessi, integrazione con il linguaggio di programmazione, alte prestazioni.
- ***Object-relational systems***: tipi di dati complessi, linguaggi di query potenti, protezione elevata.

Nota: molti sistemi reali non rispettano chiaramente questi questi limiti, e si collocano a metà strada tra le varie definizioni (ad esempio il linguaggio di programmazione persistente costruito come un "*wrapper*" (involucro esterno) su un database relazionale offre i primi due vantaggi, ma può avere prestazioni scadenti).

Cap 10: XML

1 INTRODUZIONE A XML

XML sta per ***eXtensible Markup Language***. È stato definito dal WWW Consortium (anche W3C). Derivato da SGML (*Standard Generalized Markup Language*), ma più semplice da usare rispetto ad esso.

In XML documenti contengono dei *tag* che forniscono informazioni aggiuntive sulle sezioni del documento (ad es: `<title> XML </ title> <slide> Introduzione ... </ slide>`

È un linguaggio **estensibile**, a differenza di HTML, dove gli utenti possono aggiungere nuovi *tag* e specificare separatamente come deve essere gestito il *tag* per essere visualizzato.

La possibilità di specificare nuovi *tag* e di creare strutture di *tag* nidificate rende XML un ottimo modo per scambiare dati, non solo documenti. Gran parte dell'uso di XML è infatti nelle *data exchange applications*, non in sostituzione di HTML. I tag rendono infatti i dati (relativamente) auto-documentanti. Vediamone un esempio:

```
<bank>
    <account>
        <account_number> A-101    </account_number>
        <branch_name>     Downtown </branch_name>
        <balance>           500       </balance>
    </account>
    <depositor>
        <account_number> A-101    </account_number>
        <customer_name> Johnson </customer_name>
    </depositor>
</bank>
```

XML è oggi largamente usato poiché lo scambio di dati è fondamentale allo stato del network attuale, come per esempio nell'attività bancaria (per il trasferimento di fondi), nell'elaborazione ordini (in particolare ordini interaziendali), per la gestione di dati scientifici (In chimica ChemML e altri, in genetica BSML (Bio-Sequence Markup Language) ed altri…) ecc…

↳ Il flusso cartaceo di informazioni tra le organizzazioni viene così sostituito dal flusso elettronico di informazioni. Ogni area applicativa ha una propria serie di standard per la rappresentazione delle informazioni ma XML è diventato la base per tutti i formati di interscambio di dati di nuova generazione.

I formati di interscambio di generazione precedente erano basati su ***plain text***, che aveva un concetto di fondo simile alle intestazioni delle e-mail, non consentiva strutture nidificate, né nessun linguaggio di tipo "standard" ed era legato troppo strettamente alla struttura del documento di basso livello (linee, spazi, ecc.)

Ogni standard basato su XML definisce quali sono elementi validi, usando

- *XML type specification languages* per specificare la sintassi
  - *DTD (Document Type Descriptors)*
  - *XML Schema*
- Descrizioni aggiuntive della semantica

↳ XML consente di definire nuovi tag come richiesto, tuttavia, questo può essere limitato dai DTD.

Una vasta gamma di strumenti è disponibile per l'analisi (*parsing*), la navigazione (*browsing*) e l'interrogazione (*querying*) di documenti/dati XML.

1.1 CONFRONTO CON I DB RELAZIONALI

XML è meno efficiente rispetto ai DB relazionali, infatti i *tag*, che in effetti rappresentano le informazioni sullo schema, vengono ripetuti.

XML è però migliore delle tuple dei DB relazionali come formato di scambio di dati, poichè:

- A differenza delle tuple relazionali, i dati XML sono auto-documentanti a causa della presenza di tag
- Formato non rigido: i tag possono essere aggiunti
- Permette strutture annidate
- Ampia accettazione, non solo nei *DB systems*, ma anche in browser/*tools*/*applications*.

2 STRUTTURA DELL'XML

Tag: etichetta per una sezione di dati

Element: sezione di dati che inizia con `<tagname>` e termina con la corrispondenza `</ tagname>`

Gli elementi devono essere nidificati correttamente:

- Nesting appropriato: `<account> … <balance> …. </balance> </account>`
- Nesting improprio: ~~`<account> … <balance> …. </account> </balance>`~~
- Formalmente: ogni tag di inizio deve avere un tag di fine in corrispondenza univoca, ovvero nel contesto dello stesso elemento padre.

Ogni documento deve avere un singolo elemento di livello superiore (*top-level element*)

Vediamo un esempio di XML nidificato:

```
<bank-1>
    <customer>
        <customer_name>  Hayes      </customer_name>
        <customer_street> Main      </customer_street>
        <customer_city>   Harrison  </customer_city>
        <account>
            <account_number> A-102     </account_number>
            <branch_name>    Perryridge </branch_name>
            <balance>        400       </balance>
        </account>
        <account>
        ...
        </account>
    </customer>
    .
    .
</bank-1>
```

L'annidamento (*nesting*) è utile per il trasferimento dei dati (ad esempio: gli elementi che rappresentano *customer_id*, *customer_name* e *address* nidificati all'interno di un elemento di *order*)

Il *nesting* non è supportato o ne è scoraggiato l'uso nei database relazionali. Vediamo perché in riferimento all'esempio precedente:

- Con più *order*, il *customer_name* e *address* vengono memorizzati in modo ridondante
- La normalizzazione sostituisce le strutture nidificate in ogni *order* con una *foreign key* nella tabella che memorizza il *customer_name* e *address*

↳ il Nesting è però supportato nei database relazionali agli oggetti

Non è appropriato eseguire il nesting durante il trasferimento dei dati: l'*external application* non ha accesso diretto ai dati a cui fa riferimento una *foreign key*.

In XML è permesso mischiare testo con sotto-elementi, ad esempio:

```
<account>
    This account is seldom used any more.
     <account_number> A-102</account_number>
     <branch_name> Perryridge</branch_name>
     <balance>400 </balance>
</account>
```

↳ ciò è utile per il ***markup***[1] del documento, ma non è consigliato per la rappresentazione dei dati.

2.1 ATTRIBUTI

• Gli *element* possono avere ***attributes***:

```
<account acct-type = "checking" >
    <account_number> A-102 </account_number>
    <branch_name> Perryridge </branch_name>
    <balance> 400 </balance>
</account>
```

Gli attributi sono specificati da coppie `nome = valore` nel tag iniziale di un *element*. Un *element* può avere diversi attributi, per ognuno si necessita di un nome unico.

```
<account  acct-type = "checking"  monthly-fee="5">
```

2.2 ATTRIBUTI VS. SOTTO-ELEMENTI

Esiste distinzioni tra ***subelement*** (sotto-elementi) e attributi:

- Nel contesto dei documenti gli attributi fanno parte del markup, mentre i contenuti del *subelement* fanno parte del contenuto del documento di base
- Nel contesto della rappresentazione dei dati, la differenza non è chiara e potrebbe essere fonte di confusione, poiché le stesse informazioni possono essere rappresentate in due modi:

```
– <account  account_number = "A-101">  .... </account>
– <account>
     <account_number>A-101</account_number> ...
   </account>
```

Suggerimento: utilizzare gli attributi come identificatori degli elementi e utilizzare i sotto-elementi invece per i contenuti.

2.3 NAMESPACE

I dati XML devono essere scambiati tra organizzazioni. Lo stesso nome del tag può avere un significato diverso nelle diverse organizzazioni, causando confusione sui documenti scambiati → specificare una stringa come nome univoco di un elemento ne evita la confusione. Si usa `unique-name:element-name`.

---

[1] Definiamo ***markup*** ogni mezzo per rendere esplicita una particolare interpretazione di un testo. Per esempio, tutte quelle aggiunte al testo scritto che permettono di renderlo più fruibile. Oltre a rendere il testo più leggibile, il markup permette anche di specificare ulteriori usi del testo. Con il markup per sistemi informatici (il nostro caso), specifichiamo le modalità esatte di utilizzo del testo nel sistema stesso.

Il markup non è soltanto un inevitabile e sgradevole risultato della informatizzazione dell'arte tipografica. Non è qualcosa che sta con noi a causa dell'informatica. Quando un autore scrive, da millenni a questa parte, specifica anche i delimitatori di parola (chiamati spazi), i delimitatori di frase (chiamati virgole) e i delimitatori di periodo (chiamati punti). La numerazione delle pagine o l'uso dei margini per creare effetti sul contenuto sono noti da centinaia di anni. Eppure questo a stretto rigore non fa parte del testo, ma del markup: nessuno dirà ad alta voce 'virgola' o 'punto' nel leggere un testo, ma creerà adeguati comportamenti paralinguistici (espressioni, toni, pause) per migliorare in chi ascolta la comprensione del testo. (- Fabio Vitali)

Bisogna però evitare di utilizzare nomi univoci lunghi e sparsi su tutto il documento: si utilizzano gli ***XML Namespaces*** (`Xmlns`):

```
<bank Xmlns:FB='http://www.FirstBank.com'>
  ...
  <FB:branch>
      <FB:branchname>Downtown</FB:branchname>
      <FB:branchcity>   Brooklyn   </FB:branchcity>
  </FB:branch>
  ...
</bank>
```

2.4 ULTERIORI NOTE SULLA SINTASSI XML

Gli elementi senza sotto-elementi ed i text-contents possono essere abbreviati terminando il tag di inizio con un `/>` ed eliminando il tag di fine:

```
<account  number="A-101" branch="Perryridge"  balance="200 />
```

Per memorizzare le *string* che possono contenere tag, senza che i tag vengano interpretati come sotto-elementi, si deve utilizzare `CDATA` ("*character data*") come di seguito:

```
<![CDATA[<account> … </account>]]>
```

Qui, `<account>` e `</account>` sono trattati come semplici stringhe.

3 XML DOCUMENT SCHEMA

I *Database schemas* vincolano le informazioni che possono essere memorizzate e i tipi di dati dei valori memorizzati. I documenti XML non richiedono un *associated schema*. Tuttavia, gli *schemas* sono molto importanti per lo scambio di dati XML. In caso contrario infatti, un sito non può interpretare automaticamente i dati ricevuti da un altro sito.

Esistono due meccanismi per specificare gli *XML schemas*:

- ***Document Type Definition*** **(DTD)** (Ampiamente usato)
- ***Schema XML*** (Uso più recente e crescente)

3.1 DOCUMENT TYPE DEFINITION (DTD)

Il tipo di un documento XML può essere specificato utilizzando un DTD

La struttura dei vincoli DTD per i dati in XML specifica:

- Quali elementi possono esserci
- Quali attributi possono/devono avere un elemento
- Quali sotto-elementi possono/devono esserci all'interno di ciascun elemento e quante volte.

DTD non vincola i tipi di dati! Tutti i valori sono infatti rappresentati come stringhe in XML.

Sintassi DTD:

- `<!ELEMENT element (subelements-specification) >`
- `<!ATTLIST element (attributes) >`

I sotto-elementi possono essere specificati come:

- Nomi di elementi
- #PCDATA (*parsed character data*) cioè stringhe di caratteri
- EMPTY (nessun sottoelemento) o ANY (qualsiasi cosa può essere un sottoelemento)

Ad esempio:

```
<! ELEMENT depositor (customer_name  account_number)>
<! ELEMENT customer_name (#PCDATA)>
<! ELEMENT account_number (#PCDATA)>
```

Le *subelement specifications* possono utilizzareespressioni regolari come:

```
<!ELEMENT bank ( ( account | customer | depositor)+)>
```

Notazione: "|": separa le alternative, "+": 1 o più occorrenze, "*": 0 o più occorrenze.

Esempio DTD per una banca (1):

```
<!DOCTYPE bank [
    <!ELEMENT bank ( ( account | customer | depositor)+)>
    <!ELEMENT account (account_number branch_name balance)>
    <! ELEMENT customer(customer_name customer_street
                                        customer_city)>
    <! ELEMENT depositor (customer_name account_number)>
    <! ELEMENT account_number (#PCDATA)>
    <! ELEMENT branch_name (#PCDATA)>|
    <! ELEMENT balance(#PCDATA)>
    <! ELEMENT customer_name(#PCDATA)>
    <! ELEMENT customer_street(#PCDATA)>
    <! ELEMENT customer_city(#PCDATA)>
]>
```

3.2 Attribute Specification in DTD

Per la *attribute specification* bisogna fornire, per ogni attributo:

- Nome
- Tipo di attributo: CDATA, ID o IDREF ("*reference ID*") o IDREFS (più IDREF) (v. dopo)
- Specificarne l'obbligatorietà con `#REQUIRED` (e quindi il valore predefinito) o la non obbligatorietà con `#IMPLIED`

Vediamo alcuni esempi:

```
<!ATTLIST account  acct-type CDATA "checking">
<!ATTLIST customer
   customer_id   ID        # REQUIRED
   accounts      IDREFS # REQUIRED   >
```

3.3 ID e IDREFs in DTDs

Un elemento può avere al massimo un attributo di tipo **ID**. Il valore dell'attributo ID di ciascun elemento in un documento XML deve essere distinto, quindi (ovviamente) il valore dell'attributo ID è un *object identifier*.

Un attributo di tipo **IDREF** deve contenere il valore ID di un elemento nello stesso documento

Un attributo di tipo **IDREFS** contiene un insieme di valori ID (0 o più). Ogni valore ID deve contenere il valore ID di un elemento nello stesso documento

Esempio DTD per una banca (2) (con attributi):

```
<!DOCTYPE bank-2[
   <!ELEMENT account (branch, balance)>
   <!ATTLIST account
         account_number ID        # REQUIRED
         owners           IDREFS # REQUIRED>
   <!ELEMENT customer(customer_name, customer_street,
                                          customer_city)>
   <!ATTLIST customer
         customer_id      ID       # REQUIRED
         accounts         IDREFS # REQUIRED>
   … declarations for branch, balance, customer_name,
                           customer_street and customer_city
]>
```

Esempio di dati XML con attributi:

```
<bank-2>
      <account account_number="A-401" owners="C100 C102">
            <branch_name> Downtown </branch_name>
            <balance>        500 </balance>
      </account>
…..
      <customer customer_id="C100" accounts="A-401">
            <customer_name>Joe        </customer_name>
            <customer_street> Monroe  </customer_street>
            <customer_city>     Madison</customer_city>
      </customer>
      <customer customer_id="C102" accounts="A-401 A-402">
            <customer_name> Mary      </customer_name>
            <customer_street> Erin      </customer_street>
            <customer_city>     Newark </customer_city>
      </customer>
</bank-2>
```

3.4 LIMITI DEI DTDS

Non c'è *typing* ("tipizzazione", distinzione tra tipi diversi) tra elementi di testo e attributi (tutti i valori sono stringhe, non interi/reali/ecc…)

È difficile specificare insiemi non ordinati di sotto-elementi

- L'ordine è in genere irrilevante nei database (diversamente dall'ambiente di layout del documento da cui si è evoluto l'XML)
- $(A|B)*$ consente di specificare un set non ordinato, ma non è possibile garantire che A e B si verifichino ognuna una sola volta.

ID e IDREF sono un*typed* (non tipizzati): ad esempio l'attributo *owners* di un *account* può contenere un riferimento a un altro *account*, che è privo di significato, mentre l'attributo *owner* dovrebbe riferirsi idealmente agli elementi di *customer*.

3.5 XML SCHEMA

XML Schema è uno *schema language* più sofisticato di DTD, che ne risolve gli svantaggi. Supporta infatti:

- La tipizzazione dei valori (*integer*, *string*, …) vincolandone anche dei valori min/max

- Tipi complessi definiti dall'utente (*user-defined, complex types*)
- Molte altre funzionalità tra cui: vincoli di unicità e vincoli di *foreign key*, ereditarietà…

Lo *XML Schema* è a sua volta specificato in sintassi XML, diversamente dai DTD: ha quindi una rappresentazione più standard ma sovrabbondante, verbosa.

XML Schema è inoltre integrato con i *namespace*.

Problema: *XML Schema* è significativamente più complicato rispetto ai DTD.

Esempio DTD per una banca in versione *XML Schema*:

```
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema>
<xs:element name="bank" type="BankType"/>
<xs:element name="account">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="account_number" type="xs:string"/>
            <xs:element name="branch_name"    type="xs:string"/>
            <xs:element name="balance"        type="xs:decimal"/>
        </xs:squence>
    </xs:complexType>
</xs:element>
….. definitions of customer and depositor ….
<xs:complexType name="BankType">
    <xs:squence>
        <xs:element ref="account"   minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="customer" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="depositor" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

Nota Bene:

- La scelta di "`xs:`" è stata nostra , qualsiasi altro prefisso del *namespace* poteva essere scelto
- L'elemento "*bank*" ha tipo "*BankType*", che è definito separatamente. `xs:complexType` viene utilizzato in seguito per creare il *complex type* "*BankType*"
- L'elemento "*account*" ha il suo tipo definito in linea

3.6 ALTRE FUNZIONALITÀ DI XML SCHEMA

Vediamo alcune altre funzionalità di *XML Schema*:

- Attributi specificati dal tag `xs:attribute`:
  - `<xs:attribute name="account_number"/>`
  - aggiungere l'attributo `use = "required"` significa che il valore deve essere specificato
- *Key constraints*
  - "Gli elementi *account_number* sono chiavi gli elementi di *account* sotto l'elemento *root bank*:

```
<xs:key name = "accountKey">
        <xs:selector xpath = "/ bank/account"/>
        <xs:field xpath = "account_number"/>
<\xs:key>
```

- *Foreign key constraint* da *depositor* a *account:*

```
<xs:keyref name = "depositorAccountKey" refer="accountKey">
        <xs:selector xpath = "/bank/depositor "/>
        <xs:field xpath = "account_number"/>
<\xs:keyref>
```

4 QUERY E TRADUZIONE DATI IN XML

La traduzione di informazioni da un *XML schema* ad un altro e le query su dati XML sono due operazioni strettamente correlate e gestite dagli stessi strumenti → vediamo le lingue di interrogazione/traduzione XML standard:

- XPath: linguaggio semplice composto da espressioni di percorso
- XQuery: un linguaggio di query XML con un ricco set di funzionalità
- XSLT: linguaggio semplice progettato per la traduzione da XML a XML e da XML a HTML

4.1 MODELLO AD ALBERO PER I DATI XML

I linguaggi di query/traduzione sono basati su un ***tree model*** (modello ad albero) per i dati XML: si modella il documento XML come un albero, con i nodi corrispondenti a elementi e attributi:

- I nodi elemento hanno nodi figli, che possono essere attributi o sotto-elementi
- Il testo in un elemento è modellato come un nodo di testo figlio dell'elemento
- I figli di un nodo vengono ordinati in base al loro ordine nel documento XML
- I nodi elemento e attributo (ad eccezione del nodo radice) hanno un singolo genitore, che è un nodo elemento
- Il nodo radice ha un singolo figlio, che è l'elemento principale del documento

4.2 XPATH

XPath è usato per indirizzare (selezionare) parti di documenti usando ***path expressions***. Un'espressione di percorso è rappresentata come una sequenza di passaggi separati da **"/"** (come i nomi dei file in una gerarchia di directory). Il risultato della *path expression* è l'insieme di valori, assieme ai relativi elementi/attributi in cui sono contenuti, che corrispondono al percorso specificato.

Ad esempio:

/bank-2/customer/customer_name

In riferimento al documento XML "*bank-2*" visto prima, ci restituisce:

<customer_name>Joe</customer_name>
<customer_name>Mary</customer_name>

Mentre:

/bank-2/customer/customer_name/text( )

Restituisce gli stessi nomi del primo esempio, ma senza i tag che li contengono.

L'iniziale "/" indica la radice (*root*) del documento (davanti al tag di livello superiore)

Le *path expressions* vengono valutate "leggendole" da sinistra a destra: ogni passaggio opera sull'insieme di istanze prodotte dal passaggio precedente.

Le "*select*" possono esserci in qualsiasi passaggio di un percorso, e si indicano con due parentesi quadre **[ ]**:

Ad esempio:

/bank-2/account[balance > 400]

restituisce elementi di *account* con un *balance* superiore a 400, mentre:

/bank-2/account[balance]

restituisce elementi di *account* contenenti un sotto-elemento *balance*.

Si accede agli attributi usando **"@"**, ad es. per ottenere gli *account_number* degli *account* con *balance*> 400:

/bank-2/account[balance > 400]/@account_number

Gli attributi IDREF non vengono automaticamente de-referenziati (vedremo maggiori dettagli dopo)

4.3 Funzioni in XPath

XPath offre diverse funzioni, vediamone alcune:

- La funzione `count()` alla fine di un *path* conta il numero di elementi nel set generato dal *path*
  Per esempio `/bank-2/account[count(./customer)>2]` restituisce gli *account* con > 2 clienti
  - `count()` funziona anche per trovare la posizione (1, 2, ..) di un nodo rispetto ai nodi fratelli
- I connettivi booleani `and`, `or` e la funzione `not()` possono essere utilizzati nei predicati
- Gli IDREF possono essere referenziati usando la funzione `id()`
  - `id()` può anche essere applicato a set di *references* come IDREFS e persino a stringhe contenenti più *references* separati da spazi vuoti, ad esempio:
    `/bank-2/account/id(@owner)` restituisce tutti i *customers* a cui si fa riferimento dall'attributo *owner* degli elementi di *account*.
- L'operatore `"|"` è utilizzato per implementare l'**unione**, ad esempio:
  - `/bank-2/account/id(@owner) | /bank-2/loan/id(@borrower)`
    Fornisce i *customers* aventi sia *account* sia *loan*.
    Attenzione: `"|"` non può essere annidato all'interno di altri operatori.
- L'operatore `"//"` può essere usato per saltare più livelli di nodi
  - `/bank-2//customer_name` trova qualsiasi elemento *customer_name* ovunque sotto l'elemento */bank-2* indipendentemente dall'elemento in cui è contenuto.
- Un passo nel *path* può puntare verso nodi genitori, fratelli, antenati e discendenti dei nodi generati dal passaggio precedente, non solo ai nodi figli:
  - Il `"//"` sopra descritto è una forma abbreviata per dire "tutti i discendenti"
  - Il `".."` è usato per dire "il nodo genitore".
- `doc(name)` restituisce la radice del documento con il nome specificato

4.4 XQuery

XQuery è un linguaggio di query *general-purpose* per gestire i dati XML. Attualmente è standardizzato dal World Wide Web Consortium (W3C). La descrizione che affronteremo è basata su una bozza dello standard del gennaio 2005. La versione finale potrebbe essere diversa, ma le funzionalità principali probabilmente rimarranno invariate.

XQuery è derivato dal linguaggio query Quilt, che a sua volta "si ispira" a SQL, XQL e XML-QL

XQuery utilizza la sintassi FLOWR (For, Let, Where, Order by, Return):
- **`for`** ⇔ SQL from
- **`let`** consente variabili temporanee, e non ha equivalenti in SQL
- **`order by`** ⇔ SQL order by
- **`where`** ⇔ SQL where
- **`result`** ⇔ SQL select

La "`for`" *clause* utilizza espressioni XPath, e le variabile nella "`for`" *clause* assumono i valori del set restituito da XPath

Vediamo una espressione FLWOR semplici in XQuery: "trova tutti gli *account* con *balance* > 400, con ogni risultato racchiuso in un tag `<account_number> .. </ account_number >`". La possiamo implementare con la seguente:

```
for     $x in /bank-2/account
let     $acctno := $x/@account_number
where $x/balance > 400
return <account_number> { $acctno } </account_number>
```

↳ Gli elementi nella "`return`" *clause* sono testo XML a meno che non siano racchiusi in , nel qual caso vengono valutati.

La "`Let`" *clause* non è realmente necessaria nella query appena vista, e la *select* può essere eseguita anche in XPath con le parentesi quadre `[]`. La query può essere scritta come:

```
for $x in /bank-2/account[balance>400]
return <account_number> { $x/@account_number }
          </account_number>
```

4.5 Join in XQuery

I join sono specificati in modo molto simile a SQL:

```
for $a  in  /bank/account,
     $c  in  /bank/customer,
     $d  in  /bank/depositor
where   $a/account_number = $d/account_number
     and $c/customer_name = $d/customer_name
return <cust_acct> { $c $a } </cust_acct>
```

La stessa query può essere espressa con le selezioni specificate come selezioni XPath (quindi con le parentesi quadrate):

```
for $a in /bank/account
     $c in /bank/customer
      $d in /bank/depositor[
               account_number = $a/account_number and
               customer_name  = $c/customer_name]
return <cust_acct> { $c $a } </cust_acct>
```

4.6 Query annidate in XQuery

La seguente query converte i dati dalla struttura *flat* per le informazioni usata in *bank* nella struttura nidificata utilizzata in *bank-1*:

```
<bank-1> {
    for $c in /bank/customer
    return
       <customer>
         { $c/* }
         { for $d in /bank/depositor[customer_name = $c/customer_name],
              $a in /bank/account[account_number=$d/account_number]
         return $a }
       </customer>
} </bank-1>
```

- `$c/*` indica tutti i figli del nodo a cui `$c` è associato, senza il tag di livello superiore che lo racchiude
- `$c/text()` fornisce il contenuto del testo di un elemento senza alcun sotto-elementi o tag

4.7 Sorting in XQuery

La clausola "`order by`" *clause* può essere utilizzata alla fine di qualsiasi espressione. Per esempio, per ottenere i *customers* ordinati per nome:

```
for $c in /bank/customer
order by $c/customer_name
return <customer> { $c/* } </customer>
```

↳ si può usare `order by $c/customer_name` per ordinare in ordine decrescente.

Si possono annidare gli ordini in più livelli, ad esempio ordinare per *customer_name* e per *account_number* ogni *customer:*

```
<bank-1> {
    for $c in /bank/customer
    order by $c/customer_name
    return
        <customer>
            { $c/* }
            { for $d in /bank/depositor[customer_name=$c/customer_name],
                    $a in /bank/account[account_number=$d/account_number]
            order by $a/account_number
            return <account> $a/* </account>}

        </customer>
} </bank-1>
```

4.8 Altre Funzionalità di XQuery

Le funzioni possono essere definite dall'utente con *type system* di *XMLSchema*:

```
function balances(xs:string $c) returns list(xs:decimal*) {
  for $d in /bank/depositor[customer_name = $c],
      $a in /bank/account[account_number = $d/account_number]
  return $a/balance
}
```

↳ i *type* sono opzionali per i parametri delle funzioni e per i valori di *return* di esse.

Il simbolo `*` (nell'esempio vediamo `decimal*`) indica una sequenza di valori di quel *type*.

Vediamo come usare il quantificatore universale (∀) ed esistenziale (∃) nei predicati di una *where clause:*

some $e **in** *path* **satisfies** *P*
every $e **in** *path* **satisfies** *P*

XQuery supporta anche *If-then-else clause*s.

4.9 XSLT

Uno ***stylesheet*** ("foglio di stile") memorizza le opzioni di formattazione per un documento, di solito separatamente rispetto al documento stesso. Ad esempio: uno stylesheet HTML può specificare i colori e le dimensioni dei caratteri per le intestazioni, ecc…

XML Stylesheet Language (**XSL**) è stato originariamente progettato per generare HTML da XML. **XSLT** invece è un linguaggio di trasformazione *general-purpose*, può tradurre XML in XML e XML in HTML.

Le trasformazioni XSLT sono espresse usando regole chiamate ***templates*** (modelli), che combinano la selezione di XPath con la costruzione dei risultati.

Vediamo un esempio di modello XSLT con `match` e `select`:

```
<xsl:template match="/bank-2/customer">
    <xsl:value-of select="customer_name"/>
</xsl:template>
<xsl:template match="*"/>
```

- L'attributo `match` di `xsl:template` specifica un pattern in XPath
  ↳ gli elementi nel documento XML che corrispondono al pattern specificato vengono elaborati dalle azioni contenute nell'elemento `xsl:template`
- `xsl:value-of` seleziona (in output) valori specificati (nell'esempio `customer_name`)
- Per elementi che non corrispondono a nessun *template* gli attributi e il contenuto del testo vengono restituiti in output "così come sono". I template sono applicati poi ricorsivamente sui sotto-elementi
- Il template `<xsl:template match="*"/>` corrisponde a tutti gli elementi che non corrispondono a nessun altro modello, è utilizzato per garantire che i loro contenuti non vengano restituiti in output.
- Se un elemento corrisponde (*match*) a più *templates*, viene utilizzato solo da uno, e la decisione è presa basandosi su uno complesso schema di priorità oppure sulle priorità definite dall'utente.

4.10 Creazione di output XML con XSLT

Qualsiasi testo o tag nello *stylesheet* XSL che non è nell'*XSL namespace* viene visualizzato in output "così com'è". Ad esempio per incapsulare/inserire (*wrap*) i risultati in nuovi elementi XML:

```
<xsl:template match="/bank-2/customer">
    <customer>
    <xsl:value-of select="customer_name"/>
    </customer>
</xsl;template>
<xsl:template match="*"/>
Example output:
    <customer> Joe   </customer>
    <customer> Mary </customer>
```

Nota: non è possibile inserire direttamente un tag `xsl:value-of` all'interno di un altro tag. Ad esempio, è impossibile creare un attributo per il `<customer>` dell'esempio precedente utilizzando direttamente `xsl:value-of`. XSLT fornisce un costrutto **`xsl:attribute`** per gestire questa situazione, che aggiunge l'attributo all'elemento precedente:

```
E.g. <customer>
        <xsl:attribute name="customer_id">
            <xsl:value-of select = "customer_id"/>
        </xsl:attribute>
      </customer>
results in output of the form
        <customer  customer_id="....."> ....
```

`xsl:element` è usato per creare elementi in output con nomi che siano calcolati

4.11 RICORSIONE STRUTTURALE CON XSLT

L'azione di un *template* può essere l'applicazione ricorsiva di *templates* ai contenuti degli elementi corrispondenti (*matching elements*). Ad esempio:

```
<xsl:template match="/bank">
    <customers>
    <xsl:template apply-templates/>
    </customers >
</xsl:template>
<xsl:template match="/customer">
    <customer>
    <xsl:value-of select="customer_name"/>
    </customer>
</xsl:template>
<xsl:template match="*"/>
```

```
Example output:
  <customers>
    <customer> John </customer>
    <customer> Mary </customer>
  </customers>
```

4.12 JOIN IN XSLT

Le ***XSLT keys*** consentono di cercare gli elementi (indicizzati) in base ai valori di sotto-elementi o attributi. Queste chiavi devono essere dichiarate (con un nome) ed allora la funzione `key()` può essere utilizzata per la ricerca. Ad esempio:

```
<xsl:key name="acctno" match="account"
                      use="account_number"/>
<xsl:value-of select=key("acctno", "A-101")
```

Le chiavi consentono che (alcuni) join possano essere espressi in XSLT:

```
<xsl:key name="acctno" match="account" use="account_number"/>
<xsl:key name="custno" match="customer" use="customer_name"/>
<xsl:template match="depositor">
        <cust_acct>
        <xsl:value-of select=key("custno", "customer_name")/>
        <xsl:value-of select=key("acctno", "account_number")/>
        </cust_acct>
</xsl:template>
<xsl:template match="*"/>
```

4.13 SORTING IN XSLT

L'uso di `xsl:sort` all'interno di un *template* fa sì che tutti gli elementi che corrispondono (*match*) al *template* siano ordinati. L'ordinamento viene eseguito prima di applicare altri *templates*:

```
<xsl:template match="/bank">
    <xsl:apply-templates select="customer">
    <xsl:sort select="customer_name"/>
    </xsl:apply-templates>
</xsl:template>
<xsl:template match="customer">
    <customer>
        <xsl:value-of select="customer_name"/>
        <xsl:value-of select="customer_street"/>
        <xsl:value-of select="customer_city"/>
    </customer>
<xsl:template>
<xsl:template match="*"/>
```

5 Application Program Interface

Ci sono due standard APIs per gli XML data:

- **SAX** (*Simple API for XML*)
  - Basato sul *parser model.*
  - L'utente fornisce gli *event handlers* ("gestori di eventi") per i *parsing events* ("eventi di analisi")
    Ad esempio i metodi `startElement` ed `endElement`
  - Non è adatto per *database applications*
- **DOM** (*Document Object Model*)
  - I dati XML vengono analizzati in una *tree representation*
  - Vengono fornite varie funzioni per attraversare il **DOM tree**
  - Ad esempio: l'API DOM Java fornisce la classe `Node` con i metodi `getParentNode()`, `getFirstChild()`, `getNextSibling()`, `getAttribute()`, `getData()` (per il nodo di testo), `getElementsByTagName()`, ...
  - Fornisce anche funzioni per l'aggiornamento dell'albero DOM

6 Archiviazione di dati XML

I dati XML possono essere archiviati in

- Archivi di dati non relazionali
  - ***Flat files***
    - Supporto naturale per la memorizzazione di XML
    - Ma ha tutti i problemi discussi nel Capitolo 1 (no *concurrency*, no *recovery*, …)
  - ***XML databases***
    - Database creato appositamente per la memorizzazione di dati XML, che supporta il modello DOM e le query dichiarative
    - Attualmente nessun sistema di livello commerciale
- Database relazionali
  - I dati devono essere tradotti in forma relazionale
  - Vantaggio: i sistemi di database sono sistemi "maturi"
  - Svantaggi: possibile sovraccarico nella traduzione di dati e query

↳ Tre possibili alternative:

1. ***String Representation***
2. ***Tree Representation***
3. ***Map to relations***

Vedremo più dettagliatamente, nel seguito, come funzionano.

6.1 Database relazionali: *String representation*

Memorizza ciascun elemento top-level come campo *string* di una tupla in un database relazionale.

Usa una singola relazione per memorizzare tutti gli elementi, oppure una relazione separata per ogni tipo di elemento top-level (ad esempio: *account, customer, depositor relations* ciascuno con attributo *string-valued* per memorizzare l'elemento)

Indicizzazione:

- Memorizza i valori di sott-oelementi/attributi da indicizzare come campi aggiuntivi della relazione e crea indici su questi campi (ad esempio *customer_name* o *account_number*)
- Alcuni sistemi di database supportano ***function indices***, che utilizzano il risultato di una funzione come *key value*. La funzione dovrebbe restituire il valore del sotto-elemento/attributo richiesto

Benefici di questo tipo di rappresentazione:

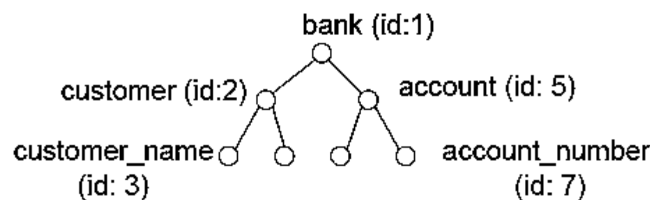
- Può memorizzare qualsiasi dato XML anche senza DTD
- Finché ci sono molti elementi top-level in un documento, le stringhe sono piccole rispetto al documento completo → Consente l'accesso rapido ai singoli elementi.

Svantaggio di questo tipo di rappresentazione: è necessario analizzare le stringhe per accedere ai valori all'interno degli elementi → l'analisi (*parsing*) è lenta.

6.2 DATABASE RELAZIONALI: *TREE REPRESENTATION*

La rappresentazione ad albero modella i dati XML come un albero e li archivia usando le relazioni:

- `nodes(id,type,label,value)`
- `child(child_id,parent_id)`



Vediamone i campi nel dettaglio:

- `id:` ad ogni elemento/attributo viene assegnato un identificatore univoco
- `type:` specifica se elemento/attributo
- `label:` specifica il nome del tag dell'elemento/nome dell'attributo
- `value:` è il valore del testo dell'elemento/attributo
- La relazione `child` rileva le relazioni genitore-figlio nell'albero. Si può aggiungere un attributo extra a `child` per registrare l'ordine dei figli.

Vantaggio di questo tipo di rappresentazione: può memorizzare qualsiasi dato XML, anche senza DTD

Svantaggi di questo tipo di rappresentazione:

- I dati possono venir suddivisi in troppi pezzi, aumentando i costi generali in termini di spazio
- Anche le query semplici richiedono un numero elevato di join, che può essere lento

6.3 DATABASE RELAZIONALI: *MAPPING XML DATA TO RELATIONS*

Relazioni create per ogni tipo di elemento il cui *schema* è noto:

- Un attributo *id* per memorizzare un ID univoco per ogni elemento
- Un attributo *relation* corrispondente ad ogni attributo dell'elemento
- Un attributo *parent_id* per tenere traccia dell'elemento "genitore" (come nella rappresentazione ad albero)
- È possibile memorizzare anche le informazioni sul posizionamento (es. i-esimo "figlio")

Tutti i sotto-elementi che occorrono una sola volta possono diventare attributi di relazione

- Per i sotto-elementi testuali (*text-valued*) si memorizza il testo come *attribute value*
- Per i sotto-elementi complessi si può memorizzare l'ID del sotto-elemento

I sotto-elementi che possono occorrere più volte sono rappresentati in una tabella separata (come per la gestione degli attributi multivalore nella conversione da diagrammi E-R a tabelle)

6.4 PUBLISHING AND SHREDDING DEI DATI XML

Vediamo quindi alcune caratteristiche e alcune definizioni della memorizzazione di dati XML nei database relazionali:

- ***Publishing***: processo di conversione dei dati relazionali in un formato XML
- ***Shredding***: processo di conversione di un documento XML in una serie di tuple da inserire in una o più relazioni
  - ↳I sistemi di database *XML-enabled* supportano *publishing* e *shredding* automatici

6.5 STORAGE NATIVO DI DATI XML IN UN DB RELAZIONALE

Alcuni database relazionali supportano l'archiviazione nativa (***native storage***) di XML. Tali sistemi memorizzano i dati XML come stringhe o come rappresentazioni binarie più efficienti, senza convertire i dati in forma relazionale. Vediamo alcune caratterisriche:

- Viene introdotto il nuovo *data type* **`xml`** per rappresentare i dati XML.
- Sono supportati i linguaggi di query XML come XPath e XQuery le per query sui dati XML.
- Una relazione con un attributo di tipo **`xml`** può essere utilizzata per memorizzare una *collection* di documenti XML; ogni documento è memorizzato come un valore di tipo **`xml`** in una tupla separata.
- Vengono creati indici speciali (*special indices*) per indicizzare i dati XML.

7 SQL / XML

Mentre XML è ampiamente utilizzato per lo scambio di dati, gli *structured data* sono ancora ampiamente memorizzati in database relazionali. Spesso è necessario convertire i dati relazionali in rappresentazione XML. Lo standard SQL/XML, sviluppato per soddisfare questa esigenza, definisce un'estensione standard di SQL, che consente la creazione di output XML nidificato.

Lo standard ha diverse parti, tra cui:

1. un modo standard per *map* (associare) gli *SQL types* agli *XMLSchema types*
2. un modo standard per *map* (associare) i *relational schemas* agli *XML schemas*
3. estensioni di linguaggio per le query SQL

Ad esempio, la rappresentazione SQL/XML `bank` ha un *XML schema* avente come elemento più esterno `<bank>`. Ogni tupla è *mapped* (associata) ad un *XML element* del tipo `<row>`, e ogni attributo … della relazione è *mapped* a un elemento XML `<…>` con lo stesso nome (salvo alcune eccezioni dovute a convenzioni per risolvere le incompatibilità con i caratteri speciali che potrebbero essere presenti nei nomi). Un intero *SQL schema*, con più relazioni, può essere quindi *mapped* in XML in questo modo. Vediamo:

```
<bank>
  <account>
    <row>
        <account_number> A-101 </account_number>
        <branch_name> Downtown </branch_name>
        <balance> 500 </balance>
    </row>
  .... more rows if there are more output tuples ...
  </account>
</bank>
```

SQL/XML aggiunge diversi *operators* e *aggregate operations* a SQL per consentire la costruzione dell'output XML direttamente dall'SQL esteso. Vediamo ad esempio:

- **`xmlelement`** crea elementi XML
- **`xmlattributes`** crea attributi

Ad esempio:

```
select xmlelement (name "account",
       xmlattributes (account_number as account_number),
       xmlelement (name "branch_name", branch_name),
       xmlelement (name "balance", balance))
from account
```

- **`xmlforest(attr1, attr2, ..)`** crea una sequenza ("*forest*") di uno o più elementi, con i nomi dei tag identici ai nomi degli attributi in SQL

8 XML Application: Web Services

Lo standard **SOAP** (*Simple Object Access Protocol*) invoca procedure tra applicazioni con database distinti. XML è utilizzato per rappresentare l'input e l'output della procedura

Un ***web service*** è un sito che fornisce una raccolta di procedure SOAP. È descritto utilizzando il **WSDL** (*Web Services Description Language*). Le directory dei *web services* sono descritte utilizzando lo standard **UDDI** (*Universal Description, Discovery and Integration*).

Cap 11: Strutture di archiviazione e di file

1 SUPPORTI FISICI DI ARCHIVIAZIONE (*PHYSICAL STORAGE MEDIA*)

1.1 CRITERI DI CLASSIFICAZIONE

La classificazione è fatta in base a:

- Velocità con cui è possibile accedere ai dati
- Costo per unità di dati
- Affidabilità (*reliability*)
  - perdita di dati in caso di interruzione di corrente o arresto anomalo del sistema
  - guasto fisico del dispositivo di archiviazione

Possiamo differenziare tra:

- ***storage* volatile:** perde il contenuto quando l'alimentazione è spenta
- ***storage* non volatile**: i contenuti persistono anche quando l'alimentazione è spenta. Include memoria secondaria, terziaria ed anche le memorie principali con sistema di backup a batteria.

1.2 TIPOLOGIE

Vediamo i principali supporti fisici di archiviazione:

- **Cache**
  - È il tipo di storage più veloce e più costoso
  - **È volatile**
  - È gestito dall'hardware del sistema informatico

- **Memoria principale**
  - Accesso rapido (10-100 nanosec.)
  - Generalmente troppo piccolo (o troppo costoso) per memorizzare l'intero DB
  - Capacità fino a pochi GB al momento molto utilizzate
  - Con il passare del tempo le capacità sono aumentate ed i costi per byte sono diminuiti costantemente e rapidamente (circa 2 volte ogni 2 o 3 anni)
  - **È volatile** (i contenuti della memoria principale di solito vengono persi in caso di interruzione di corrente o arresto anomalo del sistema)

- **Memoria flash**
  - **È non volatile** (i dati sopravvivono all'interruzione di alimentazione elettrica)
  - I dati possono essere scritti in una posizione, la quale può successivamente essere cancellata e scritta nuovamente (può supportare solo un numero limitato (10K-1M) di cicli di scrittura/cancellazione e la cancellazione della memoria deve essere eseguita su un intero banco di memoria
  - In lettura è più o meno veloce quanto la memoria principale
  - In scrittura è lenta (qualche microsec.)
  - In cancellazione è ancora più lenta
  - Il costo per unità di memoria approssimativamente simile a quello della memoria principale
  - È un tipo di memoria **EEPROM** (*Electrically Erasable Programmable Read-Only Memory*)
  - NOR Flash (memorie flash realizzate con porte NOR)
    - Letture veloci, cancellazioni molto lente, capacità inferiori
    - Utilizzato per memorizzare il codice del programma in molti dispositivi incorporati
  - NAND Flash
    - Lettura/scrittura a una pagina per volta, cancellazione di più pagine
    - Alta capacità (diversi GB)

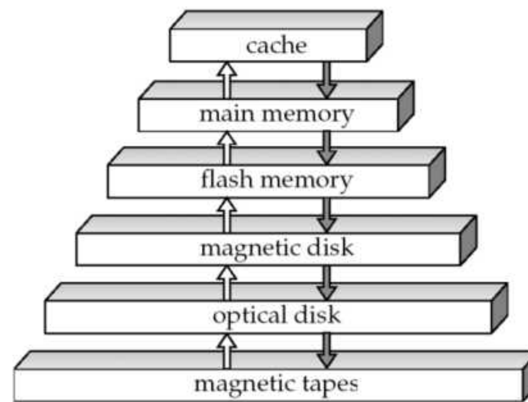
        - Ampiamente usata in dispositivi embedded come fotocamere digitali

- **Magnetic-disk**
    - I dati sono memorizzati su un disco rotante e sono letti/scritti magneticamente
    - È il supporto principale per la **memorizzazione a lungo termine** dei dati, in genere memorizza l'intero DB
    - I dati devono essere spostati dal disco alla memoria principale per l'accesso e riscritti per l'archiviazione
    - L'accesso è molto più lento rispetto a quello della memoria principale (vedremo maggiori dettagli in seguito)
    - **Accesso diretto**: è possibile leggere i dati sul disco in qualsiasi ordine, a differenza del nastro magnetico (che ha accesso sequenziale)
    - Le capacità variano fino a circa 400 GB al momento (nel 2005)
    - Capacità e costo per byte molto più elevati rispetto alla memoria principale/memoria flash
    - Sopravvive a interruzioni di corrente e arresti di sistema (una ***disk failure*** può distruggere i dati, ma è raro che accada)

- **Archiviazione ottica**
    - **Non volatile**, i dati vengono letti otticamente da un disco rotante utilizzando un laser
    - Possono essere:
        - CD-ROM (640 MB) e DVD (da 4,7 a 17 GB), i formati più comuni
        - Dischi ottici Write-One, Read-Many (WORM)
        - Dischi ottici utilizzati per l'archiviazione (CD-R, DVD-R, DVD + R)
        - Dischi ottici a scrittura multipla (CD-RW, DVD-RW, DVD + RW e DVD-RAM)
    - Le letture e le scritture sono più lente rispetto al disco magnetico
    - Si possono realizzare **sistemi juke-box**, con un numero elevato di dischi rimovibili, poche altre unità di storage ed un meccanismo per il caricamento automatico dei dischi disponibile per la memorizzazione di grandi volumi di dati
- **Archiviazione su nastro**
    - **Non volatile**, utilizzato principalmente per il backup (per il ripristino da un errore del disco) e per gli archivi
    - **Accesso sequenziale** - molto più lento del disco magnetico
    - Capacità molto elevata (disponibili nastri da 40 a 300 GB)
    - Il nastro può essere rimosso dall'unità ⇒ I costi di archiviazione sono molto più economici del disco magnetico, ma le unità sono costose
    - ***Tape jukebox*** disponibili per la memorizzazione di enormi quantità di dati (centinaia di TB fino a 1 PB[1])

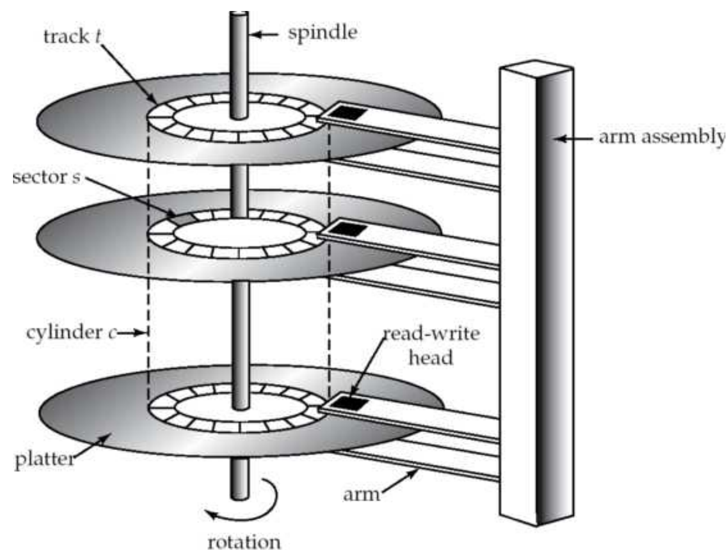
[1] Nota: 1 petabyte = 10^{12} byte

2 GERARCHIA DI ARCHIVIAZIONE



1. **Memoria principale:** *storage medium* (supporto di memorizzazione) più veloce ma volatile (cache, memoria principale).
2. **Memoria secondario:** livello successivo in gerarchia, accesso non volatile e moderatamente veloce (chiamato anche ***on-line storage***, ad es. Memoria flash, dischi magnetici…)
3. **Memoria terziaria:** livello più basso nella gerarchia, non volatile, tempo di accesso lento (chiamato anche ***off-line storage***, ad es. nastro magnetico, memoria ottica…)

3 HARD-DISK MAGNETICI



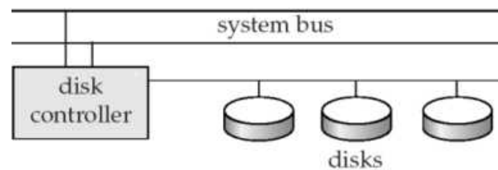
NOTA: il diagramma è schematico e semplifica la struttura dei *disk drive* reali

Un 2.1 hard-disk magnetico è composto da:

1. **Testina di lettura-scrittura**
   a. Posizionata molto vicino alla superficie del piatto (quasi a toccarlo)
   b. Legge o scrive informazioni codificate magneticamente.
2. La superficie del piatto divisa in ***circular tracks***, normalmente ve ne sono oltre 50K-100K per piatto
3. Ogni *track* è divisa in **settori**
   a. Un settore è la più piccola unità di dati che può essere letta o scritta.
   b. La dimensione del settore in genere 512 byte
   c. Settori tipici per track: 500 (su binari interni) a 1000 (su binari esterni)
4. Per <u>leggere/scrivere un settore</u>:
   a. Il braccio del disco oscilla per posizionare la testina sulla track desiderata
   b. Il piatto gira continuamente; i dati vengono letti/scritti mentre il settore è controllato

5. ***Head-disk assemblies***
    a. Più dischi su un singolo ***spindle*** (solitamente da 1 a 5)
    b. Una testa per piatto, montata su un braccio comune.
6. L'*i*-esimo **cilindro** è costituito dall *i-esima* traccia di tutti i piatti
7. I dischi di generazione precedente erano suscettibili agli "***head-crashes***":
    a. La superficie dei dischi di generazione precedente aveva rivestimenti di ossido di metallo che si disintegravano in caso di incidente alla testina e danneggiavano tutti i dati sul disco
    b. I dischi di generazione attuale sono meno suscettibili a tali *failures* disastrose, sebbene singoli settori possano risultarne corrotti
8. **Disk controller**: interfacce tra il sistema del computer e l'hardware dell'disco.
    a. Accetta comandi di alto livello per leggere o scrivere un settore
    b. Avvia azioni quali spostare il braccio sulla *track* desiderata e legge o scrive i dati
    c. Calcola e allega i ***checksum*** a ciascun settore per verificare che i dati vengano letti correttamente (se i dati sono corrotti, il *checksum* memorizzato molto probabilmente non corrisponderà al *checksum* ricalcolato)
    d. Garantisce una corretta scrittura rileggendo il settore dopo averlo scritto
    e. Esegue la rimappatura dei settori danneggiati

3.1 DISK SUBSYSTEM



Più dischi collegati a un computer tramite un ***controller*** → le funzionalità di controller (checksum, rimappatura di settori danneggiati) sono spesso eseguite dai controller singoli dischi, così da ridurre il carico sul controller principale.

Famiglie standard di *disk interfaces*:
- Gamma di standard **ATA** (*AT Adaptor*)
- **SATA** (*Serial ATA*)
- Gamma di standard **SCSI** (*Small Computer System Interconnect*)
- Diverse varianti per ogni standard (diverse velocità e capacità)

3.2 MISURE DI PRESTAZIONE DEI DISCHI

- ***Access time:*** il tempo impiegato da quando viene emessa una richiesta di lettura o scrittura all'avvio del trasferimento dei dati. Consiste di:
    - ***Seek time:*** tempo necessario per riposizionare il braccio sulla *track* corretta.
      Il *seek time* medio è ½ del *seek time* peggiore possibile, sarebbe 1/3 se tutte le *track* avessero lo stesso numero di settori ed ignorassimo il tempo di avvio e arresto del movimento del braccio)
      Normalmente va da 4 a 10 millisecondi.
    - ***Rotational latency*** (*latenza di rotazione*): il tempo impiegato perché il settore desiderato si trovi sotto la testina.
      La latenza media è ½ della latenza peggiore.
      Normalmente va da 4 a 11 millisecondi (da 5400 a 15000 rpm).
- ***Data-transfer rate:*** la velocità con cui i dati possono essere recuperati dal disco o memorizzati sul disco.
  La velocità massima va da 25 a 100 MB al secondo, ed è inferiore per le *track* interne.
  È possibile che più dischi condividano un controller, quindi è importante considerare la velocità massima che può essere gestita dal controller (ad esempio ATA-5: 66 MB/sec, SATA: 150 MB/sec, Ultra 320 SCSI: 320 MB/sec, Fiber Channel (FC2Gb): 256 MB/sec)

- ***Mean time to failure*** **(MTTF)**: tempo medio in cui si prevede che il disco funzioni continuamente senza errori.
  In genere va da 3 a 5 anni.
  La probabilità di *failure* dei dischi nuovi è piuttosto bassa, corrispondente a un "MTTF teorico" che va da 500.000 a 1.200.000 ore per un nuovo disco (ad esempio, un MTTF di 1.200.000 ore per un nuovo disco significa che dati 1000 dischi relativamente nuovi, in media uno non funzionerà ogni 1200 ore).
  MTTF diminuisce con l'invecchiamento del disco.

3.3 Ottimizzazione del *Disk-Block Access*

- ***Block***: una sequenza contigua di settori da una singola *track*.
  I dati vengono trasferiti tra il disco e la memoria principale in blocchi.
  Le dimensioni vanno da 512 byte a diversi KB
  - Blocchi più piccoli: più trasferimenti dal disco
  - Blocchi più grandi: più spazio perso a causa di blocchi parzialmente pieni
    ↳ I blocchi di dimensioni tipiche oggi vanno da 4 a 16 kilobyte

- ***Disk-arm-scheduling:*** gli algoritmi di "schedulazione" ordinano i *pending acceses* alle tracks in modo tale che il movimento del braccio del disco sia ridotto al minimo
  - ***Elevator algorithm***: sposta il braccio del disco in una direzione (da quella esterna a quella interna o viceversa), elaborando la richiesta successiva in quella direzione, fino a quando non ci sono più richieste in quella direzione, quindi inverte la direzione e ripete

- ***File organization:*** ottimizza *l'access time* ai blocchi organizzandoli nel modo o nell'ordine in cui i dati saranno richiesti (es: memorizza informazioni correlate sullo stesso cilindro o nelle vicinanze.
  - Col passare del tempo i file possono risultare frammentati (**fragmented**) (es: se i dati sono stati inseriti/eliminati dal file diverse volte oppure se i blocchi liberi su disco sono sparsi e quindi il file appena creato ha i suoi blocchi sparsi sul disco). In questo caso l'accesso sequenziale a un file frammentato determina un aumento del movimento del braccio del disco. Alcuni sistemi dispongono di utilità per deframmentare (**defragment**) il file system, al fine di accelerare l'accesso ai file

- **Write Buffers non volatili:** velocizzano le scritture su disco scrivendo immediatamente blocchi su un buffer RAM non volatile (può essere una *battery backed-up* RAM o una memoria flash), cosicché anche in caso di interruzione dell'alimentazione, i dati siano al sicuro e vengano scritti sul disco al ritorno dell'alimentazione.
  - Il controller quindi scrive sul disco quando il disco non ha altre richieste o se la richiesta è in sospeso (*pending*) da un po' di tempo
  - Le operazioni database che richiedono la memorizzazione sicura dei dati prima di continuare possono continuare senza attendere la scrittura dei dati sul disco
  - *Le scritture possono inoltre essere riordinate prima della scrittura per ridurre al minimo la frammentazione*

- **Log disk**: un disco dedicato alla scrittura di un *sequential log* (registro sequenziale) degli aggiornamenti dei blocchi. È utilizzato esattamente come una RAM non volatile (scrivere sul log disk è molto veloce poiché non sono richiesti i *seeks* e non c'è nessuna necessità di hardware speciale → NV-RAM)

- I file system tipicamente riordinano le scritture su disco per migliorare le prestazioni
  - **Journaling file systems:** scrivono i dati in modo sicuro su NV-RAM o sul disco di registro
  - *Reordering* senza *journaling*: c'è il rischio di corruzione dei *file system data*

4 RAID (Redundant Arrays of Independent Disks)

Il **RAID**, acronimo originariamente di *Redundant Array of Inexpensive Disks* ("insieme ridondante di dischi economici", dove il termine "economici" si riferisce alla necessità di migliorare le prestazioni di un sistema

costituito da memorie di massa a basso costo, sostituendo a dischi ad alte capacità più dischi a basse capacità di storage), è una tecnica di installazione raggruppata di diversi dischi rigidi in un computer (o collegati ad esso) che fa sì che gli stessi nel sistema appaiano e siano utilizzabili come se fossero un unico volume di memorizzazione.

Gli **scopi** del RAID sono:

- aumentare le performance
- rendere il sistema resiliente alla perdita di uno o più dischi
- poter rimpiazzare un disco senza dover interrompere il servizio.

Il RAID sfrutta, con modalità differenti a seconda del tipo di realizzazione, i principi di **ridondanza dei dati** e di **parallelismo** nel loro accesso per garantire, rispetto ad un disco singolo, incrementi di prestazioni, aumenti nella capacità di memorizzazione disponibile, miglioramenti nella tolleranza ai guasti e quindi migliore affidabilità.

Questo insieme di *disk organization techniques* gestiscono un numero elevato di dischi, fornendo all'utente però la vista di un singolo, unico, disco con:

- **alta capacità** e **alta velocità** utilizzando più dischi in **parallelo**,
- **alta affidabilità** memorizzando i dati in modo **ridondante**, in modo che i dati possano essere ripristinati anche in caso di guasto di un disco.

La possibilità che si rompa un disco in un set di *N* dischi è molto più alta della possibilità che il singolo disco si rompa (es: un sistema con 100 dischi, ciascuno con MTTF di 100.000 ore (circa 11 anni), avrà un sistema MTTF di 1000 ore (circa 41 giorni)) → Le tecniche per utilizzare la ridondanza per **evitare la perdita di dati** sono fondamentali per un numero elevato di dischi

Mentre originariamente i RAID (I → *inexpansive*) erano utilizzati per risparmiare sui dischi ad alte capacità, adesso i RAID (I → *independent*) sono utilizzati per la loro maggiore *reliability* e *bandwidth* (affidabilità e larghezza di banda).

4.1 Migliore Affidabilità sfruttando la Ridondanza

Ridondanza: memorizza informazioni extra che possono essere utilizzate per ricostruire le informazioni perse in caso di errore del disco.

Ad esempio, nel ***mirroring*** (o ***shadowing***) si duplica ogni disco (il "disco logico" è costituito da due dischi fisici). Ogni scrittura viene eseguita su entrambi i dischi e le letture possono essere eseguite da entrambi i dischi. Se un disco di una coppia si guasta, i dati rimangono disponibili in quello rimanente. La perdita di dati si verifica solo se un disco si guasta e anche il suo *mirror disk* fallisce prima che il sistema venga riparato, ma la probabilità di eventi combinati è molto piccola (tranne per le modalità di guasto dipendenti come incendio, crollo dell'edificio o picchi di corrente elettrica).

Il ***Mean Time to Data Loss*** (**MTTDL**) dipende dal **MTTF** e dal *Mean Time To Repair* (**MTTR**). Ad esempio con un (MTTF = 100.000 ore) ed un (MTTR = 10 ore) si ha un (MTTDL = $500*10^6$ ore, 57.000 anni) per una coppia di dischi con *mirroring* (ignorando le modalità di guasto co-dipendenti).

4.2 Migliore Prestazioni sfruttando il Parallelismo

Due obiettivi principali del parallelismo in un sistema di dischi:

1. Bilanciare gli accessi di piccole dimensioni per aumentare il *throughput* [2]
2. Parallelizzare gli accessi di grandi dimensioni per ridurre il *response time* (tempo di risposta)

---

[2] Letteralmente "quantità che passa attraverso". È la produzione o il volume trattato, la quantità di materiale processata da una linea di produzione, che passa attraverso un magazzino, la quantità di dati che vengono letti o scritti ecc… in un certo lasso di tempo.

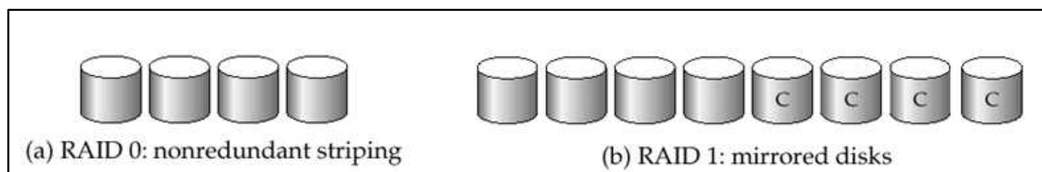
Il RAID migliora la velocità di trasferimento immagazzinato i dati su più dischi:

- **Bit-level striping** - suddivide i bit di ogni byte su più dischi
  - In una matrice di otto dischi, scrive il bit *i*-esimo di ciascun byte sul disco *i.*
  - Ogni accesso può leggere dati ad otto volte la velocità di un singolo disco.
  - Ma il tempo di ricerca/accesso è peggiore di un singolo disco
  - Lo *striping* a livello di bit non viene più utilizzato molto
- **Block-level striping** - con *n* dischi, il blocco *i*-esimo di un file va sul disco (*i* mod *n*) + 1
  - Le richieste per diversi blocchi possono essere eseguite in parallelo se i blocchi si trovano su diversi dischi
  - Una richiesta per una lunga sequenza di blocchi può utilizzare tutti i dischi in parallelo

4.3 LIVELLI RAID

Esistono schemi diversi per *redundancy* a costi inferiori utilizzando lo *striping* del disco combinato con i *parity bits* (bit di parità[3]) → diverse RAID organizations, o **livelli RAID**, hanno caratteristiche di costo, prestazioni e affidabilità diverse.

- **RAID Level 0** (Block striping e no ridondanza)
  - Utilizzato in applicazioni ad alte prestazioni in cui la perdita di dati non è fondamentale
- **RAID Level 1** (Mirrored disks con block striping)
  - Offre migliori prestazioni di scrittura
  - Popolare per applicazioni come la memorizzazione di file di registro in un sistema di database



(a) RAID 0: nonredundant striping (b) RAID 1: mirrored disks

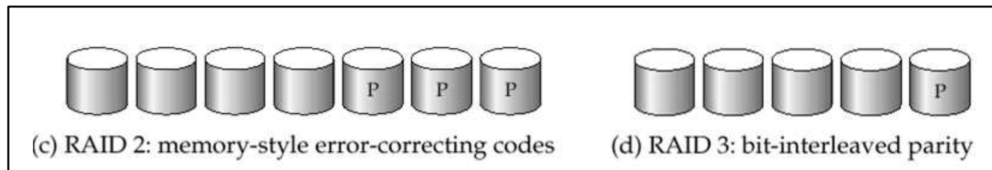
- **RAID Level 2** (Memory-Style Error-Correcting-Codes[4] (**ECC**) con bit striping)
- **RAID Level 3** (Bit-Interleaved Parity)
  - Cosa vuole dire ***Bit-Interleaved Parity*** *?* Significa che un singolo bit di parità è sufficiente per la correzione degli errori, non solo per il rilevamento, poiché sappiamo quale disco ha fallito:
    - Durante la scrittura dei dati, i bit di parità corrispondenti devono anche essere calcolati e scritti su un disco di bit di parità
    - Per recuperare i dati in un disco danneggiato, si fa lo XOR di bit da altri dischi (incluso il disco bit di parità)

---

[3] Il **bit di parità** è un codice di controllo usato per prevenire errori nella memorizzazione dei dati, che prevede l'aggiunta di un bit ridondante ai dati, calcolato a seconda che il numero di bit che valgono 1 sia pari o dispari (è uno dei codici di rilevazione e correzione d'errore più semplici). Ci sono due varianti: **bit di parità pari** (1 se il numero di "1" in un certo insieme di bit è dispari) e **bit di parità dispari** (1 se il numero di "1" in un certo insieme di bit è pari).
È un caso particolare di *Cyclic Redundancy Check* (CRC), quando il 1-bit CRC è generato dal polinomio $x + 1$.
→ Ma come rileva gli errori?
Se un numero dispari di bit (incluso il bit di parità) è cambiato durante la memorizzazione di un insieme di bit, allora il bit di parità non sarà corretto e indicherà che è avvenuto un errore durante la trasmissione (**codice di controllo**, ma **non di correzione** d'errore → nei RAID si può correggere per mezzo della ridondanza).
l metodo del bit di parità ha il vantaggio di usare un solo bit di spazio, e richiede solo un certo numero di porte XOR per esser generato.

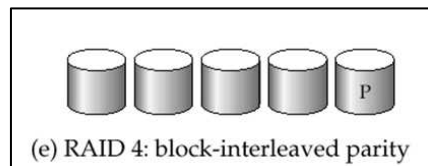
[4] Letteralmente "Codici di correzione degli errori in stile memoria". Le **memorie con ECC** hanno dei sistemi utili a rintracciare eventuali errori contenuti nell'informazione memorizzata e dei meccanismi capaci di correggere l'errore riscontrato. Questo è possibile registrando informazioni aggiuntive che rendono queste memorie più costose e poco più lente delle rispettive memorie non dotate di ECC.

- Trasferimento dati più veloce rispetto a un singolo disco, ma meno I/O al secondo poiché ogni disco deve partecipare a ogni I/O.
- Sussume il RAID 2 (fornisce tutti i suoi benefici, a un costo inferiore)



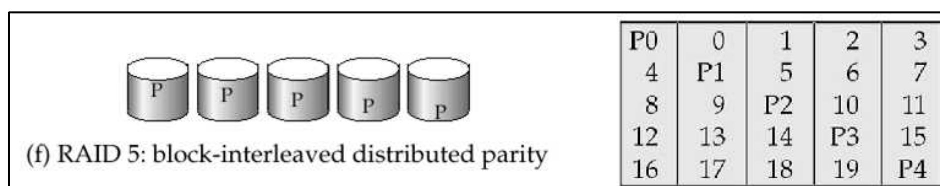
(c) RAID 2: memory-style error-correcting codes (d) RAID 3: bit-interleaved parity

- **RAID Level 4** (Block-Interleaved Parity)
  - Cosa vuole dire ***Block-Interleaved Parity*** *?* Significa che viene utilizzato il *block-level striping* mantenendo un *parity block* (blocco di parità) su un disco separato per i blocchi corrispondenti a *N* altri dischi:
    - Quando si scrive un blocco dati, anche il blocco corrispondente dei bit di parità deve essere calcolato e scritto sul disco di parità
    - Per trovare il valore di un blocco danneggiato, calcolare XOR di bit dai blocchi corrispondenti (incluso il blocco di parità) da altri dischi.
  - Fornisce frequenze I/O più elevate per le letture di blocchi indipendenti (rispetto al RAID 3), questo perché la lettura dei blocchi è sul singolo disco, quindi i blocchi memorizzati su dischi diversi possono essere letti in parallelo
  - Fornisce velocità di trasferimento elevate per le letture da più blocchi rispetto al non avere *striping*
  - Prima di scrivere un blocco, i dati di parità devono essere calcolati:
    - È possibile utilizzare il vecchio blocco di parità, il vecchio valore del blocco corrente e il nuovo valore del blocco corrente (2 letture di blocco + 2 scritture di blocco)
    - Oppure ricalcolando il valore di parità utilizzando i nuovi valori dei blocchi corrispondenti al blocco di parità (più efficiente per la scrittura di grandi quantità di dati in sequenza)
  - Il blocco di parità diventa un collo di bottiglia per scritture di blocchi indipendenti poiché ogni blocco scrive anche sul disco di parità



(e) RAID 4: block-interleaved parity

- **RAID Level 5** (Block-Interleaved Distributed Parity)
  - Cosa vuole dire ***Block-Interleaved Distributed Parity****?* Significa che si hanno blocchi di dati e blocchi di parità tra tutti gli N+1 dischi (invece di memorizzare i dati in N dischi ed i blocchi di parità in 1 disco).
  - Tassi I/O più alti rispetto al livello 4.
  - Le scritture di blocchi si verificano in parallelo se i blocchi e i relativi blocchi di parità si trovano su dischi diversi.
  - Sussume il RAID4: offre gli stessi vantaggi, ma evita il collo di bottiglia dovuto al disco di parità.

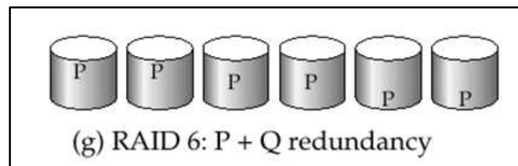


(f) RAID 5: block-interleaved distributed parity

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

- **RAID Level 6** (P+Q Redundancy scheme)
  - Cosa vuole dire avere un ***P+Q Redundancy scheme****?* È simile al RAID5, ma memorizza informazioni ridondanti extra per proteggersi da guasti multipli del disco.

- Migliore affidabilità rispetto al RAID5 ad un costo maggiore → non usato così ampiamente.



(g) RAID 6: P + Q redundancy

4.4 SCELTA DEL LIVELLO RAID

Fattori nella scelta del livello RAID:

- Costo monetario
- Prestazioni: numero di operazioni I/O al secondo e larghezza di banda a normale funzionamento
- Prestazioni durante i guasti (*failures*)
- Prestazioni durante la ricostruzione (*rebuild*) del disco guasto (compreso il tempo necessario per ricostruire il disco guasto)

→ **RAID 0**: utilizzato solo quando la sicurezza dei dati non è importante (es: i dati possono risultare accessibili rapidamente da altre fonti)

→ **RAID 2,4**: mai utilizzati poiché sussinti in RAID 3,5

→ **RAID 3**: non viene più utilizzato poiché il ***bit**-striping* per la lettura di un singolo blocco deve accedere a tutti i dischi (sprecando movimento della testina). Il ***block**-striping* invece evita ciò → **RAID 5**

→ **RAID 6:** usato raramente poiché RAID 1,5 offrono una sicurezza adeguata per quasi tutte le applicazioni

Quindi la "competizione" è tra 1 e 5:

- RAID 1 fornisce prestazioni di scrittura migliori di molto rispetto al RAID 5
  - RAID 5 richiede almeno 2 letture di blocchi e 2 scritture di blocchi per scrivere un singolo blocco, mentre il livello 1 richiede solo 2 scritture di blocco
  - → **RAID 1** è preferito per ambienti ad alto aggiornamento come i log disks

- RAID 1 ha uno *storage cost* (costo di archiviazione) superiore al RAID 5
  - Le capacità dei dischi rigidi aumentano rapidamente (50% l'anno) mentre i tempi di accesso al disco diminuiti molto di meno (x3 in 10 anni)
  - I requisiti di I/O sono notevolmente aumentati, ad esempio per i server Web
  - Quando sono stati acquistati abbastanza dischi per soddisfare il tasso richiesto di I/O, spesso hanno una *spare storage capacity* (capacità di archiviazione di riserva/inutilizzata, perciò non c'è spesso nessun costo monetario extra per il RAID 1!)
  - → **RAID 5** è preferito per le applicazioni con bassa frequenza di aggiornamento, e grandi quantità di dati
  - → **RAID 1** è preferito per tutte le altre applicazioni

4.5 PROBLEMI HARDWARE

Software RAID: implementazioni RAID eseguite interamente nel software, senza supporto hardware speciale.

Hardware RAID: implementazioni RAID con hardware speciale:

- Utilizzare una RAM non volatile per registrare le scritture che vengono eseguite
- Attenzione: un guasto dell'alimentazione durante la scrittura può causare danni al disco
  - Ad es: se avviene guasto dopo aver scritto un blocco ma prima di scrivere il secondo in un sistema con *mirroring*, tali dati corrotti devono essere identificati e riparati al ripristino dell'alimentazione, similmente a come accade in casi di disco guasto. In casi come questo la NV-RAM aiuta a rilevare in modo efficiente i blocchi potenzialmente danneggiati. Nel caso non fosse possibile, tutti i blocchi del disco devono essere letti e confrontati con il *mirror/parity block*

Hot Swapping: sostituzione del disco "a caldo", mentre il sistema è in esecuzione, senza spegnimento (supportato da alcuni sistemi RAID hardware, riduce i tempi di recupero e migliora notevolmente la disponibilità)

Molti sistemi gestiscono ***spare disks*** (dischi di riserva) che vengono tenuti online e utilizzati come sostituti dei dischi guasti appena rilevato un guasto → riduce notevolmente il tempo di recupero

Molti sistemi RAID hardware assicurano che un singolo punto guasto non interrompa il funzionamento del sistema utilizzando:

- Alimentatori “ridondanti” con una batteria di backup
- Controllori multipli e interconnessioni multiple per evitare guasti del controller/dell’interconnessione

4.6 RAID NEL SETTORE INDUSTRIALE

La terminologia RAID nel settore industriale non ha un vero e proprio standard (ad esempio molti venditori usano **RAID 1** per indicare il sistema *mirroring without striping* e **RAID 10** oppure **RAID 1+0**: per indicare il sistema *mirroring with striping*

Le implementazioni "Hardware RAID" spesso scaricano l'elaborazione RAID su un sottosistema separato, ma non offrono NVRAM. Bisogna quindi leggere attentamente le specifiche!

Il software RAID è direttamente supportato dalla maggior parte dei sistemi operativi di oggi.

5 DISCHI OTTICI

Compact disk-read only memory **(CD-ROM)**:

- dischi removibili, 640 MB per disco,
- *seek time* di circa 100 msec (la testina di lettura ottica è più pesante e più lenta rispetto ai dischi magnetici),
- maggiore latenza (3000 RPM) rispetto ai dischi magnetici
- velocità di trasferimento dati inferiore (3-6 MB/s) rispetto ai dischi magnetici

Digital Video Disk **(DVD)**:

- DVD-5 contiene 4,7 GB e DVD-9 contiene 8,5 GB
- DVD-10 e DVD-18 sono formati fronte/retro con capacità di 9,4 GB e 17 GB
- *Seek time* lento, per gli stessi motivi del CD-ROM

Sono molto diffuse le versioni **record once** (CD-R e DVD-R):

- i dati possono essere scritti solo una volta e non possono essere cancellati.
- alta capacità e lunga durata; utilizzato per la conservazione di archivi

Disponibili anche versioni **multi-scrittura** (CD-RW, DVD-RW, DVD + RW e DVD-RAM)

6 NASTRI MAGNETICI

Può immagazzinare grandi volumi di dati e fornisce velocità di trasferimento elevate:

- Pochi GB per il formato DAT (Digital Audio Tape),
- 10-40 GB con il formato DLT (Digital Linear Tape),
- 100 GB+ con il formato Ultrium
- 330 GB con il formato di scansione elicoidale Ampex
- Velocità di trasferimento da qualche a decine di MB/s

Attualmente sono il supporto di memorizzazione più economico poiché i nastri sono economici, **ma** il costo delle unità di lettura dei nastri è molto alto!!

Access time molto lento rispetto ai dischi magnetici e ai dischi ottici:

- limitato ad accesso sequenziale.
- alcuni formati (Accelis) offrono una ricerca più rapida al costo di una capacità inferiore

Utilizzati principalmente per il backup, per la memorizzazione di informazioni utilizzate di rado e come mezzo off-line per il trasferimento di informazioni da un sistema a un altro.

I **Tape jukebox** sono utilizzati per la memorizzazione di grandi capacità (terabyte $=10^{12}$ byte, petabye = 10^{15} byte, ecc…)

7 ACCESSO ALLO STORAGE

Il *database file* è suddiviso in unità di memoria di lunghezza fissa denominate **blocks** (blocchi). I blocchi sono unità di allocazione dello spazio di *storage* (archiviazione) e trasferimento dei dati.

Il *database system* cerca di minimizzare il numero di trasferimenti di blocchi tra lo *storage* e la memoria. Possiamo ridurre il numero di accessi ad esso mantenendo il maggior numero possibile di blocchi nella memoria principale (*main memory*) mediante:

- **Buffer**: porzione della memoria principale disponibile per archiviare copie dei blocchi del disco.
- **Buffer manager**: sottosistema responsabile per l'allocazione dello spazio del buffer nella memoria principale.

7.1 BUFFER MANAGER

I programmi inviano una richiesta al *buffer manager* quando hanno bisogno di un blocco dallo *storage*. Il *buffer manager* si comporta nel seguente modo:

1. Se il blocco è già nel *buffer*, il *buffer manager* restituisce l'indirizzo del blocco nella memoria principale
2. Se il blocco non è nel *buffer*, il *buffer manager*
   a. gli assegna uno spazio nel buffer sostituendo (*replacing*, “buttando via”), se necessario, qualche altro blocco, per fare spazio al nuovo blocco.
   b. il blocco sostituito, prima di essere “buttato via”, è riscritto nello *storage*, ma solo se è stato modificato dall'ultima volta in cui è stato scritto/scaricato dallo *storage*.
3. Il *buffer manager* poi copia il blocco dallo *storage* al *buffer* e restituisce l'indirizzo del blocco nella memoria principale al richiedente.

Vediamo adesso alcune delle “politiche di sostituzione (*replacement*)” del buffer:

- **Most recently used (MRU)** strategy: è adottata dalla maggior parte dei sistemi operativi, sostituisce il blocco utilizzato meno di recente.
  - Idea dietro LRU: usa il pattern dei riferimenti ai blocchi precedente come predittore dei riferimenti futuri
  - Le query hanno schemi di accesso ben definiti (ad esempio: scansioni sequenziali) e un database system può utilizzare le informazioni nella query di un utente per prevedere futuri riferimenti
  - LRU può essere una cattiva strategia per determinati schemi di accesso che comportano scansioni di dati ripetute. Ad esempio:

```
e.g. when computing the join of 2 relations r and s by a nested loops
  for each tuple tr of r do
    for each tuple ts of s do
      if the tuples tr and ts match …
```

  - È quindi preferibile una strategia mista sfruttando i suggerimenti del *query optimizer* per definire la propria “strategia di sostituzione”

- **Pinned** (Bloccato) **Block**: blocco di memoria che non può essere riscritto sul disco.

- **Toss-immediate** strategy: libera lo spazio occupato da un blocco non appena è stata elaborata la tupla finale di quel blocco.

- **Most recently used (MRU)** strategy: il sistema deve bloccare (*pin*) il blocco attualmente in fase di elaborazione. Dopo che la tupla finale di quel blocco è stata elaborata, il blocco viene sbloccato (*unpinned*) e diventa il blocco utilizzato più di recente.
- Il *buffer manager* può utilizzare le informazioni **statistiche** relative a quanto sia probabile che una richiesta faccia riferimento a una determinata relazione (ad esempio, se si accede frequentemente al *data dictionary*, si conservano i blocchi del *data dictionary* nel *buffer* della memoria principale)
- I *Buffer managers* supportano anche il ***forced output*** di blocchi a fini di *recovery* (recupero)

8 ORGANIZZAZIONE DEI FILE

Il database è memorizzato come una raccolta di ***file***. Ogni *file* è una sequenza di ***record***. Un *record* è una sequenza di ***fields*** (campi). Un approccio è quello di:

- supporre che la dimensione del *record* sia fissa (*Fixed-Length*) [5]
- supporre che ogni *file* abbia solo *record* di un particolare tipo
- supporre che diversi *file* vengano utilizzati per diverse relazioni

8.1 FIXED-LENGTH RECORDS

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Approccio semplice:

- Memorizzare *i*-esimo *record* iniziando dal byte $n*(i-1)$, dove *n* è la dimensione fissa di ciascun *record*.
- L'accesso ai *record* è semplice, ma i *record* così definiti possono "attraversare" i blocchi! → effettuiamo una modifica concettuale: non consentiremo ai *record* di superare i limiti dei blocchi

Per la cancellazione del *record i*-esimo vediamo alcune alternative:

- spostare i *record* $(i+1,\ldots,n)$-esimi in $(i,\ldots,n-1)$-esima posizione
- spostare il *record n*-esimo in *i*-esima posizione
- non spostare i *record*, ma collegare (*link*) tutti i *record* liberi in una "lista dei *record* liberi" (*free list*)

Vediamo alcune caratteristiche della ***free list:***

- Memorizza l'indirizzo del primo *record* cancellato nell'intestazione del file.
- Utilizza questo primo *record* per memorizzare l'indirizzo del secondo *record* cancellato e così via
- Si può pensare a questi indirizzi memorizzati come dei <u>puntatori</u>, poiché "puntano" alla posizione di un *record*.
- Vi è un utilizzo più efficiente dello spazio disponibile riutilizzare gli "spazi liberi" dei *record* cancellati per memorizzare i puntatori. (Nessun puntatore viene memorizzato in *record* in uso.)

header			
record 0	A-102	Perryridge	400
record 1			
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4			
record 5	A-201	Perryridge	900
record 6			
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

8.2 VARIABLE-LENGTH RECORDS

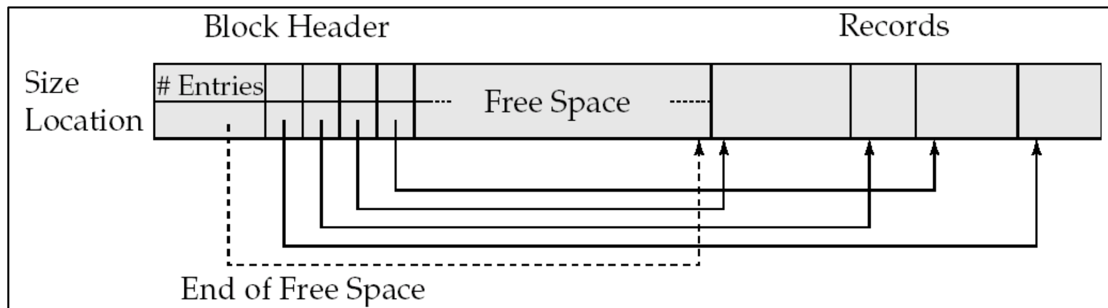
I record a lunghezza variabile si utilizzano nei sistemi di database in diversi modi:

- Per l'archiviazione di più tipi di *record* in un file.
- Registrare i *types* che consentono lunghezze variabili per uno o più campi (*fields*).
- Registrare i *types* che consentono campi (*fields*) ripetuti (presenti in alcuni vecchi *data models*).

---

[5] Questo caso è più semplice da implementare (considereremo i record di lunghezza variabile in seguito)

Variable-Length Records: Slotted Page Structure:



L'*header* **Slotted page** contiene:

- Il numero di voci di *record* (# *record* *entries* come in figura)
- La fine dello spazio libero nel blocco (*end of free space*, come in figura)
- La posizione (*location*) e la dimensione (*size*) di ogni *record* (come in figura)

I *record* possono essere spostati all'interno di una pagina per mantenerli contigui (senza spazi vuoti tra loro); la voce (*entry*) nell'header deve essere però aggiornata.

I puntatori non dovrebbero puntare direttamente al *record*, ma dovrebbero puntare l'*header* alla voce (*entry*) del *record*.

8.3 ORGANIZZAZIONE DEI RECORD NEI FILE

Heap ("mucchio, pila"): un *record* può essere inserito ovunque nel file, dove c'è spazio

Sequenziale: memorizza i *record* in ordine sequenziale, in base al valore della *search key* (chiave di ricerca) di ciascun *record*

Hashing: una funzione *hash* [6] calcolata su alcuni attributi di ciascun *record*; il risultato specifica in quale blocco del file deve essere inserito il *record*.

I *record* di ciascuna relazione possono essere archiviate in un file separato. In una ***multitable clustering file organization*** i *record* di più relazioni diverse possono essere memorizzati nello stesso file (Motivazione: archiviare i *record* correlati sullo stesso blocco, per ridurre al minimo l'I/O)

8.4 ORGANIZZAZIONE FILE SEQUENZIALE

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Adatta per applicazioni che richiedono l'elaborazione sequenziale dell'intero file

I *record* nel file sono ordinati da una **search key** (chiave di ricerca)

L'eliminazione utilizza le catene di puntatori

Nell"inserimento si inserisce il record il record in un determinata locazione con il seguente criterio:

- se c'è spazio libero, si inserisce lì
- se non ci sono spazi liberi, si inserisce il record in un blocco di overflow

[6] Nel linguaggio matematico e informatico, l'***hash*** è una funzione non invertibile che mappa una stringa di lunghezza arbitraria in una stringa di lunghezza minore. Esistono numerosi algoritmi che realizzano funzioni hash con particolari proprietà che dipendono dall'applicazione.
Nelle applicazioni di basi di dati la funzione ***hash*** è usata per realizzare una particolare struttura dati chiamata ***hash table***. In questa applicazione non occorrono proprietà crittografiche e generalmente l'unica proprietà richiesta è che non ci siano ***hash*** più probabili di altri.

In entrambi i casi, dopo l'inserimento la catena di puntatori deve essere aggiornata.

È necessario riorganizzare il file di volta in volta per ripristinare l'ordine sequenziale

8.5 MULTITABLE CLUSTERING FILE ORGANIZATION

Questa *file organization* memorizza i *record* correlati di due o più relazioni in ogni blocco. Questa organizzazione ci consente di leggere facilmente i *record* che soddisfano la condizione di join utilizzando un blocco di lettura. Quindi, siamo in grado di elaborare questa particolare query in modo più efficiente.

In figura l'attributo dept_name è omesso dai *record* dell'instructor poiché può essere desunto dal department associato; l'attributo può essere mantenuto in alcune implementazioni, per semplificare l'accesso agli attributi. Supponiamo inoltre che ogni *record* contenga l'identificatore della relazione a cui appartiene, anche se questo non è mostrato in figura.

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

The *department* relation.

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

The *instructor* relation

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

Multitable clustering file structure.

L'utilizzo del *clustering* di più tabelle in un singolo file migliora l'elaborazione di un particolare join ($department \bowtie$ instructor), ma risulta rallentata l'elaborazione di altri tipi di query. Per esempio la query [select * from department] richiede più accessi ai blocchi di quanto non facesse nello schema dove abbiamo archiviato ogni relazione in un file separato, dal momento che ogni blocco ora contiene significativamente meno *record* di department. Per localizzare in modo efficiente tutte le tuple della relazione di department nella struttura in figura, possiamo mettere insieme tutti i *record* di quella relazione usando i puntatori:

Comp. Sci.	Taylor	100000	
45564	Katz	75000	
10101	Srinivasan	65000	
83821	Brandt	92000	
Physics	Watson	70000	
33456	Gold	87000	

Multitable clustering file structure with pointer chains.

L'uso del *clustering multitable* dipende quindi dai tipi di query che il progettista del database ritiene siano più frequenti. Un suo uso attento può produrre significativi miglioramenti delle prestazioni nell'elaborazione della query.

8.6 ARCHIVIAZIONE DEL DATA DICTIONARY

I *data dictionary*, chiamati anche *system catalog*, memorizzano metadati, ovvero "*data about data*" come:

- Informazioni sulle relazioni
  - nomi di relazioni
  - nomi e tipi di attributi di ciascuna relazione
  - nomi e definizioni di *views*
  - vincoli di integrità (integrity constraints)
- Informazioni sull'utente e sulla contabilità, incluse le password
- Dati statistici e descrittivi (es: numero di tuple in ogni relazione)

- Informazioni sull'organizzazione fisica dei file
    - Come viene memorizzata la relazione (sequenziale / hash / ...)
    - Posizione fisica della relazione
- Informazioni sugli indici (vedi capitolo 12)
    - Archiviazione del dizionario dati (continua)

Vi sono diverse strutture possibili per i *data dictionaries/ system catalogs:*

- Rappresentazione relazionale su disco
- Strutture dati specializzate progettate per un accesso efficiente, in memoria

Vediamo una possibile rappresentazione del catalogo:

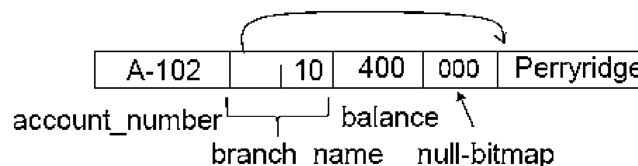
Relation_metadata = (*<u>relation_name</u>, number_of_attributes, storage_organization, location*)
Attribute_metadata = (*<u>relation_name, attribute_name</u>, domain_type, position, length*)
User_metadata = (*<u>user_name</u>, encrypted_password, group*)
Index_metadata = (*<u>relation_name, index_name</u>, index_type, index_attributes*)
View_metadata = (*<u>view_name</u>, definition*)

9 Extra

9.1 Rappresentazione di record fixed-length

I *record* ***fixed-length*** sono facili da rappresentare essendo simili ai *record* (structs) nei linguaggi di programmazione. Esistono estensioni per rappresentare valori null, ad esempio *<u>bitmaps</u>* che indicano quali attributi sono nulli.

I campi (*fields*) di lunghezza variabile possono essere rappresentati da una coppia (offset, lenght) dove offset è la posizione all'interno del *record* e lenght è lunghezza del campo. Tutti i campi iniziano nella posizione predefinita, ma è richiesta un'ulteriore indicazione per i campi di lunghezza variabile:



Example record structure of *account* record

9.2 Rappresentazione di record variable-length

Il modo più immediato per rappresentare i *record* a lunghezza variabile (*variable-lenght*) è il seguente:

Byte-string representation: allega un carattere *end-of-record* ⊥ alla fine di ogni *record.* Vi sono però difficoltà di eliminazione e di crescita (→ vedremo sotto delle alternative)

0	Perryridge	A-102	400	A-201	900	A-218	700	⊥
1	Round Hill	A-305	350	⊥				
2	Mianus	A-215	700	⊥				
3	Downtown	A-101	500	A-110	600	⊥		
4	Redwood	A-222	700	⊥				
5	Brighton	A-217	750	⊥				

Per rappresentare i *record* a lunghezza variabile (*variable-lenght*) con i *record* a lunghezza fissa (*fixed-lenght*) si possono utilizzare due metodi: il metodo dello spazio riservato ed il metodo dei puntatori. Vediamoli:

Spazio riservato (*reserved space*): rappresentare i *variable-lenght record* **ma** di una lunghezza massima nota; lo spazio inutilizzato nei *record* più brevi viene riempito con un un carattere null o *end-of-record* ⊥

0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-110	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	Brighton	A-217	750	⊥	⊥	⊥	⊥

Metodo dei puntatori: Un *variable-length record* è rappresentato da una lista di *fixed-length record*, concatenati tramite puntatori. Può essere utilizzato anche se la lunghezza massima del record non è nota.

0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

Lo svantaggio della struttura a puntatore è che dello spazio viene sprecato in tutti i record, tranne che nel primo di una catena (nella figura notare le caselle vuote nella prima colonna).

La soluzione è quella di consentire due tipi di blocchi nel file:

- *Anchor block* ("blocco di ancoraggio"): contiene i primi record di catena
- *Overflow block*: contiene record diversi da quelli che sono i primi record delle catene.

anchor block

Perryridge	A-102	400	
Round Hill	A-305	350	
Mianus	A-215	700	
Downtown	A-101	500	
Redwood	A-222	700	
Brighton	A-217	750	

overflow block

A-201	900	
A-218	700	
A-110	600	

9.3 Argomenti non trattati

Dalla slide 11.77 alla slide 11.90 sono presenti i seguenti argomenti non trattati a lezione

- Mapping of Objects to Files (11.77)
- Management of Persistent Pointers (11.79)
- Hardware Swizzling (11.81)
- Disk versus Memory Structure of Objects (11.88)
- Large Objects (11.89) and Modifying Large Objects (11.90)

Cap 12: Indicizzazione e hashing

1 CONCETTI BASILARI

I meccanismi di indicizzazione sono utilizzati per accelerare l'accesso ai dati desiderati (ad es., Catalogo degli autori in una libreria)

Search key (Chiave di ricerca): attributo al set di attributi utilizzati per la ricerca registra in un file.

Un ***index file*** (file di indice) è costituito da dei *record* chiamati ***index entries*** ("voci di indice"), nella forma:

[search-key|pointer]

Brighton → A-217 | Brighton | 750

index record — record

Gli *index files* sono in genere molto più piccoli del file originale. Esistono due tipi base di indici:

- ***Ordered indices***: le *search keys* sono memorizzate in ordine
- ***Hash indices***: le *search keys* sono distribuite uniformemente in "secchi" ("*bucket*") utilizzando una *funzione di hash*

1.1 METRICHE DI VALUTAZIONE DEGLI INDICI

Vediamo alcune metriche di valutazione degli indici (*Index Evaluation Metrics*):

- *Access types* supportati in modo efficiente. Per esempio:
  - registra con un valore specificato nell'attributo
  - registra con un valore di attributo che cade in un intervallo specificato dei valori (ad esempio 10000 < salario <40000)
- *Access time*
- *Insertion time*
- *Deletion time*
- *Space overhead* (overhead = spese generali)

2 ORDERED INDICES

In un "indice ordinato" le *index entries* vengono archiviate ordinate secondo un valore della *search key* (ad esempio, il catalogo degli autori in una libreria)

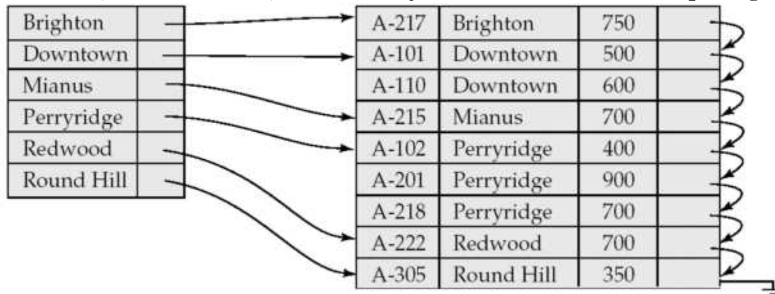
Primary index:: in un file ordinato sequenzialmente, l'indice di cui la *search key* specifica l'ordine sequenziale del file. È chiamato anche ***clustering index*** ("indice di raggruppamento). La *search key* di un indice primario è solitamente, ma non necessariamente, la *primary key*

Secondary index: un indice la cui *search key* specifica un ordine diverso dall'ordine sequenziale del file. Chiamato anche ***non-clustering index***.

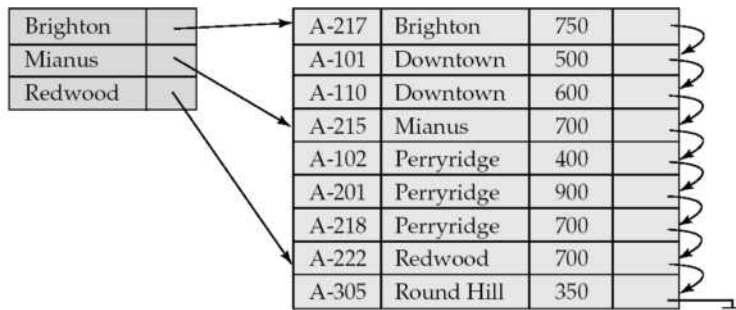
Un ***Index-sequential file*** è un file sequenziale ordinato con un indice primari.

2.1 DENSE INDEX E SPARSE INDEX

Dense index (“indice denso”) ***file***: l’*index file* contiene *index records* per ogni *search key* nel file:



Sparse index (“indice scarso, rado”) ***file***: l’*index file* contiene *index records* per solo per alcuni *search key* nel file



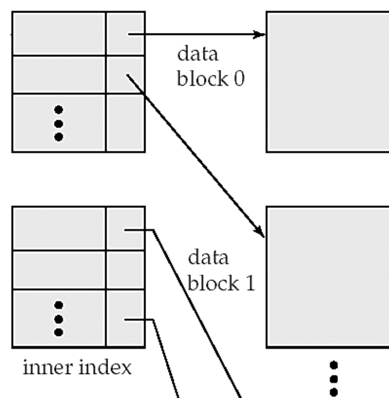
Uno ***sparse index file*** è applicabile quando i *record* sono ordinati sequenzialmente rispetto alla *search key*. Per localizzare un *record* con il valore della *search key* K si deve:

- trovare l’*index record* con maggior valore della *search key* tra quelle che hanno valore < K
- cercare il file sequenzialmente iniziando dal *record* a cui l'*index record* trovato punta

Rispetto a un *dense index file*:

- necessita di meno spazio e meno *maintenance overhead* (“spese generali di mantenimento”) per inserimenti e eliminazioni
- generalmente è più lento del *dense index file* nel localizzare i record.

Un buon compromesso è utilizzare lo *sparse index file* con una *index entry* per ogni blocco nel file, corrispondente al valore minimo della *search key* nel blocco:

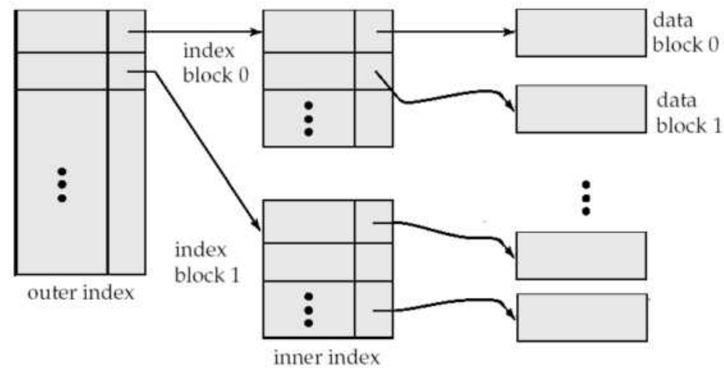


2.2 MULTILEVEL INDEX

Se il *primary index* è troppo grande per essere inserito nella memoria principale, la soluzione consta nel memorizzarlo su disco come un file sequenziale e costruire uno *sparse index file* su di esso:

- ***Outer index*** – uno *sparse index* di un *primary index*
- ***Outer index*** – il *primary index file*

Se anche l'*outer index* è troppo grande per essere inserito nella memoria principale, si necessita ancora di un altro livello di indice, e così via.



Gli indici, all'inserimento o alla cancellazione dal file, devono essere aggiornati, a tutti i livelli.

2.3 AGGIORNAMENTO DEGLI INDICI

Dopo l'eliminazione di un *record*:

Se il *record* eliminato era l'unico *record* nel file con la sua particolare *search key*, anche la *search key* viene cancellata dall'*index file*.

Eliminazione da un indice a singolo livello:

- *Dense indeces* - cancellazione della *search key*: simile alla cancellazione di un *record* in un file.
- *Sparse indeces*:
    - se il *key value* eliminato esiste nell'*index*, il valore viene sostituito dal valore più prossimo al valore della *search key* nel file (seguendo l'ordine delle *search key*)
    - se il successivo valore della *search key* ha già un *index entry*, la *entry* viene cancellata invece di essere sostituita.

Dopo l'inserimento di un *record*:

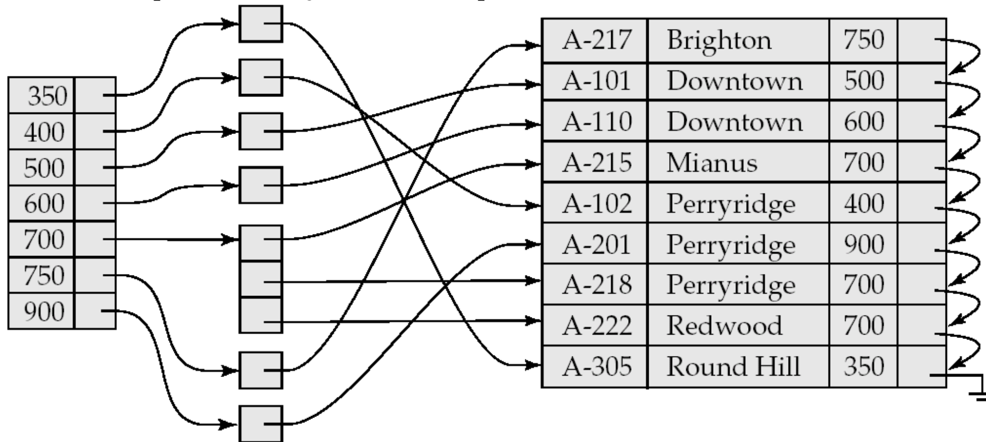
Inserimento di un indice a singolo livello:

- Eseguire una ricerca utilizzando il *key value* del *record* inserito
- *Dense indeces* - se il valore della *search key* non appare nell'*index*, inseriscilo.
- *Sparse indeces* - se l'*index* memorizza una voce (*entry*) per ogni *block* del file, non è necessario apportargli modifiche a meno che non sia stato creato un nuovo blocco.
    - Se invece è stato creato un nuovo blocco, il primo *search key value* che appare nel nuovo blocco viene inserito nell'*index*.

Gli algoritmi di inserimento multilivello (come anche quelli per l'eliminazione) sono semplici estensioni degli algoritmi a livello singolo

2.4 Esempio di indici secondari

Vediamo un esempio di *Secondary index* sul campo *balance* di *account*



↳ Gli *index* records puntano ad un *bucket* che contiene puntatori a tutti i *record* effettivi (quelli del file indicizzato) con quel particolare valore della *search key*.

Gli indici secondari devono essere *dense indexes*

2.5 Dettagli su indici primari e secondari

Gli indici offrono sostanziali benefici nella ricerca di *record* MA aggiornarli impone un costo aggiuntivo (*overhead*) alle modifiche del database: quando un file viene modificato infatti, ogni indice sul file deve essere aggiornato.

Per quanto riguarda la scansione sequenziale:

- usando l'indice primario è efficiente,
- usando un indice secondario è costosa: ogni accesso al *record* può prelevare (*fetch*) un nuovo blocco dal disco, ed il *block fetch* richiede da 5 a 10 microsecondi, contro i circa 100 nanosecondi del *memory access*.

Vedremo quindi come ovviare ad alcuni svantaggi derivanti dall'utilizzare file sequenziali indicizzati.

3 I B-Alberi (da Wikipedia)

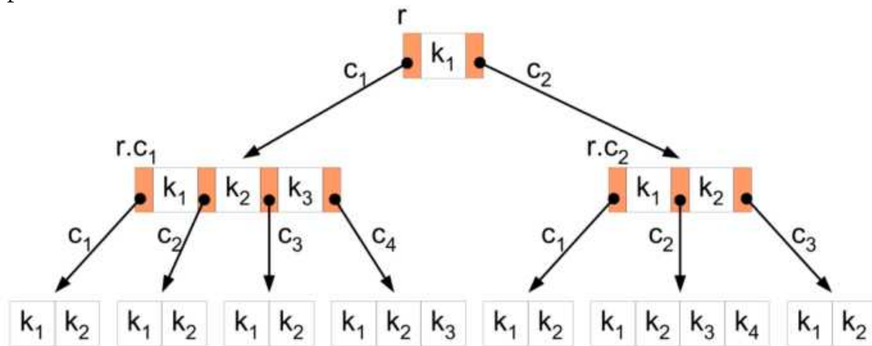
I **B-alberi** (*B-tree*) sono delle strutture di dati/metodi che permettono la rapida localizzazione dei file (*records* o *keys*), specie nei database, riducendo il numero di volte che un utente necessita per accedere alla memoria in cui il dato è salvato.

Essi derivano dagli alberi di ricerca, in quanto ogni chiave appartenente al sottoalbero sinistro di un nodo è di valore inferiore rispetto a ogni chiave appartenente ai sottoalberi alla sua destra; inoltre, la loro struttura ne garantisce il bilanciamento: per ogni nodo, le altezze dei sottoalberi destro e sinistro differiscono al più di una unità.

↳ Questo è il vantaggio principale dei *B-Tree*, e permette di compiere operazioni di inserimento, cancellazione e ricerca in tempi ammortizzati logaritmicamente.

Sono utilizzati spesso nell'ambito dei database, in quanto permettono di accedere ai nodi in maniera efficiente sia nel caso essi siano disponibili in memoria centrale (tramite una cache), sia qualora essi siano presenti solo sulla memoria di massa.

Vediamo un esempio di struttura del B-albero:



3.1 Definizione del B-Albero

Un B-Albero è un albero radicato (la cui radice può essere indicata come root[T] che soddisfa le seguenti proprietà:

1. Ogni nodo x ha i seguenti attributi:
   - $n[x]$, il numero di chiavi memorizzate in x
   - indicando con $key_i[x]$ l'i-esima chiave del nodo x si ha $key_1[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x]$
   - $leaf[x]$ è un valore booleano che può assumere valore $\mathbf{true}$ se il nodo x è una foglia, $\mathbf{false}$ altrimenti.
2. Ogni nodo interno ha $n[x]+1$ puntatori $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ ai suoi figli.
3. Vale $c_1[x] \leq key_1[x] \leq c_2[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x] \leq c_{n[x]+1}[x]$
4. Tutte le foglie hanno la stessa profondità (l'albero è *completamente bilanciato*)
5. Il numero di chiavi per nodo è limitato superiormente ed inferiormente. $t \geq 2$ è il grado minimo dell'albero
   - Ogni nodo contiene almeno $t-1$ chiavi
   - Ogni nodo contiene al massimo $2t-1$ chiavi

3.2 Vantaggi dei B-Alberi

I B-Alberi portano forti vantaggi in termini di velocità ed efficienza rispetto ad implementazioni alternative quando la maggior parte dei nodi si trovano in una memoria secondaria, ad esempio in un disco fisso.

Massimizzando il numero di nodi figli per ogni nodo, l'altezza dell'albero si riduce, l'operazione di bilanciamento è necessaria meno spesso e quindi l'efficienza aumenta. Generalmente questo numero è impostato in modo tale che ogni nodo occupi per intero un gruppo di settori: così, dato che le operazioni di basso livello accedono al disco per cluster, si minimizza il numero di accessi ad esso.

Offrono ottime prestazioni per quanto riguarda sia le operazioni di ricerca che quelle di aggiornamento, poiché entrambe possono avvenire con complessità logaritmica e attraverso l'utilizzo di procedure alquanto semplici. Su di essi è anche possibile effettuare elaborazioni di tipo sequenziale dell'archivio primario senza alcuna necessità di sottoporlo a riorganizzazione.

3.3 Varianti del B-Tree

Esistono diverse varianti al B-Tree. Le tre più diffuse sono:

- Il **B+Tree** (quello che vedremo)
- Il B*Tree
- Il prefix B-Tree
- Il predictive B+Tree

4 B^+TREE INDEX FILES

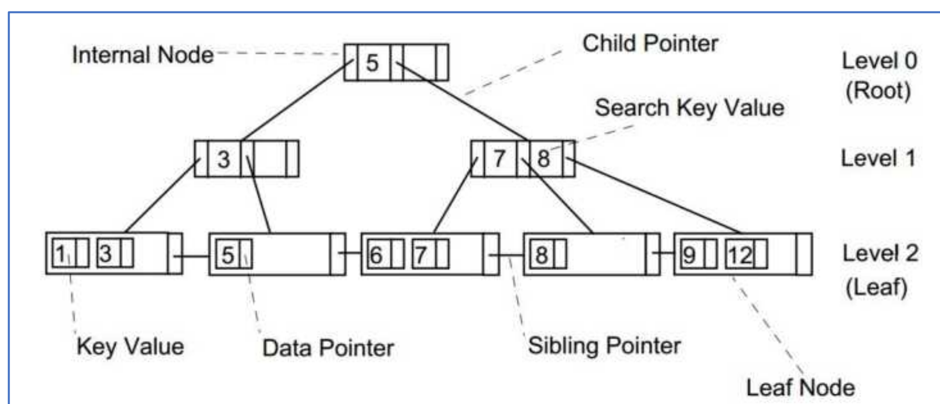
A differenza del *B-Tree*, nel **B+Tree:**

- Tutti i dati sono salvati nelle foglie.
- I nodi interni contengono solamente chiavi e puntatori.
- Tutte le foglie sono allo stesso livello.
- I nodi foglia sono collegati assieme come una lista per rendere il recupero di informazioni più semplici.
  ↳ Tale collegamento consente di svolgere in maniera efficiente anche interrogazioni su un intervallo di valori ammissibili.
- Il numero massimo di chiavi in un record è detto ordine R del *B+Tree*.
- Il numero minimo di chiavi per record è R/2.

Il numero di chiavi che può essere indicizzato utilizzando un B+Tree è in funzione di R e dell'altezza dell'albero. Per un B+Tree di ordine n-esimo e di altezza h:

- Il massimo numero di chiavi è n^h
- Il minimo numero di chiavi è $2(n/2)^{h-1}$

Di tutte le varianti del B-Tree, questa è la più usata, perché tutti i primi nodi interni che la memoria centrale può contenere vengono mantenuti su di essa mentre il resto dei nodi e le foglie vengono lasciate su memoria di massa. Ciò permette una maggior velocità di ricerca (questo tipo di struttura trova applicazione nei file system Journaled File System, HPFS, Be File System, ReFS, ReiserFS e XFS).



4.1 PERCHÉ USARE UN B^+TREE

Esistono, come abbiamo visto, alcuni svantaggi nell'utilizzare file sequenziali indicizzati:

- le prestazioni si riducono con l'aumentare delle dimensioni del file, a causa della creazione di molti *overflow blocks*
- è necessaria una riorganizzazione periodica dell'intero file

Esistono quindi dei particolari *idex files* detti *B^+Tree Index Files* che presentano a questo proposito alcuni vantaggi:

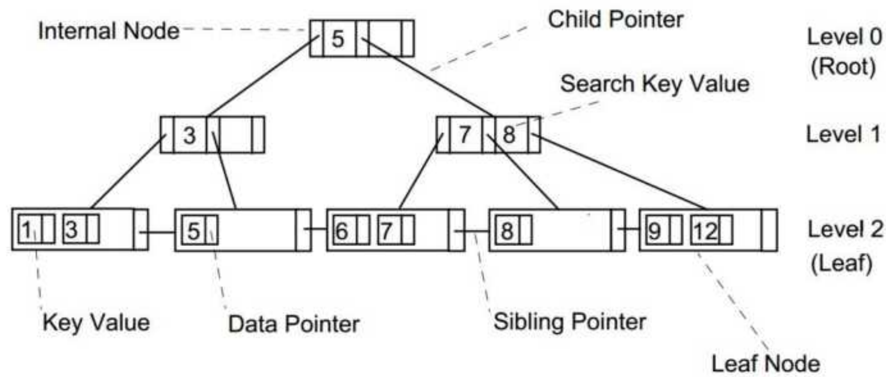
- si riorganizzano automaticamente, in seguito ad inserzioni e cancellazioni, con piccole modifiche locali
- non è richiesta la riorganizzazione dell'intero file per mantenere le prestazioni

Esistono alcuni svantaggi dei B^+*Tree*, comunque minori, tra cui:

- un *overhead* per inserimenti e cancellazioni extra
- *space overhead*

I vantaggi dei B^+*Trees* superano gli svantaggi → sono ampiamente utilizzati, sono una valida alternativa ai file sequenziali indicizzati.

4.2 STRUTTURA DEL B^+TREE

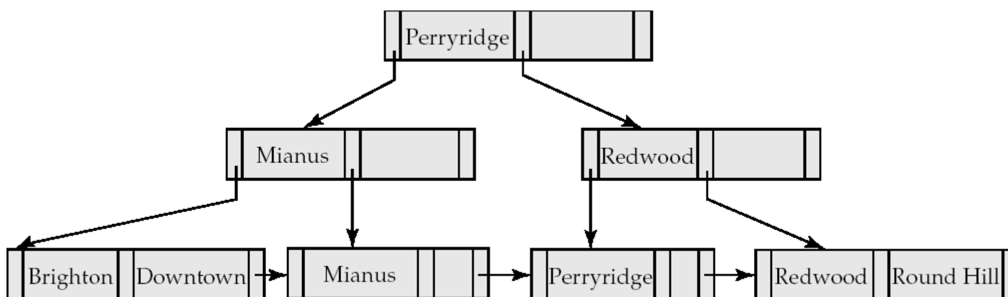


Un B^+-*Tree* è un albero con radice (*rooted tree*) che soddisfa le seguenti proprietà

- Tutti i percorsi (*path*) dalla radice (*root*) alla foglia (*leaf*) hanno la stessa lunghezza
- Ogni nodo (*node*) che non è una radice o una foglia ha tra gli $\lceil n/2 \rceil$ ed i n nodi figli (*childs*).
- Un nodo foglia ha valori tra $\lceil (n-1)/2 \rceil$ e $\text{n}-1$

Casi notevoli:

- se la radice non è una foglia, ha almeno 2 figli.
- se la radice è una foglia (ovvero, non ci sono altri nodi nell'albero), può avere valori tra 0 e $\text{n}-1$.



4.3 STRUTTURA DEL NODO B^+TREE

Un nodo ha tipicamente la seguente struttura:

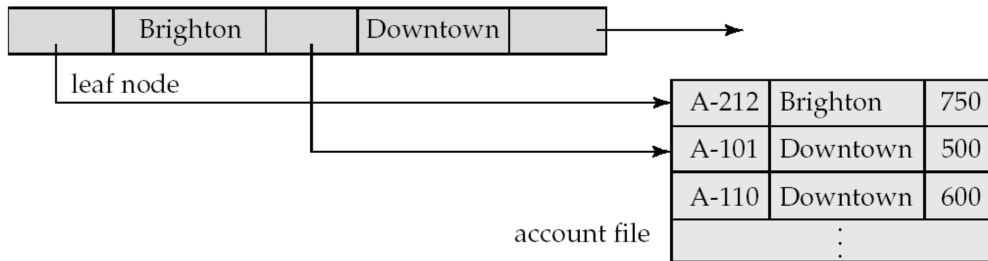
1. K_i sono i *search key values*
2. P_i sono puntatori ai *childs* (per nodi non foglia) o puntatori a *record* o *bucket* di *record* (per nodi foglia)

I *search key values* in un nodo sono ordinati: $K_1 < K_2 < \cdots < K_{n-1}$

4.4 NODI FOGLIA NEI B^+TREE

Proprietà di un nodo foglia:

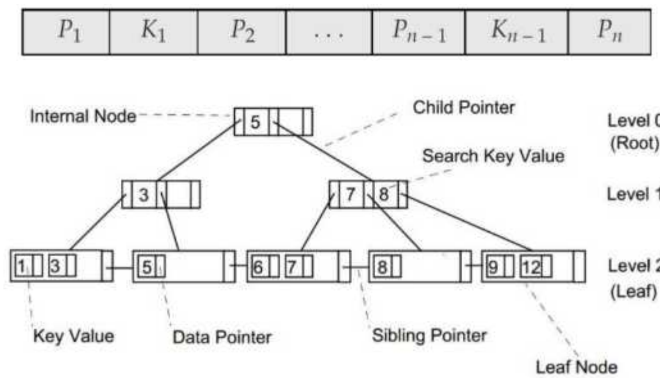
- Per $\text{i} = 1, 2, \dots, \text{n}-1$, il puntatore P_i punta al *record* di un file avente il valore della *search key* K_i, o un *bucket* di puntatori a *record* di file, ogni *record* con il valore della *search key* K_i. Serve una struttura a *bucket* solo se la *search-key* non è una chiave primaria.
- Se $\text{L}_\text{i}, \text{L}_\text{j}$ sono nodi foglia e $\text{i} < \text{j}$, i *search key values* di L_i sono meno dei *search key values* di L_j
- P_n punta al prossimo nodo foglia seguendo l'ordine delle *search key*



4.5 Nodi non foglia nei B^+Tree

I nodi non foglia formano un *multi-level sparse index* sui nodi foglia. Infatti, preso un nodo non foglia con m puntatori:

- Tutte le *search key* nel sotto-albero a cui P_1 punta hanno valori $< K_1$
- (per $2 \leq i \leq n-1$) Tutte le *search key* nel sotto-albero a cui P_i punta hanno valori $\geq K_{i-1}$ e $< K_i$
- Tutte le *search key* nel sotto-albero a cui P_n punta hanno valori $\geq K_{n-1}$



Vediamo due esempi di B+Tree:

Example of a B+-tree

B+-tree for *account* file ($n = 3$)

B+-tree for *account* file ($n = 5$)

Leaf nodes must have between 2 and 4 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 5$).

Non-leaf nodes other than root must have between 3 and 5 children ($\lceil n/2 \rceil$ and n with $n = 5$).

Root must have at least 2 children.

4.6 Osservazioni su B^+Tree

Poiché le connessioni tra nodi sono fatte da puntatori, i blocchi "logicamente" vicini non devono necessariamente essere "fisicamente" vicini.

I livelli non-foglia del *B+Tree* formano una gerarchia di *sparse indexes*.

Il *B+Tree* contiene un numero relativamente piccolo di livelli:

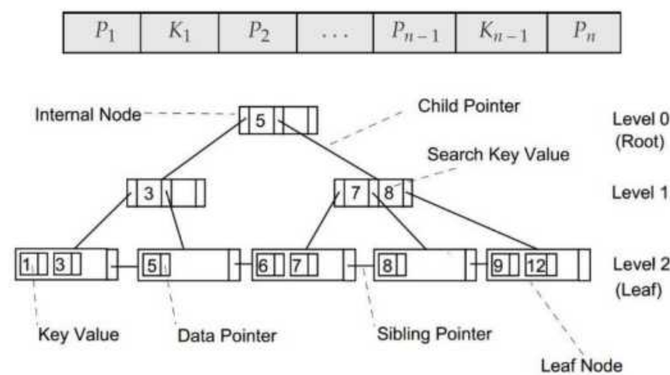
- Il livello sotto la radice ha almeno $2 * \lceil n/2 \rceil$ valori;
- Il livello successivo ha almeno $2 * \lceil n/2 \rceil \lceil n/2 \rceil$ valori; ecc…
- Se nel file sono presenti K *search-key values*, l'altezza dell'albero è non più di $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  ↳ Le ricerche possono così essere eseguite in modo efficiente.

È possibile gestire inserimenti e cancellazioni sul file principale in modo efficiente, poiché l'indice può essere ristrutturato in tempo logaritmico (come vedremo).

4.7 QUERY SU B^{+}TREES

Trova tutti i record con un *search-key value* che sia k:

1. $N = root$
2. Ripeti i seguenti tre passaggi fino a quando N non è un nodo foglia:
   a. Cercare in N il più piccolo *search key value* $> k$
   b. Se tale valore esiste, supponiamo che sia K_i . Quindi si imposta $N = P_i$
   c. Altrimenti $k \geq K_{n-1}$ e allora imposta $N = P_n$
3. Se per un determinato i si ha $K_i = k$, seguire il puntatore P_i al *record* desiderato o al *bucket*.
4. Altrimenti non esiste un record con il *search-key value* che sia k.



Se nel file sono presenti K *search-key values* nel file, l'altezza dell'albero non è altro che $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.

Un nodo ha generalmente le stesse dimensioni di un blocco del disco (tipicamente 4 kilobyte) e n è in genere circa 100 (40 byte per ogni *index entry*, voce di indice)

Facciamo un esempio: con 1 milione di *search key values* e $n = 100$ abbiamo al massimo $\log_{50}(1.000.000) = 4$ nodi accessibili in un *lookup* (consultazione del DB). Confrontiamolo con un albero binario bilanciato con 1 milione di *search key values:* è possibile accedere a circa 20 nodi in una ricerca → la differenza è significativa poiché per ogni accesso al nodo potrebbe essere necessario un I/O del disco, che costa circa 20 millisecondi.

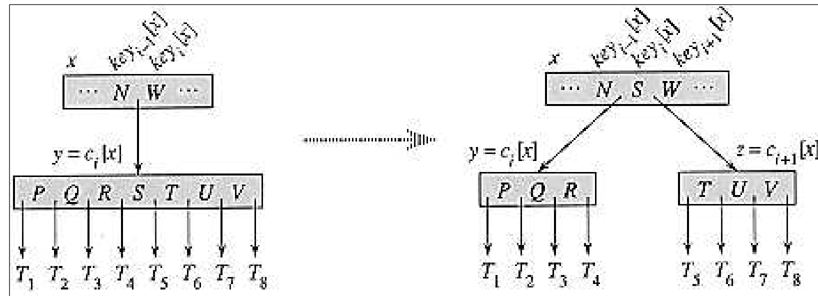
4.8 AGGIORNAMENTI SU B^{+}TREES

Inserzione:

- Trova il nodo foglia in cui dovrebbe apparire il *search-key value*
- Se il *search-key value* è già presente nel nodo foglia
  - aggiungi il *record* al file
- Se il *search-key value* non è presente, allora
  - aggiungi il *record* al file principale (e crea un *bucket* se necessario)
  - se c'è spazio nel nodo foglia, inserire la coppia (*search-key value*, puntatore) nel nodo foglia
  - altrimenti, ***split the node*** (dividi il nodo), assieme alla nuova *entry* (*search-key value*, puntatore)

Divisione (*splitting*) **di un nodo foglia**:

- Prendi, in ordine, le n coppie (*search-key value*, puntatore), inclusa quella da inserire. Inserisci le prime $\lceil n/2 \rceil$ nel nodo originale e il resto in un nuovo nodo.
- Sia p il nuovo nodo, e k il *search key value* minimo in p. Inserisci (k, p) nel genitore del nodo che viene diviso.
- Se il genitore è pieno, dividi anche lui e propaga ulteriormente lo *splitting*.



Vediamo ora esempi di inserzioni (http://condor.depaul.edu/ntomuro/courses/417/notes/lecture3.html):

At the leaf node,
1. Look at each element from the beginning. If X is found, terminate (duplicate element).
2. If there is room in the leaf, insert X in the appropriate position. This may require shifting elements.

Insert 25

3. If leaf is full, split it into 2 leaves, and copy and insert the middle element into the (internal) parent node (which may require shifting internal elements).

Insert 6 (causes leaf split)

4. If leaf's parent is full, split the parent into 2 internal nodes, and copy and insert the middle element into its parent node.

After inserting 27, 28, 29

Insert 30 (causes internal split)

La divisione dei nodi procede verso l'alto finché non viene trovato un nodo che non è pieno. Nel peggiore dei casi il nodo radice può essere diviso aumentando l'altezza dell'albero di 1.

Come abbiamo visto nell'esempio precedente può avvenire uno ***splitting* di un nodo non foglia**, quando si inserisce (k, p) in un nodo interno N che è pieno:

- Copia N in un'area di memoria M con spazio per $n + 1$ puntatori e n chiavi
- Inserisci (k, p) in M
- Copia $P_1, K_1, \dots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ da M di nuovo nel nodo N
- Copia $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \dots, K_n, P_{n+1}$ da M a un nuovo nodo allocato N'
- Inserisci $(K_{\lceil n/2 \rceil}, N')$ nel genitore N

Cancellazione:

- Trova il record da e rimuovilo dal file principale e dal *bucket* (se presente)
- Rimuovi la coppia (*search-key value*, puntatore) dal nodo foglia se non c'è un *bucket* o se il *bucket* è diventato vuoto
- Se il nodo ha troppo poche *entries* (voci) a causa della rimozione, e le *entries* nel nodo e un nodo "fratello" (*sibling*) possono stare in un singolo nodo, bisogna ***merge the siblings***, "unire i fratelli"
- Altrimenti, se il nodo ha troppe poche *entries* a causa della rimozione, ma le *entries* nel nodo e in un nodo "fratello" (*sibling*) non potrebbero stare in un singolo nodo, bisogna **ridistribuire i puntatori**.
- Le cancellazioni del nodo possono propagarsi verso l'alto fino a un nodo che ha $\lceil n/2 \rceil$ o più puntatori viene trovato.
- Se il nodo radice ha un solo puntatore dopo la cancellazione, viene eliminato e l'unico figlio diventa la radice.

Vediamo in dettaglio cosa vuole dire ***merge the siblings:***

- Inserire tutti i *search key values* dei due nodi in un singolo nodo (quello a sinistra) ed eliminare l'altro nodo.
- Eliminare la coppia (K_{i-1}, P_i), dove P_i è il puntatore al nodo eliminato, dal suo genitore, e farlo ricorsivamente usando la procedura precedente.

Invece effettuare la **redistribuzione dei puntatori** significa**:**

- Ridistribuire i puntatori tra il nodo e un "fratello" (*sibling*) tale che entrambi abbiano più del numero minimo di *entries*.
- Aggiornare il *search key value* corrispondente nel nodo genitore.

Vediamo ora da (http://www.euroinformatica.ro/documentation/programming/!!!Algorithms_CORMEN!!!/DDU0111.html):

1. If the key k is in node x and x is a leaf, delete the key k from x.
2. If the key k is in node x and x is an internal node, do the following.
   a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be performed in a single downward pass.)

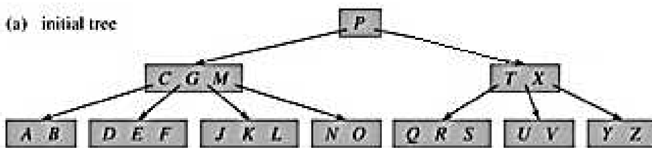
   b. Symmetrically, if the child z that follows k in node x has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be performed in a single downward pass.)

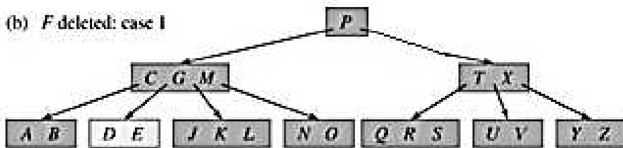
   c. Otherwise, if both y and z have only $t - 1$ keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains $2t - 1$ keys. Then, free z and recursively delete k from y.
3. If the key k is not present in internal node x, determine the root $c_i[x]$ of the appropriate subtree that must contain k, if k is in the tree at all. If $c_i[x]$ has only $t - 1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then, finish by recursing on the appropriate child of x.
   a. If $c_i[x]$ has only $t - 1$ keys but has an immediate sibling with at least t keys, give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$, moving a key from $c_i[x]$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $c_i[x]$.

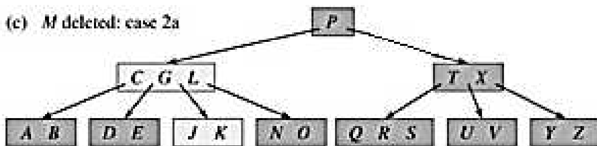
   b. If $c_i[x]$ and both of $c_i[x]$'s immediate siblings have $t - 1$ keys, merge $c_i[x]$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

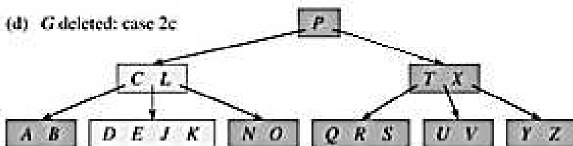
Deleting keys from a B-tree. The minimum degree for this B-tree is t = 3, so a node (other than the root) cannot have fewer than 2 keys. Nodes that are modified are lightly shaded.

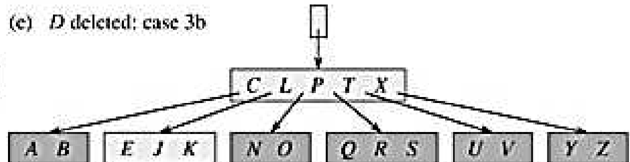
(a) The B-tree
(b) Deletion of F. This is case 1: simple deletion from a leaf.
(c) Deletion of M. This is case 2a: the predecessor L of M is moved up to take M's position.
(d) Deletion of G. This is case 2c: G is pushed down to make node DEGJK, and then G is deleted from this leaf (case 1).
(e) Deletion of D. This is case 3b: the recursion can't descend to node CL because it has only 2 keys, so P is pushed down and merged with CL and TX to form CLPTX; then, D is deleted from a leaf (case 1).
(e') After (d), the root is deleted and the tree shrinks in height by one.
(f) Deletion of B. This is case 3a: C is moved to fill B's position and E is moved to fill C's position.

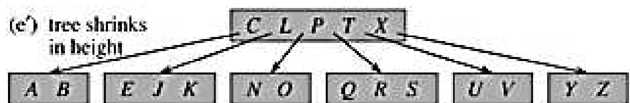


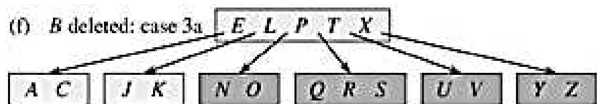












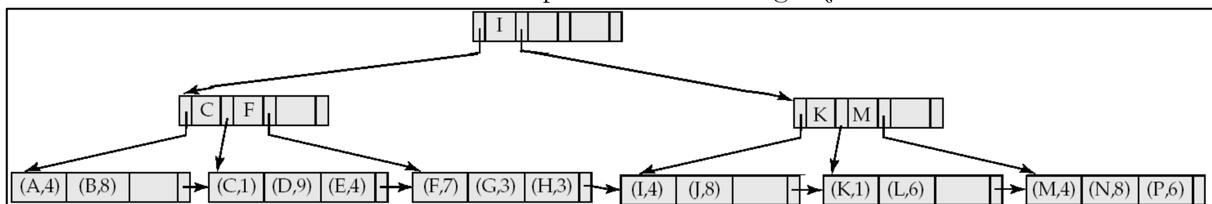
4.9 B^+TREE FILE ORGANIZATION

Il problema della degradazione dell'*index file* viene risolto utilizzando gli indici B+Tree. Il problema di degradazione dei ***data files*** viene risolto utilizzando ***B^+ Tree File Organization***. In essi i nodi foglia memorizzano *record*, invece che puntatori. Rimane però il requisito che i nodi foglia debbano essere "mezzi pieni".

Essendo i *record* più grandi dei puntatori, il numero massimo di *record* che possono essere memorizzati in un nodo foglia è inferiore al numero di puntatori in un nodo non foglia.

L' inserimento e la cancellazione vengono gestiti allo stesso modo dell'inserimento e cancellazione di *entries* (voci) in un indice B^+Tree.

Vediamo un esempio di B^+*Tree File Organization:*



In questo caso un buon utilizzo dello spazio è importante dal momento che i *record* utilizzano più spazio dei puntatori.

Per migliorare l'utilizzo dello spazio, si possono utilizzare più nodi di pari livello per delle ridistribuzioni durante i *merge* e gli *splitting* → utilizzare 2 nodi "fratelli" (*siblings*) per una ridistribuzione (per evitare *split*/*merge* dove possibile) permette di avere in ogni nodo almeno $\lfloor 2n/3 \rfloor$ inserimenti.

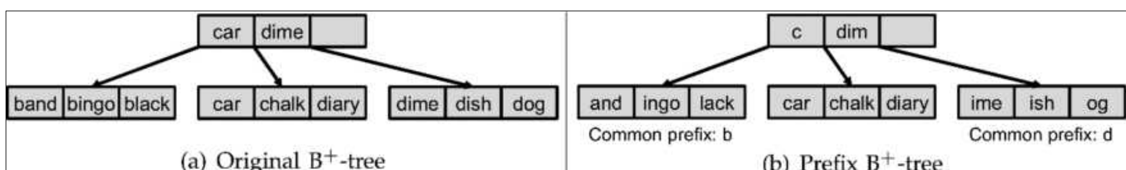
4.10 STRINGHE DI INDICIZZAZIONE

Utilizzo di stringhe di lunghezza variabile come chiavi:

- *Fanout* (apertura, "sparpagliamento") variabile.
- Adottare come criterio per *splitting* lo spazio utilizzato, non il numero di puntatori.

Prefix compression:

- I *key values* nei nodi interni possono essere dei prefissi della chiave completa (*full key*)
  - Bisogna mantenere abbastanza caratteri per distinguere le voci (*entries*) nei sottoalberi separati dai *key values* (Ad esempio "Silas" e "Silberschatz" possono essere separati da "Silb")
- Le *keys* nel nodo foglia possono essere compresse visto che condividono un prefisso comune.

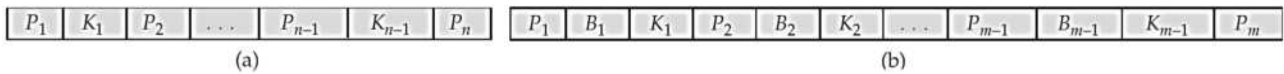


5 B-TREE INDEX FILES

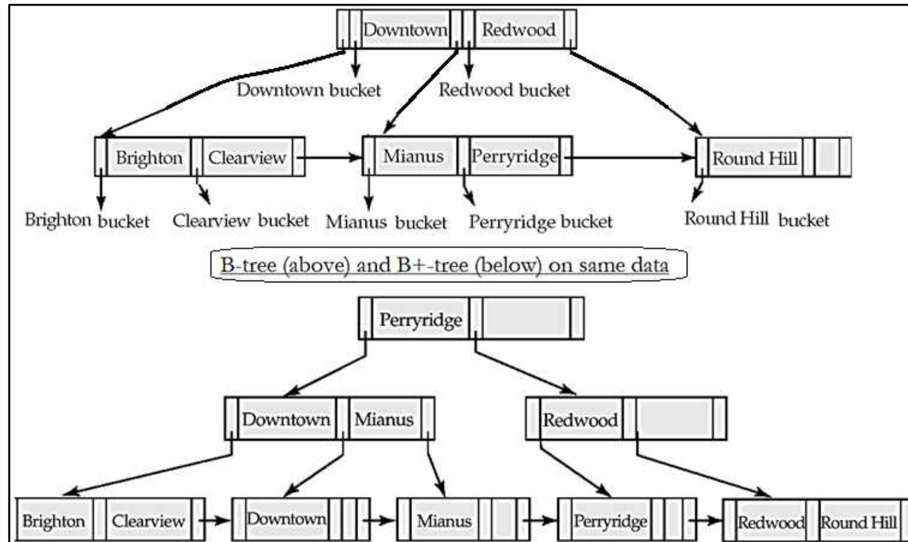
Simili ai B+tree, i B-tree consentono però che i *search key value* appaiano solo una volta → viene eliminato il problema l'archiviazione ridondante di *search key*.

Le *search key* nei nodi non foglia non appaiono in nessun altro punto del B-albero → deve essere incluso un puntatore aggiuntivo per ogni *search key* in un nodo non foglia.

Vediamo la differenza tra un nodo foglia (a) ed un nodo non foglia (b) di un B-alberi (I puntatori B_i sono i puntatori a *bucket* o a *file record*):



Vediamo la differenza tra un approccio B+tree e uno B-tree:



5.1 Vantaggi e svantaggi di un *B*-Tree

Vantaggi degli indici B-Tree:

- Si possono utilizzare meno nodi rispetto a un B+Tree corrispondente
- A volte è possibile trovare il *search key value* prima di raggiungere nodo foglia

Svantaggi degli indici B-Tree:

- Solo una piccola parte di tutti i *search key value* viene trovata in anticipo
- I nodi non foglia sono più grandi, quindi il *fanout* è ridotto. Quindi i B-alberi hanno in genere una profondità (altezza) maggiore rispetto corrispondente B+Tree
- Inserimento e cancellazione sono più complicati rispetto a B+Tree
- L' implementazione è più difficile rispetto a B+Tree

In genere, i vantaggi dei B-Tree non ne superano gli svantaggi.

6 Indicizzazione: caratteristiche avanzate

6.1 Multiple-key e Composite-key

Il ***multiple-key access*** ("accesso a più chiavi") consiste nell'utilizzare più indici per determinati tipi di query. Ad esempio:

select *account_number*
from *account*
where *branch_name* = "Perryridge" **and** *balance* = 1000

Possibili strategie per l'elaborazione della query utilizzando indici su singoli attributi:

1. Usare l'indice su *branch_name* per trovare gli account con *branch_name*="Perryridge", verificare poi *balance*=1000
2. Usare l'indice su *balance* per trovare gli account con *branch_name*=1000, verificare poi *branch_name*="Perryridge"

3. Usare l'indice *branch_name* per trovare i puntatori a tutti i *record* appartenente al ramo "Perryridge". Allo stesso modo utilizzare l'indice su *balance*. Prendere poi l'intersezione di entrambi i gruppi di puntatori ottenuti.

Le ***Composite search keys*** sono *search keys* che contengono più di un attributo
→ ad esempio: (Branch_name, balance)

[**Lexicographic ordering**: $(a_1, a_2) < (b_1, b_2)$ se $[a_1 < b_1]$ oppure $[(a_1 = b_1) \wedge (a_2 < b_2)]$]

Supponiamo di avere un indice sulla *combined search-key* (Branch_name, balance)

- Per la condizione in SQL seguente `(where branch_name="Perryridge" and balance=1000)` l'indice (Branch_name, balance) può essere utilizzato per recuperare solo *record* che soddisfano entrambe le condizioni.
  - Usare indici separati è molto meno efficiente, si potrebbero infatti recuperare molti *record* (o puntatori) che soddisfano solo una delle condizioni.
- L'indice (Branch_name, balance) può anche gestire in modo efficiente la condizione in SQL seguente `(where branch_name="Perryridge" and balance<1000)`
- L'indice (Branch_name, balance) <u>non</u> può però gestire in modo efficiente la seguente condizione `(where branch_name<"Perryridge" and balance=1000)`
  - Potrebbe recuperare molti *record* che soddisfano la prima condizione ma non la seconda

6.2 *SEARCH KEYS* NON UNIVOCHE

Se una relazione ha più di un *record* contenente lo stesso *search key value* (ovvero, due record o più che hanno gli stessi valori negli attributi indicizzati), si dice che la *search key* è una ***nonunique search key***.

Un problema con le *search keys* non univoche è nell'efficienza dell'eliminazione dei record. Supponiamo che un particolare *search key value* sia presente un numero elevato di volte e che uno dei *record* con quella *search key* debba essere cancellato. La cancellazione potrebbe dover cercare attraverso un gran numero di voci (*entries*), potenzialmente attraverso più nodi foglia, per trovare la voce (*entry*) corrispondente al particolare *record* che deve essere eliminato.

Vediamo alcune soluzioni alternative:

- Creare un *Bucket* su un blocco separato (ma è una cattiva idea)
- Creare una lista di *tuple pointers* con rispettive *keys*
  - Basso *space overhead*, nessun costo aggiuntivo per le query
  - Codice extra per gestire la lettura/l'aggiornamento di lunghe liste
  - La cancellazione di una tupla può però essere comunque costosa in termini computazionali se ci ne sono molti duplicati sulla *search key*
- Si può rendere unica la *search key*

Una semplice soluzione a questo problema, utilizzata dalla maggior parte dei sistemi di database, è l'ultima alternativa. Si può infatti realizzare delle *search keys* univoche mediante la creazione di una <u>*composite search key*</u> contenente l'originale *search key* ed un altro attributo, che assieme sono unici tra tutti i *record*. L'attributo extra è chiamato ***uniquifier attribute***. Quando un *record* è da cancellare, il *composite search key value* viene estrapolato dal record e poi usato per cercare l'indice. Poiché il valore è univoco, la voce corrispondente del livello foglia può essere trovata con un singolo attraversamento da radice a foglia, senza ulteriore accessi a livello foglia. Di conseguenza, la cancellazione dei record può essere eseguita in modo efficiente.

6.3 ALTRI PROBLEMI NELL'INDICIZZAZIONE

Covering indices ("indici di copertura"):

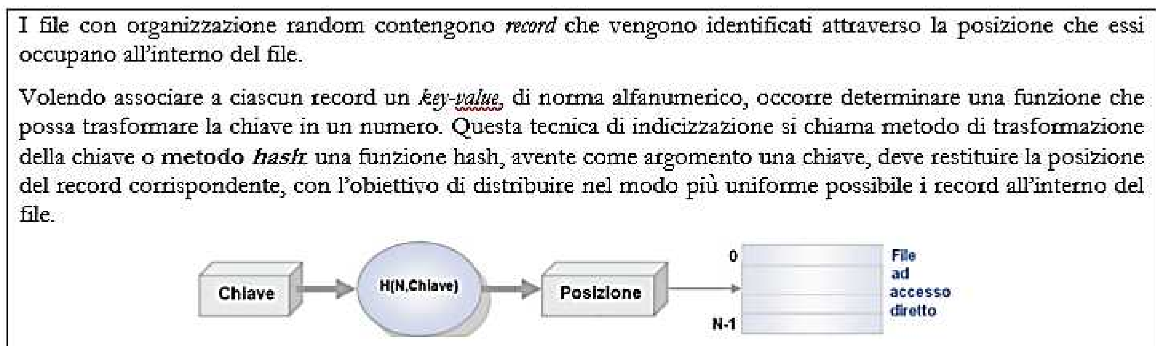
- Aggiungere ulteriori attributi all'indice in modo che (alcune) query possano evitare il *fetching* di *record* attuali → particolarmente utile per indici secondari (perché? Prova a darti una risposta).

- Grazie ad essi è inoltre possibile memorizzare attributi aggiuntivi su un solo nodo foglia.

Record relocation (*ri-locazione*) e indici secondari:

- Se si sposta un record, tutti gli indici secondari che memorizzano i puntatori al *record* devono essere aggiornati
- Lo *splitting* dei nodi in *B+tree file organizations* diventa molto costoso
  ↳ Soluzione: utilizzare la *search key* dell'indice primario anziché il puntatore al *record* nell'indice secondario
  - Vi è attraversamento extra dell'indice primario per localizzare il *record* → maggiore costo per le query, ma gli *splitting* dei nodi sono economici
  - Aggiungere un *record-ID* se la *search key* dell'indice primario non è univoca

7 HASHING STATICO



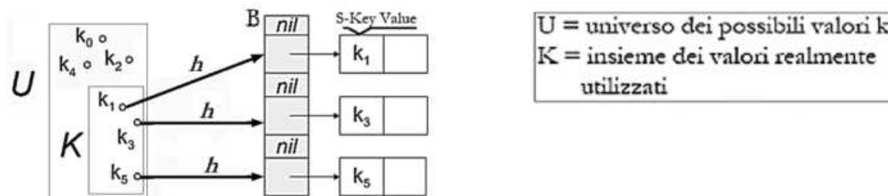
I file con organizzazione random contengono *record* che vengono identificati attraverso la posizione che essi occupano all'interno del file.

Volendo associare a ciascun record un *key-value*, di norma alfanumerico, occorre determinare una funzione che possa trasformare la chiave in un numero. Questa tecnica di indicizzazione si chiama metodo di trasformazione della chiave o metodo ***hash***: una funzione hash, avente come argomento una chiave, deve restituire la posizione del record corrispondente, con l'obiettivo di distribuire nel modo più uniforme possibile i record all'interno del file.

Un ***bucket*** è un'unità di *storage* (archiviazione) che contiene uno o più *record* (è in genere un *block* del disco)

In una ***hash file organization*** otteniamo direttamente il *bucket* di un *record* dal suo *search-key value* usando una **funzione *hash:***

– La funzione *hash* h è una funzione $h: K \rightarrow B$. [1]



– Se $|K| \sim |U|$ → si utilizzano delle tabelle ad indirizzamento diretto

– Se $|K| \ll |U|$ → non conviene utilizzare delle tabelle ad ind. diretto, è soluzione non praticabile:
  - Ad esempio: *student* con chiave *student_ID*. Se il numero di matricola *student_ID* ha 6 cifre, l'array deve avere spazio per contenere 10^6 elementi. Ma se gli studenti del corso sono ad esempio 30, lo spazio realmente occupato dalle chiavi memorizzate è $\frac{30}{10^6} = 0.00003 = 0.003\%$!!!

– La funzione *hash* viene utilizzata per localizzare i *record* per l'accesso, l'inserimento e la cancellazione

Record con *search-key value* diversi possono essere associati a stesso *bucket*; quindi l'intero *bucket* deve essere analizzato sequenzialmente per trovare un *record*.

---

[1] $K =$ Insieme di tutti i *search-key values* utilizzati ($U =$ universo di tutte le possibili chiavi)
$B =$ Insieme di tutti i *bucket addresses*

Funzioni di hash

Da Wikiversità, l'apprendimento libero.

Una funzione di hash h, o *message digest function*, è una funzione che mappa un messaggio arbitrariamente lungo in una stringa di lunghezza prefissata, cercando di far in modo che da questa stringa non si possa risalire al messaggio che l'ha generata. La lunghezza della stringa finale è direttamente correlata con la sicurezza della funzione di hash, perché più una stringa è lunga, minore sarà la probabilità di trovare due messaggi con lo stesso *digest*, con la stessa stringa finale.

La stringa $d = h(m)$ può essere vista come l'*impronta digitale* del messaggio, teoricamente irripetibile.

Ciò che si chiede da una funzione di hash è:

1. l'efficienza computazionale, perché il messaggio m potrebbe essere molto lungo;
2. la probabilità che accada $d = h(m)$ per un qualsiasi messaggio casuale m deve essere 2^{-n}, dove il digest è lungo n bit.
3. *one way property* o *preimage resistance*: dato il digest d, dev'essere computazionalmente impossibile calcolare il messaggio che l'ha generato;
4. dato il messaggio m tale per cui $d = h(m)$, deve essere computazionalmente impossibile trovare un altro messaggio m' tale per cui $h(m) = h(m') = d$;
5. deve essere computazionalmente impossibile trovare due messaggi m' ed m'' che collidono, qualunque sia il loro digest.

Teorema: *Paradosso dei gemelli, o* birthday problem

Dato un numero n di bit ed un digest in uno spazio di dimensione 2^n, quanti messaggi devono essere scelti per avere il 50% di possibilità di trovarne almeno 2 che collidono, cioè che hanno lo stesso digest? $2^{\frac{n}{2}}$

Vediamo un esempio di *hash file organization* del file *account*, utilizzando *branch_name* come chiave:

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

bucket 7

A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

bucket 9

Ci sono 10 *bucket*. Si supponga che il numero intero i sia la rappresentazione binaria dell'i-esimo carattere. Si porenda, ad esempio, una funzione *hash* che restituisce la somma delle rappresentazioni binarie dei caratteri in modulo 10, ad esempio:

$$h(Perryridge) = 5, \qquad h(Round\ Hill) = 3, \qquad h(Brighton) = 3$$

7.1 Funzioni *hash* ideali e non ideali

La **peggiore** funzione di *hash* possibile è quella che mappa tutti i *search-key values* nello stesso *bucket*; questo rende il tempo di accesso proporzionale al numero di chiavi di ricerca valori nel file.

Una funzione di *hash* **ideale** è **uniforme**, vale a dire che a ciascun *bucket* viene assegnato lo stesso numero di *search-key values* dall'insieme di tutti i valori possibili.

La funzione di *hash* **ideale** è **casuale**, vale a dire che ciascun *bucket* avrà lo stesso valore numero di *record* assegnati, indipendentemente dall'effettiva distribuzione di *search-key values* nel file.

Le tipiche funzioni di *hash* eseguono calcoli sulla rappresentazione binaria interna della *search-key*. Ad esempio, per una *search-key* di tipo *string*, le rappresentazioni binarie di tutti i caratteri nella stringa potrebbero essere sommate e la somma potrebbe essere restituita in modulo *n* (dove *n* è il numero di bucket).

7.2 GESTIRE I *BUCKET OVERFLOW*

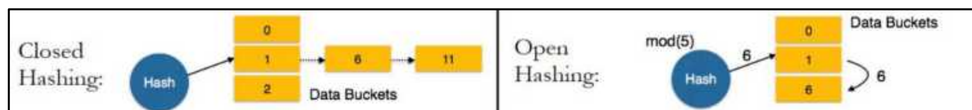
La condizione di *overflow* del *bucket* è nota come ***collision*** (collisione). Questo è uno stato fatale per qualsiasi funzione di hash statica.

Il ***Bucket Overflow*** può verificarsi a causa di:

- *Bucket* di capacità insufficiente
- Non uniformità nella distribuzione dei *record*. Questo può accadere a causa di due motivi:
  - più *record* hanno lo stesso *search-key value*
  - la funzione *hash* scelta produce una distribuzione non uniforme dei *search-key values*

In questi casi, è possibile utilizzare due metodi:

- ***Overflow Chaining*** - Quando i *bucket* sono pieni, un nuovo *bucket* (un ***overflow bucket***) viene allocato per lo stesso risultato di hash e viene collegato dopo il precedente in una lista linkata (*linked list*). Questo meccanismo è chiamato ***Closed Hashing***.
- ***Linear Projection*** - Quando una funzione hash genera un indirizzo al quale i dati sono già memorizzati, viene assegnato il successivo *bucket* libero. Questo meccanismo è chiamato ***Open Hashing***, e non è adatto ad applicazioni di Database.



Sebbene la probabilità di *collisions* (a.k.a *bucket overflows*) possa essere ridotta, non è possibile eliminarla; viene gestita utilizzando gli *overflow buckets*.

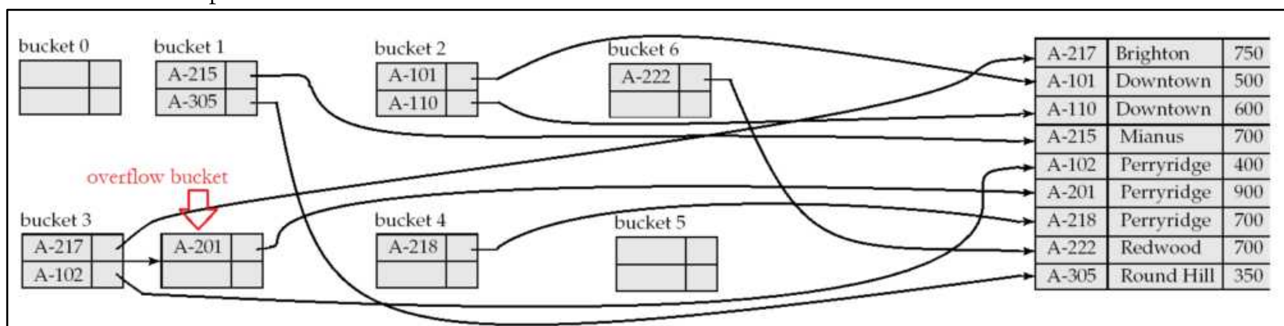
7.3 INDICI HASH

L' *hashing* può essere utilizzato non solo per l'organizzazione dei file, ma anche per la creazione della struttura degli indici (*index-structure creation*)

Un ***hash index*** (indice hash) organizza le *search-keys*, con i loro *record pointers* associati, in una *hash file structure*. A rigor di termini, gli indici *hash* sono sempre indici secondari:

- Se il file stesso è organizzato usando l'*hashing*, non è necessario un *primary hash index* su di esso utilizzando la stessa *search-key*.
- Tuttavia, usiamo il termine *hash index* per fare riferimento sia alle *secondary index structures*, sia agli *hash organized files*.

Vediamo un esempio di indice hash:



7.4 PROBLEMI (*DEFICIENCIES*) DELL'HASHING STATICO

Nell'*hashing* statico, la funzione h mappa i *search-key values* su un sottoinsieme fisso B di indirizzi del *bucket*. I database invece crescono o si riducono con il tempo:

- Se il numero iniziale di *bucket* è troppo piccolo e il file cresce, le prestazioni si ridurranno a causa dei troppi *overflow*.
- Se lo spazio è allocato prevedendo la successiva crescita, una quantità significativa di spazio verrà inizialmente sprecata (e i bucket saranno molto vuoti, *underfull*).
- Se il database si riduce, lo spazio verrà nuovamente sprecato (di nuovo *underfull*).

Una soluzione possibile è la riorganizzazione periodica del file con una nuova funzione di hash, ma è costoso, ed interrompe le normali operazioni → una soluzione migliore è il consentire la modifica dinamicamente del numero di *bucket*.

8 HASHING DINAMICO

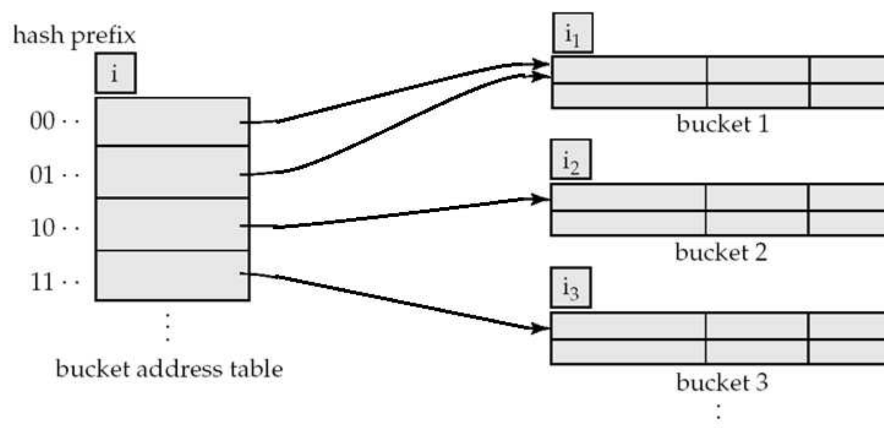
È un metodo buono per database le cui dimensioni si ingrandiscono e si restringono nel tempo. Consente la modifica della funzione di hash in modo dinamico.

Extendable hashing (hashing estendibile): è una forma di hashing dinamico:

- La funzione hash genera valori su un ampio intervallo - tipicamente b-bit interi, con $b = 32$.
- In qualsiasi momento utilizza solo un prefisso della funzione di hash per indicizzare in una tabella di indirizzi dei *bucket*.
- Supponiamo che la lunghezza del prefisso sia i bit, con $0 \leq i \leq 32$.
  - $Bucket\ address\ table\ size = 2^i$. Inizialmente $i = 0$
  - Il valore di i cresce e si riduce man mano che aumenta o si restringe la dimensione del database.
- Più voci nella tabella degli indirizzi del bucket possono puntare a un *bucket* (chiediti perché)
- Pertanto, il numero effettivo di *bucket* è $< 2^i$
  - Il numero di *bucket* cambia anche dinamicamente a causa di ***coalescing*** ("coalescenza"[2]) e ***splitting*** dei *bucket*.

8.1 STRUTTURA GENERALE *EXTENDABLE HASH*

Vediamo una *extendable hash structure* (struttura hash generale estendibile):



In questa struttura, $i_2 = i_3 = i$, mentre $i_1 = i - 1$ (vedremo dopo il perché)

[2] **Coalescènza** s.f. [der. del lat. *coalescens -entis*, part. pres. di *coalescĕre* «unirsi insieme»]. Unione, fusione, saldatura.

8.2 USO DELLA *EXTENDIBLE HASH STRUCTURE*

Ogni bucket j memorizza un valore i_j. Tutte le voci (*entries*) che puntano allo stesso *bucket* hanno gli stessi valori sui primi i_j bit.

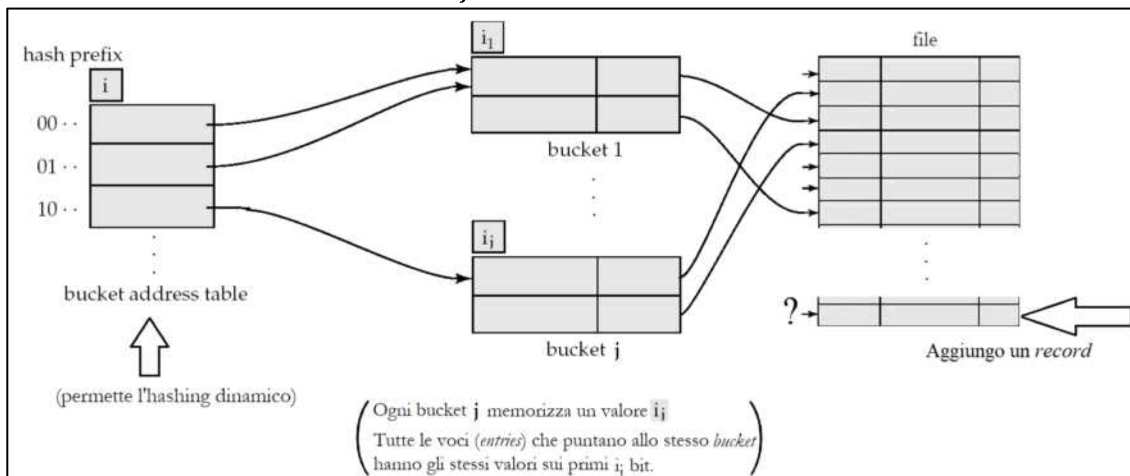
8.2.1 Ricerca di un *bucket*

Per **individuare il *bucket* contenente** la *search-key* $\boldsymbol{K_j}$:

1. Calcola $h(K_j) = X$
2. Usa i primi i bit (di ordine superiore) di X come uno spostamento nella *bucket address table* e segui il puntatore al *bucket* appropriato

8.2.2 Inserimento di un *record*

Per **inserire un *record* con** il *search-key value* $\boldsymbol{K_j}$:



- Seguire la stessa procedura della ricerca sopradescritta e individuare il *bucket*, ad esempio j.
- Se c'è spazio nel *bucket* j → inserire il record nel *bucket*.
- Se non c'è spazio nel *bucket* j → il secchio deve essere diviso (*splitting*) e l'inserimento poi ri-tentato → (**) (vedi sotto)
- In alcuni casi, invece, sempre per inserire un *record* con il *search-key value* K_j, vengono utilizzati gli o*verflow buckets* (vedi sotto)

↳ ***splitting* di un *bucket* j** mentre si inserisce un *record* con il *search-key value* K_j:

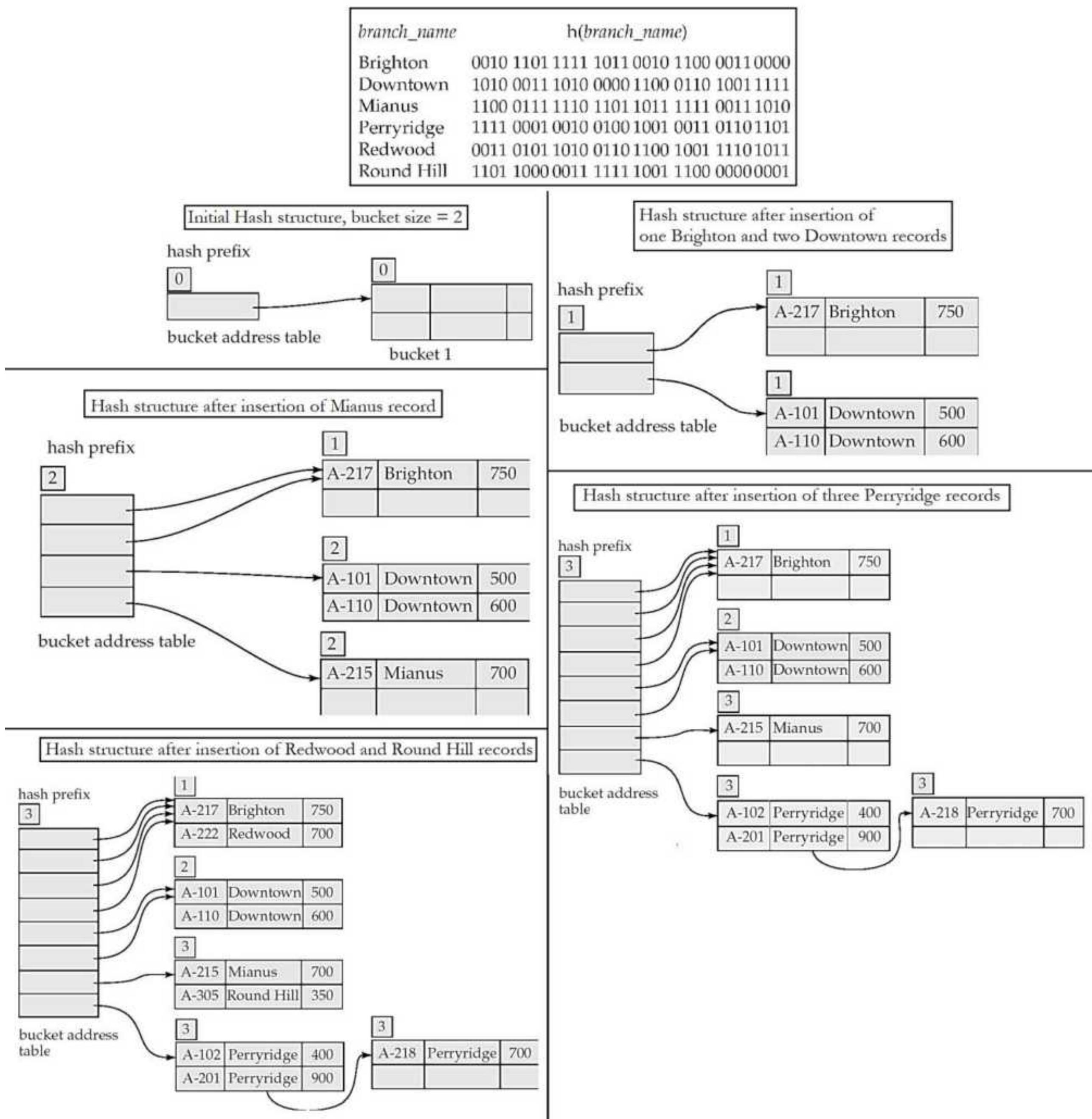
- Se $i > i_j$ (più di un puntatore al *bucket* j):
  - Allocare un nuovo bucket z e impostare $i_j = i_z = (i_j + 1)$
  - Aggiornare la seconda metà delle *entries* della *bucket address table* che originariamente puntavano a j, facendole puntare a z
  - Rimuovere ciascun *record* nel *bucket* j e reinserirlo (in j o z)
  - Ricalcola il nuovo *bucket* per K_j e inserisci record nel bucket (la divisione è ulteriormente necessaria se il *bucket* risulta ancora pieno)
- Se $i = i_j$ (solo un puntatore al *bucket* j):
  - Se i raggiunge un certo limite b, o sono avvenuti troppi *splitting* per questo inserimento, bisogna creare un *overflow bucket*
  - Altrimenti
    - incrementare i e raddoppiare la dimensione della *bucket address table*
    - sostituire ciascuna voce (*entry*) della tabella con due voci (*entries*) che puntano allo stesso *bucket*
    - ricalcola la nuova voce (*entry*) della *bucket address table* per K_j.
    - Adesso $i > i_j$ quindi prosegue come nel primo caso sopra descritto

8.2.3 Cancellazione di un *key-value*

Per eliminare un *key-value*:

- Ricercare il *bucket* corrispondente e rimuovere il *key-value*
- Il *bucket* stesso può essere rimosso se diventa vuoto (con aggiornamenti appropriati alla *bucket address table*)
- La coalescenza di secchi può essere eseguita (si possono però coalizzare solo due "*buddy" buckets* aventi lo stesso valore di i_j e lo stesso $i_j - 1$ prefisso, se presente)
- È anche possibile ridurre la dimensione della *bucket address table*
    - Nota: la riduzione della dimensione della *bucket address table* è un'operazione costosa e dovrebbe essere fatta solo se il numero di *bucket* diventa molto più piccolo delle dimensioni della tabella

8.3 ESEMPIO DI USO DELLA STRUTTURA HASH ESTENDIBILE



8.4 CONFRONTO *EXTENDIBLE HASHING* CON AD ALTRI SCHEMI

Vantaggi dell'*hash* estendibile:

- Le prestazioni dell'*hash* non peggiorano con l'aumento di dimensioni del file
- Minimo ingombro dello spazio

Svantaggi dell'*hash* estendibile:

- Si attraversa un livello indiretto in più per trovare il *record* desiderato
- La *bucket address table* può diventare essa stessa molto grande (più grande della memoria)
  - Non è possibile allocare aree molto grandi come contigue sul disco
  - Soluzione: *B+-tree file organization* per memorizzare la *bucket address table*
- La modifica della dimensione della *bucket address table* è un'operazione costosa

Il ***linear hashing*** è un meccanismo alternativo: consente la crescita incrementale della sua *directory* (equivalente alla *bucket address table*) al costo di più *bucket overflows*.

8.5 CONFRONTO TRA L'*ORDERED INDEXING* E L'HASHING

Abbiamo visto diversi schemi di indicizzazione ordinata e diversi schemi di *hashing*. Possiamo organizzare file di *record* come file ordinati utilizzando una *index-sequential organization* o una *B+-tree organizations*.. In alternativa, possiamo organizzare i file usando l'*hashing*. [3]

Ogni schema ha i suoi vantaggi in determinate situazioni. Nell'implementare un DB si potrebbero fornire molti schemi, lasciando la decisione finale di quali schemi usare al progettista del database. Tuttavia, tale approccio richiede che l'implementatore scriva più codice, aumentando sia il costo del sistema sia lo spazio occupato dal sistema stesso. La maggior parte dei sistemi di DB supporta i *B+-trees* e può supportare qualche forma di *hash file organization* o *hash indices*.

Per scegliere *file organization* e tecniche di indicizzazione per una relazione, l'implementatore o il progettista del database devono considerare i seguenti problemi:

- Il costo della riorganizzazione periodica dell'indice o dell'organizzazione hash è accettabile?
- Qual è la frequenza relativa di inserimento e cancellazione?
- È auspicabile ottimizzare il tempo di accesso medio a scapito dell'aumento il tempo di accesso peggiore?
- Quali tipi di query è probabile che gli utenti pongano?

Per quanto riguarda il tipo di query previsto, l'***hash*** è generalmente migliore nel *fetching* dei *record* che hanno un *key-value* specificato, se invece le *range queries* (query di intervallo) sono usate spesso, devono essere preferiti gli **indici ordinati**

- In pratica:
  - *PostgreSQL* supporta indici *hash*, ma scoraggia l'uso a causa di scarse prestazioni
  - *Oracle* supporta l'organizzazione *hash* statica, ma non gli indici *hash*
  - *SQLServer* supporta solo *B+-trees*

9 INDICI BITMAP

Una ***bitmap*** è semplicemente una serie di bit. Gli **indici *bitmap*** sono un tipo speciale di indice progettato per eseguire in modo più efficiente query su più chiavi (*querying on multiple keys*).

Si presume che i *record* in una relazione siano numerati sequenzialmente, diciamo, dal numero 0. Dato un numero n, deve essere facile recuperare il record n, il che è particolarmente facile se i *record* sono di dimensioni fisse.

---

[3] Potremmo anche organizzarli come **file *heap***, in cui i *record* non sono ordinati in un modo particolare.

Gli indici *bitmap* sono applicabili su attributi che assumono un numero relativamente ridotto di valori distinti (ad esempio "sesso", "paese", "stato", ...oppure "reddito" ma suddiviso in un ridotto numero di livelli intermedi, ad esempio 0-9999, 10000-19999, 20000-50000, 50000-∞)

Nella sua forma più semplice, un indice *bitmap* su di un attributo (ad es: *gender*) ha una *bitmap* per ogni valore dell'attributo (ad es: "m"). Una *bitmap* ha tanti bit quanti record (nell'esempio "m" ne ha 5). In una *bitmap* per il valore v, il bit per un *record* è 1 se il record ha il valore v per l'attributo ed è 0 altrimenti. Vediamo esaurientemente l'esempio:

record number	*name*	*gender*	*address*	*income _level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

9.1 BITMAP OPERATIONS

Gli indici bitmap sono utili per le query su più attributi, ma non particolarmente utile per query su un attributo singolo.

Le query vengono portate a termine utilizzando ***bitmap operations***:

- Intersezione (∧, AND)
- Unione (∨, OR)
- Complemento (¬, NOT)

Ogni operazione richiede due *bitmap* della stessa dimensione e applica l'operazione sui bit corrispondenti per ottenere la *bitmap* del risultato. Ad esempio:

- 100110 AND 110011 = 100010;
- 100110 OR 110011 = 110111
- NOT 100110 = 011001
- Maschi con reddito L1: 10010 AND 10100 = 10000 (in riferimento all'esempio precedente)
  - Abbiamo visto quindi che si possono recuperare le tuple richieste.
  - Il conteggio del numero di tuple corrispondenti è ancora più veloce.

9.2 ACCORGIMENTI NELL'USO DI *BITMAP*

Nell'utilizzare gli indici *bitmap* bisogna avere a mente alcune loro caratteristiche:

- Gli indici *bitmap* sono generalmente molto piccoli rispetto alla dimensione della relazione.
  - Ad esempio, se il record è 100 byte, lo spazio per una *bitmap* singolo è 1/800 di spazio usato dalla relazione. Se il numero di valori di attributo distinti è 8, bitmap è solo l'1% di dimensione della relazione.
- La **cancellazione** deve essere gestita correttamente, ovvero va tenuta traccia di essa:
  - a tal proposito ci sono le ***Existence bitmap*** per capire se esiste un *record* valido in una *locazione*
- Le Existence bitmap sono inoltre necessarie per l'operazione di complementazione:
  - Per eseguire *NOT(Attributo=valore)* si fa *(NOT bitmap-Attributo-valore) AND (ExistenceBitmap)*
- Bisognerebbe utilizzare *bitmap* per tutti i valori, anche per il valore *null*:
  - Per gestire correttamente la semantica SQL del *null* SQL per l'espressione *NOT(Attributo=valore)* bisogna fare:
    (NOT bitmap-Attributo-valore) AND (ExistenceBitmap) AND (NOT bitmap-Attributo-Null)

9.3 Implementazione efficiente delle operazioni bitmap

Le *bitmap* sono raggruppate in *words*; una **AND** a singola *word* (è una istruzione di base della CPU) calcola la AND di 32 o 64 bit contemporaneamente. Ad esempio, una AND su *bitmap* da 1 milione di bit può essere eseguita con solo 31.250 istruzioni.

Il **conteggio del numero di "1"** in una *bitmap* può essere eseguito rapidamente con un trucco:

- Possiamo utilizzare un *array* con 256 *entries* (voci), in cui la voce *i*-esima memorizza il numero di bit a "1" nella rappresentazione binaria di *i*. Impostiamo il conto inizialmente su 0.
- Prendiamo ogni byte della *bitmap* ed usiamolo per indicizzare all'interno di questo *array*.
- Aggiungiamo poi il conteggio memorizzato al conteggio totale.
- Il numero di operazioni di aggiunta è 1/8 del numero di tuple e quindi il processo di conteggio è molto efficiente. Un array di grandi dimensioni (ad es: 2^{16} = 65.536 voci), indicizzato da coppie di byte, darebbe ancora maggiore velocità, ma ad un costo di archiviazione più elevato.

È possibile utilizzare *bitmap* anziché liste di Tuple-ID a livello-foglia nei *B+-trees*, per valori con un numero elevato di *record* corrispondenti:

- Vale la pena se più di 1/64 dei *record* ha quel valore, assumendo che una Tuple-ID sia da 64 bit
- La tecnica sopra unisce i vantaggi di indici *bitmap* e *B+-tree*

10 Definizione degli indici in SQL

Creare un indice:

```
create index <index-name> on <relation-name>
                          (<attribute-list>)

         E.g.: create index b-index on branch(branch_name)
```

Si usa per ***create unique index*** per specificare indirettamente e applicare la condizione che la *search-key* sia una *candidate key* (non è davvero richiesto se è invece supportato il vincolo di integrità ***unique*** di SQL).

Per eliminare un indice:

```
drop index <index-name>
```

11 Appendice

11.1 Hashing partizionato

Significa che gli *hash values* sono suddivisi (*splitted*) in segmenti, dipendenti ognuno da un singolo attributo della *search-key:*

$(A_1, A_2, \dots, A_n)$ per n attributi della *search-key*

Vediamo un esempio per $n = 2$ nella relazione *customer*, avente la *search-key* (*customer-street, customer-city*):

search-key value	*hash value*
(Main, Harrison)	101 111
(Main, Brooklyn)	101 001
(Park, Palo Alto)	010 010
(Spring, Brooklyn)	001 001
(Alma, Palo Alto)	110 010

Per rispondere ad una *equality query* ("interrogazione sull'uguaglianza") su un singolo attributo, è necessario cercare in più *bucket* (è simile in effetti ai *grid files* → vedi dopo).

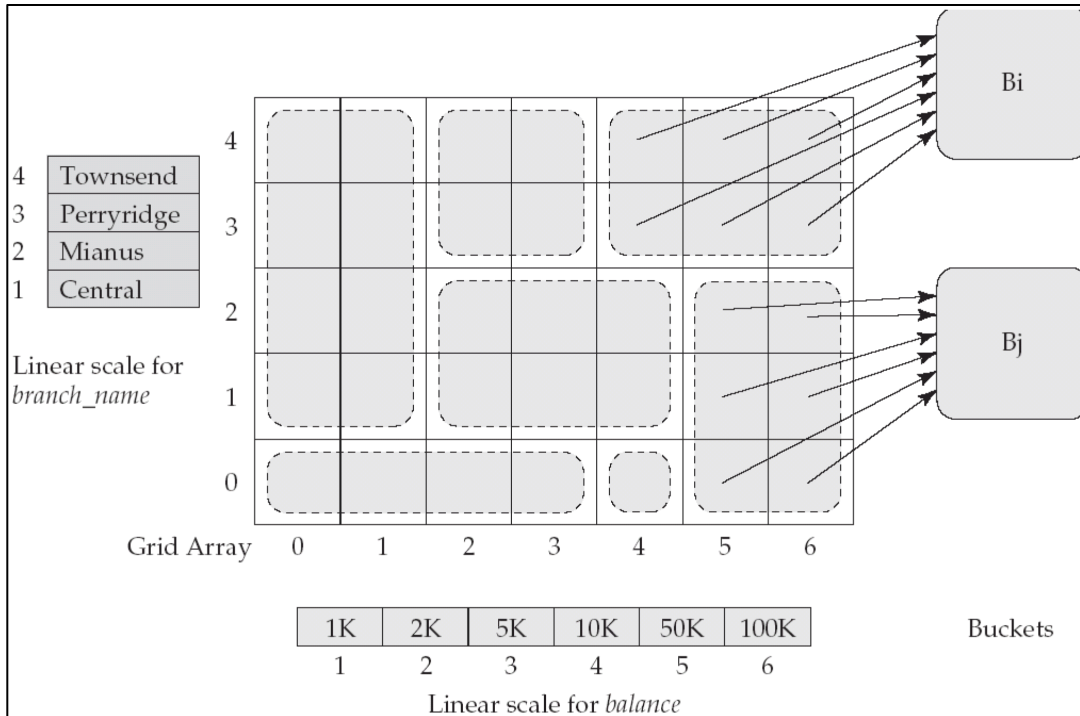
11.2 Grid Files

Sono una struttura utilizzata per velocizzare l'elaborazione di *general multiple search-key queries* relative a uno o più operatori di confronto (comparison operators).

Il *grid file* ha un *single grid array* e una *linear scale* per ciascun attributo della *search-key*. Il *grid array* ha un numero di dimensioni pari al numero di attributi della *search-key*.

Più celle del *grid array* possono puntare allo stesso *bucket*.

Per trovare il *bucket* per un *search-key value* bisogna individuare la riga e la colonna della relativa cella utilizzando le *linear scales* (scale lineari) e seguire il puntatore



Un *grid file* su due attributi A e B può gestire **query** di tutti i seguenti tipi con efficienza ragionevole:

- $(a_1 \leq A \leq a_2)$
- $(b_1 \leq B \leq b_2)$
- $(a_1 \leq A \leq a_2) \wedge (b_1 \leq B \leq b_2)$

Ad esempio, per rispondere a $(a_1 \leq A \leq a_2) \wedge (b_1 \leq B \leq b_2)$ utilizza le scale lineari per trovare le corrispondenti celle candidate del *grid array* e cerca tutti i *bucket* indicati da quelle celle.

Durante l'**inserimento**, se un *bucket* diventa pieno, è possibile creare un nuovo *bucket* se più di una cella punta ad esso:

- Idea simile all'*extendible hash*, ma su più dimensioni
- Se solo una cella punta ad essa, deve essere creato un *overflow bucket* o deve essere aumentata la dimensione della griglia.

Le *linear scales* devono essere scelte per distribuire uniformemente i *record* sulle celle, altrimenti ci saranno troppi *overflow bucket*. Una riorganizzazione periodica per aumentare la dimensione della griglia aiuterà. Attenzione però, la riorganizzazione può essere molto costosa.

Lo *space overhead* della griglia può essere elevato.

Gli alberi R (capitolo 23) sono un'alternativa