



DISCLAIMER

Il materiale contenuto nel drive è stato raccolto e richiesto tramite autorizzazione ai ragazzi frequentanti il corso di studi di Informatica dell'Università degli Studi di Salerno. Gli appunti e gli esercizi nascono da un uso e consumo degli autori che li hanno creati e risistemati per tanto non ci assumiamo la responsabilità di eventuali mancanze o difetti all'interno del materiale pubblicato.

Il materiale sarà modificato aggiungendo il logo dell'associazione, in tal caso questo possa recare problemi ad alcuni autori di materiale pubblicato, tale persona può contattarci in privato ed eliminaremo o modificheremo il materiale in base alle sue preferenze.

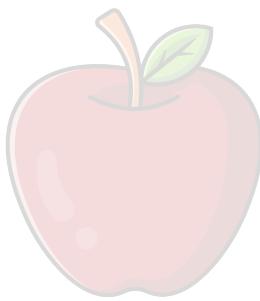
Ringraziamo eventuali segnalazioni di errori così da poter modificare e fornire il miglior materiale possibile a supporto degli studenti.



CoScienze
Associazione



2019/2020



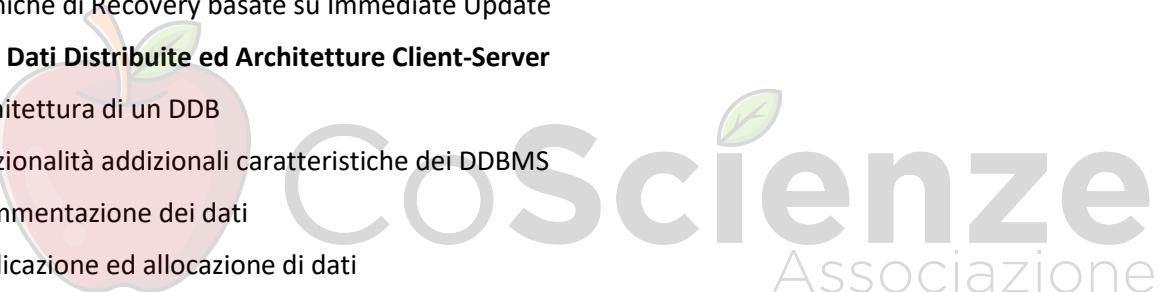
BASI DI DATI 2 - APPUNTI

CORSO DI LAUREA MAGISTRALE – SICUREZZA INFORMATICA
Coscienze
Associazione

Sommario

Dipendenze funzionali e Normalizzazione di DB Relazionali	1
Dipendenze funzionali	7
Regole di inferenza per dipendenze funzionali	9
Forme Normali	12
Algoritmi per la progettazione di DB Relazionali e ulteriori dipendenze	22
Decomposizione delle relazioni e insufficienza delle forme normali	22
Decomposizione e conservazione delle dipendenze	23
Decomposizione e Join senza perdita (non-additivi)	25
Problemi con valori nulli e tuple dangling	30
Dipendenze multivalore e quarta forma normale	33
Regole di inferenza per FD e MVD	35
Applicazioni di design e Tuning di database	37
Il ciclo di vita di un sistema informativo	38
Le fasi del micro-ciclo di vita	39
Il processo di progettazione di un database	40
La progettazione fisica nei database relazionali	47
Memorizzazione di record ed Organizzazione dei file	49
Gerarchie di memoria e dispositivi di memorizzazione	49
Dispositivi di memoria secondaria	51
Collocazione su disco dei record di un file	54
Ripartizione dei record in blocchi e confronto tra record con spanning e record senza spanning	56
Header dei file	57
File di record non ordinati (file heap)	59
File di record ordinati (file sorted)	60
Tecniche hash	63
Tecnologia RAID	68
Strutture di indici per i file	71
Indici primari	72
Indici di cluster	73
Indici secondari	74
Indice multilivello	77
Indici dinamici multilivello implementati da alberi B e alberi B ⁺	79
Alberi di ricerca e alberi B	79
Alberi B ⁺	83

Gestione delle Transazioni	88
Controllo della concorrenza	89
Tecniche di recovery	91
Il System Log	93
Proprietà delle transazioni	93
Gli Schedule	94
Schedule serializzabili	97
Supporto alle transazioni in SQL	99
Tecniche per il Controllo della Concorrenza	101
Lock binari	101
Lock shared/esclusivi	102
Protocollo Two-Phase Locking	105
Tecniche di Recovery	110
Protocollo WAL (Write-Ahead Logging)	112
Tecniche di recovery basate su Deferred Update	113
Tecniche di Recovery basate su Immediate Update	114
Basi di Dati Distribuite ed Architetture Client-Server	117
Architettura di un DDB	118
Funzionalità addizionali caratteristiche dei DDBMS	120
Frammentazione dei dati	121
Replicazione ed allocazione di dati	122
Tipi di DDB	132
Architettura Client-Server	135
XML e basi di dati in Internet	137
XML Schema	139
Attributi	140
Restrizioni	141
Enumerazione	141
Elementi complessi (complex elements)	141
Costrutto sequence	141
Costrutto all	142
Costrutto choice	142
Elementi EMPTY	142
Interrogazioni in XML	143
Database NoSQL	148
Transazioni BASE	149



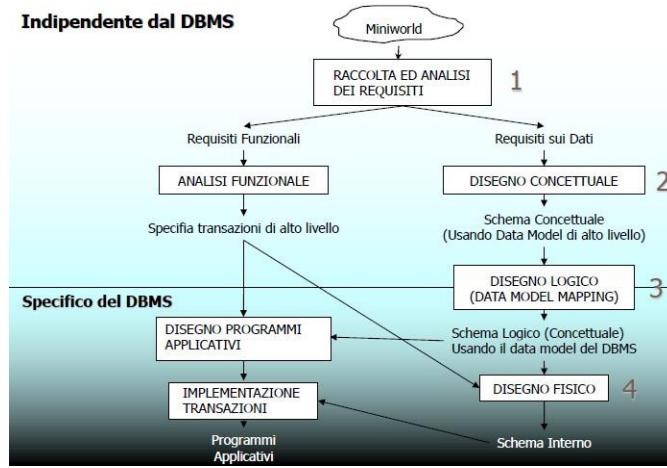
Vantaggi e svantaggi del NoSQL	149
Classificare i DBMS NoSQL	150
Basi di dati semantiche, SPARQL e Linked Open Data	153
Modello RDF (Resource Description Framework)	153
SPARQL (Simple Protocol and RDF Query Language)	157
SPARQL 1.1	163
Linked e Open Data	165
Data Warehouse	166



CoScienze
Associazione

Dipendenze funzionali e Normalizzazione di DB Relazionali

Le fasi di progettazione di un database



Ogni schema di relazione è composto da un certo numero di attributi, e lo schema di base di dati relazionale è composto da un certo numero di schemi di relazione. Non sempre dalla fase di progettazione si ottiene uno schema privo di difetti, infatti possono sorgere dei problemi nella fase di mapping dal modello **ER** allo **schema relazionale**. Si può ottenere uno schema non ottimale progettando direttamente uno schema logico saltando la fase di analisi concettuale. Tali difetti possono portare ad anomalie nella base di dati.

Ci sono due livelli ai quali è possibile esaminare la “bontà” di uno schema di relazione; il primo è il **livello logico**, come gli utenti interpretano gli schemi di relazione e il significato dei loro attributi. L'avere buoni schemi di relazione a questo livello consente agli utenti di capire chiaramente il significato dei dati nelle relazioni, e perciò di formulare correttamente le loro interrogazioni. Il secondo è il **livello d'implementazione**, come le tuple in una relazione di base sono memorizzate ed aggiornate. Questo livello si applica solo a schemi di relazione di base, che saranno memorizzati fisicamente come file, mentre a livello logico siamo interessati a schemi sia di relazioni di base sia di viste (relazioni virtuali).

Inoltre, come qualsiasi tipo di progettazione, la progettazione di una base di dati può essere eseguita usando due approcci: **bottom-up** (dal basso verso l'alto, ossia dai concetti generali a quelli specifici). Tale metodologia considera come punto di partenza le associazioni di base tra singoli attributi, e le usa per costruire relazioni. Questo approccio è poco popolare nella pratica e soffre del problema di collezionare un gran numero di associazioni tra coppie di attributi come punto di partenza. Al contrario, una metodologia di progettazione **top-down** comincia con un certo numero di raggruppamenti di attributi in relazioni; raggruppamenti che sono già stati ottenuti dalle attività di progettazione concettuale e di traduzione. Tale progettazione viene quindi applicata alle relazioni individualmente e collettivamente, portando a decomposizioni successive finché non si ottengono tutte le proprietà desiderate.

Esistono delle **misure informali di qualità** per la progettazione di uno schema di relazione:

1. SEMANTICA DEGLI ATTRIBUTI
2. RIDUZIONE DEI VALORI RIDONDANTI NELLE TUPLE
3. RIDUZIONE DEI VALORI **NULL** NELLE TUPLE
4. NON CONSENTIRE TUPLE SPURIE

Come si vedrà, queste misure non sono sempre indipendenti tra loro.

Semantica degli attributi di una relazione

Ogni volta che si raggruppano degli attributi per formare uno schema di relazione, si suppone che ad essi sia associato un certo significato. Questo significato, o semantica, specifica come interpretare i valori degli attributi memorizzati in una tupla della relazione; in altre parole, come i valori degli attributi in una tupla sono in relazione tra loro. Se è stata eseguita correttamente la progettazione concettuale, seguita da una traduzione nelle relazioni, la maggior parte della semantica dovrebbe essere spiegata e il progetto risultante dovrebbe avere un significato chiaro. In generale il progetto di uno schema di relazione sarà tanto migliore quanto più facile è spiegare la semantica della relazione. Per illustrare ciò si consideri la Figura 1, una versione semplificata dello schema del database Company e la Figura 2, che presenta un esempio di relazioni popolate di questo schema. Il significato dello schema di relazione IMPIEGATO è piuttosto semplice: ogni tupla rappresenta un impiegato, con i valori per il nome dell'impiegato (NOME_I), il numero di previdenza sociale (SSN), la data di nascita (DATA_N) e l'indirizzo (INDIRIZZO), nonché il numero del dipartimento per cui l'impiegato lavora (NUMERO_D). L'attributo NUMERO_D è una chiave esterna che rappresenta un'*associazione implicita* tra IMPIEGATO e DIPARTIMENTO.

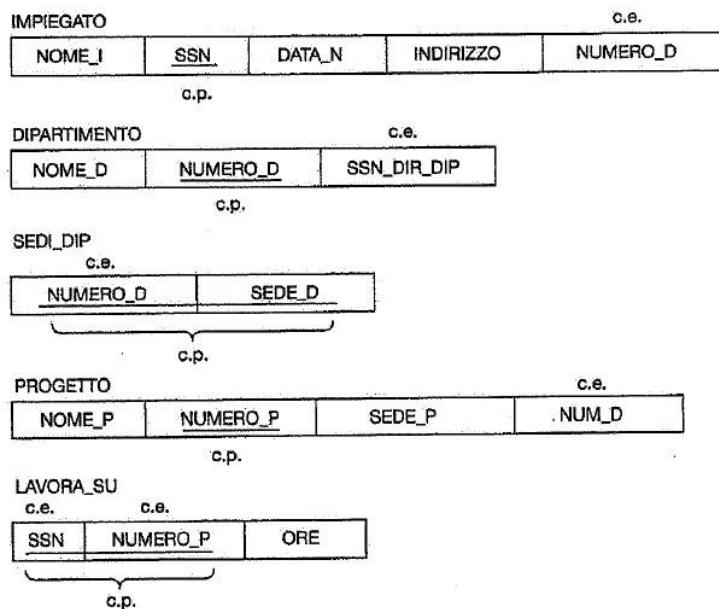


Figura 1 - Versione semplificata dello schema del database Company

IMPIEGATO				
NOME_I	SSN	DATA_N	INDIRIZZO	NUMERO_D
Smith,John B.	123456789	1965-01-09	731 Fondren,Houston,TX	5
Wong,Franklin T.	333455555	1955-12-08	638 Voss,Houston,TX	5
Zelaya,Alicia J.	999887777	1968-07-19	3321 Castle,Spring,TX	4
Wallace,Jennifer S.	987654321	1941-06-20	291 Berry,Bellaire,TX	4
Narayan,Ramesh K.	666884444	1982-09-15	975 Fire Oak,Humble,TX	5
English,Joyce A.	453453453	1972-07-31	5631 Rice,Houston,TX	5
Jabbar,Ahmed V.	987987987	1969-03-29	980 Dallas,Houston,TX	4
Borg,James E.	888865555	1937-11-10	450 Stone,Houston,TX	1

DIPARTIMENTO		
NOME_D	NUMERO_D	SSN_DIR_DIP
Ricerca	5	333445555
Amministrazione	4	987654321
Sede centrale	1	888865555

SEDI_DIP	
NUMERO_D	SEDE_D
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

LAVORA_SU		
SSN	NUMERO_P	ORE
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888865555	20	null

PROGETTO			
NOME_P	NUMERO_P	SEDE_P	NUM_D
ProdottoX	1	Bellaire	5
ProdottoY	2	Sugarland	5
ProdottoZ	3	Houston	5
Computerizzazione	10	Stafford	4
Riorganizzazione	20	Houston	1
Nuove opportunità	30	Stafford	4

Figura 2 - Relazioni d'esempio per lo schema di Figura 1

più complessa. Ogni tupla in SEDI_DIP fornisce un numero di dipartimento (NUMERO_D) e una delle sedi del dipartimento (SEDE_D). Ogni tupla in LAVORA_SU dà il numero di previdenza sociale di un impiegato (SSN),

il numero di progetto di *uno dei progetti* su cui lavora l'impiegato (NUMERO_P) e il numero di ore settimanali lavorate dell'impiegato su quel progetto (ORE). Entrambi gli schemi hanno comunque un'interpretazione ben definita e non ambigua. Lo schema SEDI_DIP rappresenta un attributo multivalore di DIPARTIMENTO, mentre LAVORA_SU rappresenta un'associazione M:N tra IMPIEGATO e PROGETTO. Tutti gli schemi di relazione in Figura 1 possono perciò essere considerati "buoni" dal punto di vista di una **semantica chiara**. La seguente linea guida informale sviluppa ulteriormente la progettazione di uno schema di relazione.

LINEA GUIDA 1. Si progetti ogni schema di relazione in modo tale che sia semplice spiegarne il significato. Non si uniscano attributi provenienti da più tipi di entità e tipi di associazione in un'unica relazione. Intuitivamente, se uno schema di relazione corrisponde a un solo tipo di entità o a un tipo di associazione, il suo significato tende a essere chiaro. Altrimenti la relazione corrisponde a un miscuglio di più entità e associazioni e perciò diviene semanticamente oscura.

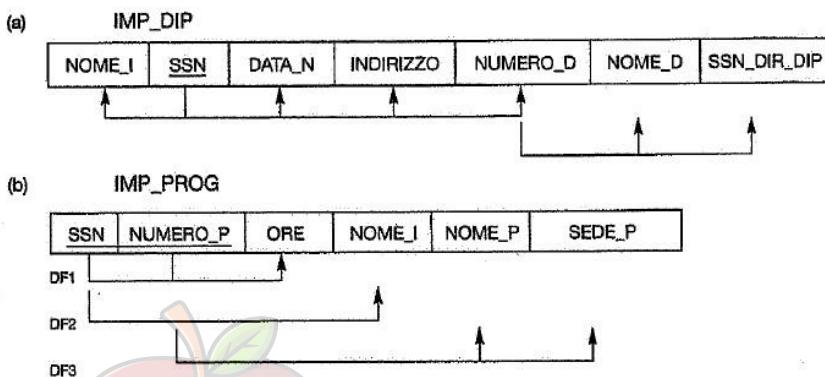


Figura 3 - Due schemi di relazione e le rispettive dipendenze funzionali

Anche gli schemi di relazione nella Figura 3 (a) e (b) presentano una semantica chiara. Una tupla nello schema di relazione IMP_DIP rappresenta un singolo impiegato ma comprende informazioni aggiuntive, vale a dire il nome (NOME_D) del dipartimento per cui l'impiegato lavora e il numero di previdenza sociale (SSN_DIR_DIP) del direttore del dipartimento.

Per la relazione IMP_PROG, ogni tupla collega un impiegato a un progetto, ma comprende anche il nome dell'impiegato (NOME_I), il nome del progetto (NOME_P) e la sede del progetto (SEDE_P). Anche se non c'è alcunché di sbagliato dal punto di vista logico in queste due relazioni, esse violano la linea guida 1, mischiando attributi provenienti da distinte entità del mondo reale; IMP_DIP mescola attributi impiegati e dipartimenti, e IMP_PROG mescola attributi di impiegati e progetti. Queste relazioni possono essere usate come viste, ma causano problemi quando vengono usate come relazioni di base.

Riduzione dei valori ridondanti nelle tuple

Uno scopo della progettazione di schemi è quello di ridurre al minimo lo spazio di memoria occupato dalle relazioni di base (file). Il raggruppamento di attributi in schemi di relazione ha un effetto significativo sullo spazio di memoria. Ad esempio, si confronti lo spazio occupato dalle due relazioni di base IMPIEGATO e DIPARTIMENTO in Figura 2 con lo spazio della relazione di base IMP_DIP in Figura 4, che è il risultato dell'applicazione dell'operazione di JOIN NATURALE a IMPIEGATO e DIPARTIMENTO. In IMP_DIP, i valori degli attributi che riguardano uno specifico dipartimento (NUMERO_D, NOME_D, SSN_DIR_DIP) sono ripetuti per *ogni impiegato che lavora per quel dipartimento*. Al contrario, l'informazione su ogni dipartimento appare solo una volta nella relazione DIPARTIMENTO di Figura 2. Solo il numero di dipartimento (NUMERO_D) è ripetuto nella relazione IMPIEGATO per ogni impiegato che lavora in quel dipartimento. Considerazioni simili si applicano alla relazione IMP_PROG (Figura 4), che aumenta la relazione LAVORA_SU con ulteriori attributi di IMPIEGATO e PROGETTO.

Un altro grave problema che si presenta usando le relazioni in Figura 4 come relazioni di base è il problema delle **anomalie di aggiornamento**. Esse possono essere classificate in *anomalie di inserimento*, *anomalie di cancellazione* e *anomalie di modifica*.

IMP_DIP						
NOME_I	SSN	DATA_N	INDIRIZZO	NUMERO_D	NOME_D	SSN_DIR_DIP
Smith,John B.	123456789	1965-01-09	731 Foreign Houston,TX	5	Ricerca	333445555
Wong,Franklin T.	333445555	1955-12-08	638 West Houston,TX	5	Ricerca	333445555
Zelaya,Alicia J.	999887777	1988-07-19	3321 Castle Spring,TX	4	Amministrazione	987654321
Wallace,Jennifer S.	987654321	1941-06-20	291 Berry Bellair,TX	4	Amministrazione	987654321
Narayan,Ramesh K.	666884444	1962-09-15	975 FireOak Humble,TX	5	Ricerca	333445555
English,Joyce A.	453453453	1972-07-31	5631 Rice, Houston,TX	5	Ricerca	333445555
Jabber,Ahmad V.	987987987	1969-03-29	980 Dallas,Houston,TX	4	Amministrazione	987654321
Borg,James E.	888665555	1937-11-10	450 Stone Houston,TX	1	Sede centrale	888665555

IMP_PROG					
SSN	NUMERO_P	ORE	NOME_I	NOME_P	SIDE_P
123456789	1	32.5	Smith,John B.	ProdottoX	Bellair
123456789	2	7.5	Smith,John B.	ProdottoY	Sugarland
666884444	3	40.0	Narayan,Ramesh K.	ProdottoZ	Houston
453453453	1	20.0	English,Joyce A.	ProdottoX	Bellair
453453453	2	20.0	English,Joyce A.	ProdottoY	Sugarland
333445555	2	10.0	Wong,Franklin T.	ProdottoY	Sugarland
333445555	3	10.0	Wong,Franklin T.	ProdottoZ	Houston
333445555	10	10.0	Wong,Franklin T.	Computerizzazione	Stafford
333445555	20	10.0	Wong,Franklin T.	Riorganizzazione	Houston
999887777	30	30.0	Zelaya,Alicia J.	Nuova opportunità	Stafford
999887777	10	10.0	Zelaya,Alicia J.	Computerizzazione	Stafford
987987987	10	35.0	Jabber,Ahmad V.	Computerizzazione	Stafford
987987987	30	5.0	Jabber,Ahmad V.	Nuove opportunità	Stafford
987654321	30	20.0	Wallace,Jennifer S.	Riorganizzazione	Houston
987654321	20	15.0	Wallace,Jennifer S.	Riorganizzazione	Houston
888665555	20	null	Borg,James E.	Riorganizzazione	Houston

Figura 4 - Relazioni esemplificative per gli schemi in Figura 3, ottenute applicando il JOIN NATURALE alle relazioni in Figura 2.

Anomalie di inserimento. Possono essere distinte in due tipi, illustrati dai seguenti esempi basati sulla relazione IMP_DIP.

- Per inserire una nuova tupla impiegato in IMP_DIP occorre inserire i valori degli attributi del dipartimento per cui l'impiegato lavora, o valori nulli (se l'impiegato ancora non lavora per un dipartimento). Ad esempio, per inserire una nuova tupla per un impiegato che lavora nel dipartimento numero 5, bisogna inserire correttamente i valori degli attributi del dipartimento 5 in modo tale che siano *consistenti* con i valori del dipartimento 5 in altre tuple di IMP_DIP. Nel progetto di Figura 2 non ci si deve preoccupare di questo problema di consistenza, dato che nella tupla impiegata viene inserito solo il numero del dipartimento; tutti gli altri valori di attributi di dipartimento 5 sono registrati solo una volta nella base di dati, come tupla singola nella relazione DIPARTIMENTO.
- È difficile inserire nella relazione IMP_DIP un nuovo dipartimento che non ha ancora impiegati. Il solo modo per farlo è quello di porre valori nulli negli attributi relativi a impiegato. Ciò causa un problema perché SSN è la chiave primaria di IMP_DIP, e si suppone che ogni tupla rappresenti un'entità impiegato, non un'entità dipartimento. Inoltre, quando viene assegnato il primo impiegato a quel dipartimento, non si ha più bisogno della tupla con valori nulli. Questo problema non si presenta nel progetto di Figura 2, perché un dipartimento viene inserito nella relazione DIPARTIMENTO che in esso lavorino impiegato o non lavorino, e ogni volta che un impiegato è assegnato a quel dipartimento una tupla corrispondente è inserita in IMPIEGATO.

Anomalie di cancellazione. Questo problema è legato alla seconda situazione di anomalia di inserimento discussa sopra. Se si cancella da IMP_DIP una tupla impiegato che è quella che rappresenta l'ultimo impiegato che lavora per un particolare dipartimento, l'informazione riguardante quel dipartimento non è più presente nella base di dati. Il problema non si presenta nella base di dati di Figura 2 perché le tuple di DIPARTIMENTO sono memorizzate separatamente.

Anomalie di modifica. In IMP_DIP, se si cambia il valore di uno degli attributi di un particolare dipartimento, ad esempio il direttore del dipartimento 5, occorre aggiornare le tuple di tutti gli impiegati che lavorano in quel dipartimento, altrimenti la base di dati diverrà inconsistente. Se ci si dimentica di aggiornare alcune tuple, per lo stesso dipartimento verranno mostrati due diversi valori di direttore in diverse tuple impiegato, il che non dovrebbe verificarsi.

Basandosi sulle tre anomalie precedenti, è possibile fornire la seguente linea guida.

LINEA GUIDA 2. Si progettino gli schemi di relazione di base in modo che nelle relazioni non siano presenti anomalie di inserimento, cancellazione o modifica. Se sono presenti delle anomalie, le si rilevi chiaramente e ci si assicuri che i programmi che aggiornano la base di dati operino correttamente.

La seconda linea guida è consistente con la prima e, in un certo senso, ne è una riaffermazione. È importante notare che queste linee guida possono talora *dover esser violate* al fine di *incrementare le prestazioni* di certe interrogazioni. Ad esempio, se un'interrogazione importante recupera informazioni riguardanti il dipartimento di un impiegato, insieme con gli attributi di impiegato, lo schema IMP_DIP può essere usato come relazione di base. Le anomalie in IMP_DIP devono però essere notate e ben comprese, in modo tale che non si finisca col produrre inconsistenze ogni volta che viene aggiornata la relazione di base. In generale è consigliabile usare relazioni di base prive di anomalie, e specificare viste che utilizzino i JOIN per raggruppare gli attributi frequentemente richiamati in interrogazioni importanti. Ciò riduce il numero di termini JOIN specificati nell'interrogazione, rendendo più semplice scrivere l'interrogazione correttamente, e in molti casi migliora le prestazioni.

Riduzione dei valori null nelle tuple

È possibile che nei progetti di alcuni schemi vengano raggruppati numerosi attributi a formare una “grossa” relazione. Se molti di questi attributi non riguardano tutte le tuple della relazione, quelle tuple finiscono con l'avere molti valori nulli. Ciò può dar luogo a uno spreco di spazio di memoria e può anche portare a problemi di comprensione del significato degli attributi e di specificazione delle operazioni di JOIN a livello logico. Un altro problema con i valori null è come rispondere della loro presenza quando vengono applicate operazioni aggregate come COUNT o SUM. Inoltre, i valori null possono avere più interpretazioni, tra cui le seguenti:

- L'attributo *non è pertinente* per questa tupla;
- Il valore dell'attributo per questa tupla è *sconosciuto*;
- Il valore è *noto ma assente*, cioè non è ancora stato memorizzato;

L'uso della stessa rappresentazione per tutti i valori null non consente di cogliere i diversi significati che essi possono assumere. Si può pertanto enunciare un'altra linea guida.

LINEA GUIDA 3. Per quanto possibile, si eviti di porre in una relazione di base attributi i cui valori possono essere frequentemente null. Se i valori null sono inevitabili, ci si assicuri che essi si presentino solo in casi eccezionali e che non riguardino una maggioranza di tuple nella relazione.

Ad esempio, se solo il 10% degli impiegati ha un ufficio personale, non c'è ragione di includere un attributo NUMERO_UFFICIO nella relazione IMPIEGATO; piuttosto può essere progettata una relazione IMP_UFFICI (SSN_I, NUMERO_UFFICIO) che comprenda tuple solo per impiegati che hanno uffici privati.

Non consentire tuple spurie

Si considerino i due schemi di relazione IMP_SEDI e IMP_PROG1 di Figura 5, che possono essere usati al posto della relazione IMP_PROG di Figura 3(b). Una tupla in IMP_SEDI indica che l'impiegato il cui nome è NOME_I lavora su *qualche progetto* la cui sede è SEDE_P. Una tupla in IMP_PROG1 indica che l'impiegato il cui numero di previdenza sociale è SSN lavora un certo numero di ORE a settimana sul progetto il cui nome, numero e

(a)

IMP_SEDI	
NOME_I	SEDE_P
o.p.	

IMP_PROG1

SSN	NUMERO_P	ORE	NOME_P	SEDE_P
c.p.				

(b)

IMP_SEDI	
NOME_I	SEDE_P
Smith, John B.	Bellaire
Smith, John B.	Sugarland
Narayan, Ramesh K.	Houston
English, Joyce A.	Bellaire
English, Joyce A.	Sugarland
Wong, Franklin T.	Sugarland
Wong, Franklin T.	Houston
Wong, Franklin T.	Stafford
Zelina, Alicia J.	Stafford
Jabbar, Ahmad V.	Stafford
Wallace, Jennifer S.	Stafford
Wallace, Jennifer S.	Houston
Borg, James E.	Houston

IMP_PROG1

SSN	NUMERO_P	ORE	NOME_P	SEDE_P
123456789	1	32.5	Prodotto X	Bellaire
123456789	2	7.5	Prodotto Y	Sugarland
666884444	3	40.0	Prodotto Z	Houston
453453453	1	20.0	Prodotto X	Bellaire
453453453	2	20.0	Prodotto Y	Sugarland
333445555	2	10.0	Prodotto Y	Sugarland
333445555	3	10.0	Prodotto Z	Houston
333445555	10	10.0	Computerizzazione	Stafford
333445555	20	10.0	Computerizzazione	Houston
999887777	30	30.0	Riorganizzazione	Stafford
999887777	30	30.0	Nuove opportunità	Stafford
987987987	10	10.0	Computerizzazione	Stafford
987987987	30	35.0	Computerizzazione	Stafford
987987987	30	5.0	Nuove opportunità	Stafford
987654321	30	20.0	Nuove opportunità	Stafford
987654321	20	15.0	Riorganizzazione	Houston
888665555	20	null	Riorganizzazione	Houston

Figura 5 - Rappresentazione alternativa (di cattiva qualità) della relazione IMP_PROG. (a) Rappresentazione di IMP_PROG di Figura 3(b) tramite due schemi di relazione: IMP_SEDI e IMP_PROG1. (b) Risultato della proiezione della relazione popolata IMP_PROG di Figura 4 sugli attributi di IMP_SEDI e IMP_PROG1.

e IMP_PROG1 produce effetti indesiderati perché, quando si applica NATURALE, non si ottiene l'informazione originale corretta. Ciò si verifica perché in questo caso SEDE_P è l'attributo che collega IMP_SEDI e IMP_PROG1, e SEDE_P non è né una chiave primaria né una chiave esterna in IMP_SEDI o in IMP_PROG1. È possibile ora enunciare informalmente un'altra linea guida di progettazione.

SSN	NUMERO_P	ORE	NOME_P	SEDE_P	NOME_I
123456789	1	32.5	ProdottoX	Bellaire	Smith,John B.
* 123456789	1	32.5	ProdottoX	Bellaire	English, Joyce A.
123456789	2	7.5	ProdottoY	Sugarland	Smith, John B.
* 123456789	2	7.5	ProdottoY	Sugarland	English, Joyce A.
* 123456789	2	7.5	ProdottoY	Sugarland	Wong, Franklin T.
666884444	3	40.0	ProdottoZ	Houston	Narayan, Ramesh K.
* 666884444	3	40.0	ProdottoZ	Houston	Wong, Franklin T.
* 453453453	1	20.0	ProdottoX	Bellaire	Smith, John B.
453453453	1	20.0	ProdottoX	Bellaire	English, Joyce A.
* 453453453	2	20.0	ProdottoY	Sugarland	Smith, John B.
453453453	2	20.0	ProdottoY	Sugarland	English, Joyce A.
* 453453453	2	20.0	ProdottoY	Sugarland	Wong, Franklin T.
* 333445555	2	10.0	ProdottoY	Sugarland	Smith, John B.
* 333445555	2	10.0	ProdottoY	Sugarland	English, Joyce A.
333445555	2	10.0	ProdottoY	Sugarland	Wong, Franklin T.
* 333445555	3	10.0	ProdottoZ	Houston	Narayan, Ramesh K.
333445555	3	10.0	ProdottoZ	Houston	Wong, Franklin T.
* 333445555	10	10.0	Computerizzazione	Stafford	Wong, Franklin T.
* 333445555	20	10.0	Riorganizzazione	Houston	Narayan, Ramesh K.
333445555	20	10.0	Riorganizzazione	Houston	Wong, Franklin T.

Figura 6 - Risultato dell'applicazione dell'operazione JOIN NATURALE alle tuple che stanno sopra le linee tratteggiate in IMP_PROG1 e IMP_SEDI; le tuple spurie che sono state generate sono indicate con un asterisco.*

sede sono NOME_P, NUMERO_P e SEDE_P. In Figura 5(b) sono mostrate estensioni di relazione di IMP_SEDI e IMP_PROG1 corrispondenti alla relazione IMP_PROG di Figura 4, ottenute applicando l'operazione di PROIEZIONE (π) appropriata a IMP_PROG. Si supponga di aver usato IMP_PROG1 e IMP_SEDI invece di IMP_PROG come relazioni di base. Ciò dà luogo a un progetto di schema particolarmente infelice, perché da IMP_PROG e IMP_SEDI non si può recuperare l'informazione originariamente presente in IMP_PROG. Se si tenta un'operazione di JOIN NATURALE su IMP_PROG1 e IMP_SEDI, il risultato produce molte più tuple rispetto all'originaria popolazione di tuple in IMP_PROG. In Figura 6 è mostrato il risultato dell'applicazione del join alle sole tuple sopra le linee tratteggiate di Figura 5(b) (per ridurre la dimensione della relazione risultato). Tuple aggiuntive, non presenti in IMP_PROG, sono dette **tuple spurie** perché rappresentano un'informazione spuria o *sbagliata* che non è valida. In Figura 6 le tuple spurie sono contrassegnate con asterisco (*).

La decomposizione di IMP_PROG e IMP_SEDI si riuniscono le due relazioni usando il JOIN si riuniscono le due relazioni usando il JOIN

LINEA GUIDA 4. Si progettino schemi di relazione in modo tale che essi possano essere riuniti, tramite JOIN, con condizioni di uguaglianza su attributi che sono o chiavi primarie o chiavi esterne in modo da garantire che non vengano generate tuple spurie. Non si abbiano relazioni che contengono attributi di accoppiamento diversi dalle combinazioni chiave esterna – chiave primaria. Se relazioni di questo tipo sono inevitabili, non si effettui su di esse un'operazione di join sulla base di questi attributi, perché il join può produrre tuple spurie.

Sommario ed esame delle linee guida di progettazione

In precedenza, sono state esaminate informalmente situazioni che portano a schemi di relazione problematici, e sono state proposte linee guida informali per un buon progetto relazionale. I problemi evidenziati, che possono essere rivelati senza strumenti aggiuntivi di analisi, sono i seguenti:

- Anomalie che richiedono che venga fatto un lavoro aggiuntivo durante l'inserimento in una relazione e la modifica di una relazione, che possono causare una perdita accidentale d'informazione durante una cancellazione da una relazione;
- Perdita di spazio di memoria a causa di valori null e difficoltà di eseguire operazioni di aggregazione e operazioni di join a causa della presenza di valori null;
- Generazione di dati non validi e spuri durante i join su relazioni di base unite in modo improprio.

Successivamente si presenteranno concetti formali e una teoria che possono essere usati per definire più precisamente i concetti di "buona qualità" e "cattiva qualità" dei *singoli* schemi di relazione. Verrà esaminata prima di tutto la dipendenza funzionale come strumento d'analisi. Quindi si specificheranno le tre forme normali e la forma normale di Boyce e Codd (BCNF: Boyce-Codd Normal Form) per schemi di relazione.

Dipendenze funzionali

Il concetto più importante nella progettazione di schemi relazionali è quello di dipendenza funzionale. Si definirà, in un primo momento, formalmente questo concetto, mentre successivamente si vedrà come esso possa essere usato per definire forme normali per schemi di relazione.

Definizione di dipendenza funzionale

Una dipendenza funzionale è un vincolo tra due insiemi di attributi della base di dati. Si supponga che il nostro schema di base di dati relazionale abbia n attributi A_1, A_2, \dots, A_n ; si pensi all'intera base di dati come se fosse descritta da un solo schema di relazione **universale** $R = \{A_1, A_2, \dots, A_n\}$. Ciò non significa che si memorizzerà effettivamente la base di dati con una sola tabella universale; questo concetto verrà utilizzato solo per sviluppare la teoria formale delle dipendenze tra dati.

Una **dipendenza funzionale**, indicata con $X \rightarrow Y$, tra due insiemi di attributi X e Y che siano sottoinsiemi di R specifica un *vincolo* sulle tuple che possono formare uno stato di relazione r di R . Il vincolo è che, per ogni coppia di tuple t_1 e t_2 in r tali che $t_1[X] = t_2[X]$, si deve avere anche $t_1[Y] = t_2[Y]$. Ciò significa che i valori della componente Y di una tupla in r **dipendono da**, o sono **determinati da**, i valori della componente X , o in alternativa, che i valori della componente X di una tupla **determinano** univocamente (o **funzionalmente**) i valori della componente Y . Si dice che c'è una dipendenza funzionale da X e Y o che Y è **funzionalmente dipendente** da X . L'abbreviazione per dipendenza funzionale è **DF** o **d.f.** (in inglese f.d: *functional dependency*). L'insieme di attributi X è detto **parte sinistra** della DF, e Y è detto **parte destra**.

Pertanto, X determina funzionalmente Y in uno schema di relazione R se e solo se, ogni volta che due tuple di $r(R)$ concordano sul loro valore X , esse devono necessariamente concordare anche sul loro valore Y . Si noti quanto segue:

- Se un vincolo su R stabilisce che non ci possa essere più di una tupla con un dato valore per X in una generica istanza di relazione $r(R)$ – cioè X è una **chiave candidata** di R – ciò implica che $X \rightarrow Y$ per qualsiasi sottoinsieme Y di attributi di R (perché il vincolo di chiave implica che nessuna coppia di tuple in un qualsiasi stato valido $r(R)$ avrà lo stesso valore di X);
- Se $X \rightarrow Y$ in R , ciò non dice se in R , $Y \rightarrow X$ oppure no.

La dipendenza funzionale è una proprietà della **semantica** o del **significato dei attributi**. I progettisti della base di dati useranno la loro conoscenza della semantica degli attributi di R – vale a dire come sono collegati tra di loro – per specificare le dipendenze funzionali che dovrebbero sussistere su *tutti* gli stati di relazione (estensioni) r di R . Ogni volta che la semantica di due insiemi di attributi in R indica che deve sussistere una dipendenza funzionale, la dipendenza viene specificata con un vincolo. Estensioni di relazione $r(R)$ che soddisfano i vincoli di dipendenza funzionale sono dette **estensioni valide** (o **stati validi di relazione**) di R , perché soddisfano i vincoli di dipendenza funzionale. L'uso principale delle dipendenze funzionali è perciò quello di descrivere ulteriormente uno schema di relazione R , tramite una specificazione dei vincoli sui suoi attributi che devono valere *sempre*. Certe dipendenze funzionali possono essere specificate senza riferirsi a una particolare relazione, ma come una proprietà degli attributi coinvolti.

Ad esempio, $\{Stato, Numero_patente\} \rightarrow SSN$ deve valere per ogni adulto negli Stati Uniti. È anche possibile che certe dipendenze funzionali possano cessare di esistere nel mondo reale se l'associazione cambia. Ad esempio, un tempo nei Sati Uniti esisteva un'associazione tra i codici di avviamento postale e i prefissi telefonici, che si poteva esprimere tramite la dipendenza funzionale $Codice_avv_postale \rightarrow Prefisso_telefonico$, ma con la proliferazione dei prefissi telefonici ciò non è più vero.

Si consideri lo schema di relazione IMP_PROG in Figura 3(b); dalla semantica degli attributi sappiamo che devono sussistere le seguenti dipendenze funzionali:

- a) $SSN \rightarrow NOME_I$
- b) $NUMERO_P \rightarrow \{NOME_P, SEDE_P\}$
- c) $\{SSN, NUMERO_P\} \rightarrow ORE$

Queste dipendenze funzionali specificano che (a) il valore del numero di previdenza sociale di un impiegato (SSN) determina univocamente il nome dell'impiegato (NOME_I), (b) il valore del numero di un progetto (NUMERO_P) determina univocamente il nome del progetto (NOME_P) e la sua sede (SEDE_P), e (c) una combinazione dei valori di SSN e NUMERO_P determina univocamente il numero di ore lavorate a settimana sul progetto dall'impiegato (ORE). In alternativa si dice che NOME_I è determinato funzionalmente da (o è funzionalmente dipendente da) SSN, o che “dato un valore di SSN, noi sappiamo il valore di NOME_I” e così via.

Quella di dipendenza funzionale è una *proprietà dello schema di relazione* (intensione) R , non di un

particolare stato valido di relazione (estensione) r di R . Perciò una DF *non può* essere dedotta automaticamente da una data estensione di relazione r , ma deve essere definita esplicitamente da qualcuno che conosce la semantica degli attributi di R . Ad esempio, in Figura 7 viene mostrato uno stato particolare dello schema di relazione INSEGNA. Anche se a prima vista si può pensare che $TESTO \rightarrow INSEGNAMENTO$, non è possibile confermare questa ipotesi se non si sa che ciò è vero *per tutti i possibili stati validi* di INSEGNA. È però

Figura 7 - Lo stato della relazione INSEGNA con una dipendenza funzionale apparente $TESTO \rightarrow INSEGNAMENTO$. $INSEGNAMENTO \rightarrow TESTO$ è invece da escludere

INSEGNA		
DOCENTE	INSEGNAMENTO	TESTO
Smith	Strutture dati	Bartram
Smith	Gestione dati	Al-Nour
Hall	Compilatori	Hoffman
Brown	Strutture dati	Augenthaler

sufficiente descrivere un solo controesempio per escludere una dipendenza funzionale. Ad esempio, dato che ‘Smith’ insegna sia ‘Strutture dati’ che ‘Gestione dati’, è possibile concludere che DOCENTE *non* determina funzionalmente INSEGNAMENTO.

In Figura 3 è introdotta una **notazione diagrammatica** per rappresentare graficamente le dipendenze funzionali: ogni DF è rappresentata graficamente con una linea orizzontale. Gli attributi della parte sinistra della DF sono collegati tramite linee verticali alla linea che rappresenta la DF, mentre gli attributi della parte destra sono collegati tramite frecce che puntano verso gli attributi stessi.

Regole di inferenza per dipendenze funzionali

Si indichi con F l'insieme di dipendenze funzionali specificate sullo schema di relazione R . Tipicamente il progettista dello schema specifica le dipendenze funzionali *semanticamente evidenti*; di solito, però, sussistono molte altre dipendenze funzionali in *tutte* le istanze di relazione valide che soddisfano le dipendenze in F . Queste altre dipendenze possono essere *inferite* o *dedotte* dalle dipendenze funzionali presenti in F . L'insieme di tutte queste dipendenze è detto **chiusura** di F ed è indicato con F^+ . Ad esempio, si supponga di specificare il seguente insieme F di ovvie dipendenze funzionali sullo schema di relazione di Figura 3 (a):

```

$$F = \{ \text{SSN} \rightarrow \{\text{NOME\_I}, \text{DATA\_N}, \text{INDIRIZZO}, \text{NUMERO\_D}\}, \\ \text{NUMERO\_D} \rightarrow \{\text{NOME\_D}, \text{SSN\_DIR\_DIP}\} \\ \}$$

```

È possibile *inferire* da F le seguenti dipendenze funzionali aggiuntive:

```

$$\begin{aligned} \text{SSN} &\rightarrow \{\text{NOME\_D}, \text{SSN\_DIR\_DIP}\}, \\ \text{SSN} &\rightarrow \text{SSN}, \\ \text{NUMERO\_D} &\rightarrow \text{NOME\_D} \end{aligned}$$

```

Una DF $X \rightarrow Y$ è **inferita da** un insieme di dipendenze F specificate su R se $X \rightarrow Y$ sussiste in *ogni* stato di relazione r che sia un'estensione valida di R ; cioè, ogni volta che r soddisfa tutte le dipendenze in F , in r sussiste anche $X \rightarrow Y$. La chiusura F^+ di F è l'insieme di tutte le dipendenze funzionali che possono essere dedotte da F . Per determinare un modo sistematico per inferire dipendenze, occorre trovare un insieme di **regole di inferenza** che possono essere usate per inferire nuove dipendenze a partire da un insieme dato di dipendenze. Ora si considereranno alcune di queste regole di inferenza. Verrà usata la notazione $F \models X \rightarrow Y$ per indicare che la dipendenza funzionale $X \rightarrow Y$ è inferita dall'insieme di dipendenze funzionali F .

Nella discussione seguente quando vengono esaminate le dipendenze funzionali si userà la notazione abbreviata. Le variabili di attributo verranno concatenate e le virgolette omesse per comodità. Perciò la DF $\{X, Y\} \rightarrow Z$ è abbreviata in $XY \rightarrow Z$, e la DF $\{X, Y, Z\} \rightarrow \{U, V\}$ è abbreviata in $XYZ \rightarrow UV$. Le seguenti sei regole (da RI1 a RI6) sono note regole di inferenza per dipendenze funzionali:

- RI1 (regola riflessiva): se $X \supseteq Y$, allora $X \rightarrow Y$;
- RI2 (regola di arricchimento): $\{X \rightarrow Y\} \models XZ \rightarrow YZ$;
- RI3 (regola transitiva): $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$;
- RI4 (regola di decomposizione, o di proiezione): $\{X \rightarrow YZ\} \models X \rightarrow Y$;
- RI5 (regola di unione, o additiva): $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$;
- RI6 (regola pseudo-transitiva): $\{X \rightarrow Y, WY \rightarrow Z\} \models WX \rightarrow Z$;

La regola riflessiva (RI1) afferma che un insieme di attributi determina sempre sé stesso o qualsiasi dei suoi sottoinsiemi, il che è ovvio. Dato che RI1 genera dipendenze che sono sempre vere, queste dipendenze sono dette banali. Formalmente una dipendenza funzionale $X \rightarrow Y$ è **banale** se $X \supseteq Y$; altrimenti è **non-banale**.

La regola di arricchimento (RI2) sostiene che aggiungendo lo stesso insieme di attributi alla parte sinistra e alla parte destra di una dipendenza si ottiene un'altra dipendenza valida. Secondo la RI3 le dipendenze funzionali sono transitive. La regola di decomposizione (RI4) sostiene che si possono rimuovere attributi dalla parte destra di una dipendenza; l'applicazione ripetuta di questa regola può decomporre la DF $X \rightarrow \{A_1, A_2, \dots, A_n\}$ nell'insieme di dipendenze $\{X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n\}$. La regola di unione (RI5) consente di fare l'opposto: è possibile combinare un insieme di dipendenze $\{X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n\}$ nella singola DF $X \rightarrow \{A_1, A_2, \dots, A_n\}$.

Ognuna delle precedenti regole di inferenza può essere dimostrata a partire dalla definizione di dipendenza funzionale, con prova diretta o **per assurdo**. Una prova per assurdo suppone che la regola non valga e mostra che ciò porta a una contraddizione. Verrà ora dimostrato che le prime tre regole (da RI1 a RI3) sono valide.

PROVA DI RI1

Si supponga che se $X \supseteq Y$ e che esistano due tuple t_1 e t_2 in una certa istanza di relazione r di R tali che $t_1[X] = t_2[X]$. Allora $t_1[Y] = t_2[Y]$ perché $X \supseteq Y$; perciò in r deve valere $X \rightarrow Y$.

PROVA DI RI2

Si supponga che in un'istanza di relazione r di R valga la $X \rightarrow Y$, ma che non valga la $XZ \rightarrow YZ$. Devono perciò esistere due tuple t_1 e t_2 in r tali che

- (1) $t_1[X] = t_2[X]$,
- (2) $t_1[Y] = t_2[Y]$,
- (3) $t_1[XZ] = t_2[XZ]$,
- (4) $t_1[YZ] \neq t_2[YZ]$.

Ciò non è possibile, perché da (1) e (3) si deduce

$$(5) \quad t_1[Z] = t_2[Z],$$

e da (2) e (5) si deduce

$$(6) \quad t_1[YZ] = t_2[YZ],$$

contraddicendo la (4).



PROVA DI RI3

Si supponga che in una relazione r sussistano sia

- (1) $X \rightarrow Y$ che,
- (2) $Y \rightarrow Z$.

Allora, per ogni coppia di tuple t_1 e t_2 in r tali che $t_1[X] = t_2[X]$, si deve avere

$$(3) \quad t_1[Y] = t_2[Y], \text{ dall'assunzione (1);}$$

perciò occorre anche avere

$$(4) \quad t_1[Z] = t_2[Z],$$

dalla (3) e dall'assunzione (2); quindi in r deve valere $X \rightarrow Z$.

Usando argomentazioni simili è possibile provare le regole di inferenza da RI4 a RI6 e ogni altra regola di inferenza valida. Però un modo più semplice per dimostrare che una regola di inferenza per dipendenze funzionali è valida consiste nel provarla usando regole di inferenza che sono già state dimostrate valide. Ad esempio, si può provare le regole da RI4 a RI6 usando le regole da RI1 a RI3 come segue:

PROVA DI RI4 (USANDO DA RI1 a RI3)

1. $X \rightarrow YZ$ (per ipotesi).
2. $YZ \rightarrow Y$ (usando da RI1 e sapendo che $YZ \supseteq Y$).
3. $X \rightarrow Y$ (usando RI3 su 1 e 2).

PROVA DI RI5 (USANDO RA RI1 A RI3)

1. $X \rightarrow Y$ (per ipotesi).
2. $X \rightarrow Z$ (per ipotesi).
3. $X \rightarrow XY$ (usando RI2 su 1 arricchendo con X ; si noti che $XX = X$).
4. $XY \rightarrow YZ$ (usando RI2 su 2 arricchendo con Y).
5. $X \rightarrow YZ$ (usando RI3 su 3 e 4).

PROVA DI RI6 (USANDO DA RI1 A RI3)

1. $X \rightarrow Y$ (per ipotesi).
2. $WY \rightarrow Z$ (per ipotesi).
3. $WX \rightarrow WY$ (usando RI2 su 1 arricchendo con W).
4. $WX \rightarrow Z$ (usando RI3 su 3 e 2).

Regole di inferenza di Armstrong

È stato provato da Armstrong (1974) che le regole di inferenza da RI1 a RI3 sono **corrette e complete**. Per corrette si intende che, dato un insieme di dipendenze funzionali F specificate su uno schema di relazione R , tutte le dipendenze che è possibile inferire da F usando le regole da RI1 a RI3 sussistono in ogni stato di relazione r di R che *soddisfa le dipendenze* in F . Per complete si intende che, usando ripetutamente le regole da RI1 a RI3 per inferire dipendenze finché non se ne possono dedurre più, si ottiene come risultato l'insieme completo di *tutte le possibili dipendenze* che possono essere dedotte da F . In altre parole, l'insieme di dipendenze F^+ , che è stato detto chiusura di F , può essere determinato da F usando solo le regole di inferenza da RI1 a RI3. Le regole di inferenza da RI1 a RI3 sono nome come **regole di inferenza di Armstrong**.

Tipicamente i progettisti di basi di dati dapprima specificano l'insieme F delle dipendenze funzionali che possono essere facilmente determinate dalla semantica degli attributi di R ; poi usando RI1, RI2 e RI3 per inferire ulteriori dipendenze funzionali, le quali saranno pure valide su R . Un modo sistematico per determinare queste dipendenze funzionali aggiuntive è quello di determinare prima di tutto ogni insieme X di attributi che appare come parte sinistra di qualche dipendenza funzionale in F , e poi di determinare l'insieme di *tutti gli attributi* che sono dipendenti da X . Perciò per ogni insieme X di attributi di questo tipo, si calcola l'insieme X^+ di attributi che sono determinati funzionalmente da X sulla base di F ; X^+ è detto **chiusura di X rispetto a F** . Per calcolare X^+ può essere l'usato l'Algoritmo 1.

```
X+ := X;
repeat
    oldX+ := X+;
    for ogni dipendenza funzionale Y → Z in F do
        if X+ ⊇ Y then X+ := X+ ∪ Z;
    until (X+ = oldX+);
```

Algoritmo 1 - Determinazione di X^+ , chiusura di X rispetto a F .

Tale algoritmo inizia ponendo X^+ uguale a tutti gli attributi in X . Da RI1 è noto che tutti questi attributi sono funzionalmente dipendenti da X . Servendosi delle regole di inferenza RI3 e RI4 si aggiungono attributi a X^+ , usando tutte le dipendenze funzionali in F . Si continuano a considerare tutte le dipendenze in F (il ciclo repeat) finché non vengono più aggiunti attributi a X^+ durante un ciclo completo (il ciclo for) sulle dipendenze in F . Ad esempio, si consideri lo schema di relazione IMP_PROG di Figura 3(b); dalla semantica degli attributi è possibile specificare il seguente insieme F di dipendenze funzionali che devono valere su IMP_PROG:

```
F = {SSN → NOME_I,  
     NUMERO_P → {NOME_P, SEDE_P},  
     {SSN, NUMERO_P} → ORE  
}
```

Usando l'Algoritmo 1 si possono calcolare i seguenti insiemi di chiusura rispetto a F :

$$\begin{aligned}\{SSN\}^+ &= \{SSN, NOME_I\} \\ \{NUMERO_P\}^+ &= \{NUMERO_P, NOME_P, SEDE_P\} \\ \{SSN, NUMERO_P\}^+ &= \{SSN, NUMERO_P, NOME_I, NOME_P, SEDE_P, ORE\}\end{aligned}$$

Equivalenza di insiemi di dipendenze funzionali

Un insieme di dipendenze funzionali E è **coperto da** un insieme di dipendenze funzionali F – alternativamente si dice che F **copre** E – se ogni DF in E è presente anche in F^+ , cioè se ogni dipendenza in E può essere inferita da F . Due insiemi E ed F di dipendenze funzionali sono **equivalenti** se $E^+ = F^+$. Perciò l'equivalenza implica che ogni DF in E possa essere inferita da F , e ogni DF in F possa essere inferita da E , ossia E è equivalente a F se sussistono entrambe le condizioni E copre F e F copre E .

Si può determinare se F copre E calcolando X^+ rispetto a F per ogni DF $X \rightarrow Y$ in E , e quindi verificando se questo X^+ comprende gli attributi presenti in Y . Se è così per *ogni* DF in E , allora F copre E . Si determina se E ed F sono equivalenti verificando se E copre F e F copre E .

Forme Normali

Dopo aver studiato le dipendenze funzionali e alcune delle loro proprietà, si è ora pronti per usarle come fonti d'informazione sulla semantica degli schemi di relazione. Si supporrà qui che per ogni relazione venga dato un insieme di dipendenze funzionali, e che ogni relazione abbia una chiave primaria designata; queste informazioni, combinate con i test (condizioni) per forme normali, guidano il processo di normalizzazione. Si rivolgerà l'attenzione alle prime tre forme normali per schemi di relazione e all'intuizione che sta dietro di esse.

Introduzione alla normalizzazione

Il processo di normalizzazione sottopone uno schema di relazione a una serie di test per “certificare” se soddisfa a una certa **forma normale**. Il processo, che procede in modo top-down valutando ogni relazione con i criteri per le forme normali e decomponendo le relazioni quando necessario, può pertanto essere considerato come una *progettazione relazionale per analisi*. Inizialmente Codd ha proposto tre forme normali, che ha chiamato prima, seconda e terza forma normale. Una definizione più restrittiva di 3NF – detta forma normale di Boyce e Codd (BCNF) – è stata proposta in seguito da Boyce e Codd. Tutte queste forme normali sono basate sulle dipendenze funzionali tra gli attributi di una relazione. Più tardi sono state proposte una quarta forma normale (4NF) e una quinta forma normale (5NF), basate sui concetti di **dipendenza multivaleore** e **dipendenza di join**, rispettivamente.

La **normalizzazione dei dati** può perciò essere considerata come un processo di analisi degli schemi di relazione forniti, basato sulle loro dipendenze funzionali e chiavi primarie, per raggiungere le proprietà desiderate di (1) minimizzazione della ridondanza e (2) minimizzazione delle anomalie di inserimento, cancellazione e modifica discusse in precedenza. Schemi di relazione inadeguati, che non soddisfano certe condizioni – i **test di forma normale** – vengono decomposti in schemi di relazione più piccoli che superano i

test e pertanto possiedono le proprietà desiderate. Quindi la procedura di normalizzazione fornisce ai progettisti di basi di dati:

- Una struttura formale di analisi degli schemi di relazione basata sulle loro chiavi e sulle dipendenze funzionali tra i loro attributi;
- Una serie di test di forma normale che possono essere effettuati sui singoli schemi di relazione, in modo che la base di dati relazione possa essere **normalizzata** fino al livello desiderato, qualunque esso sia.

La **forma normale** di una relazione fa riferimento alla più alta condizione di forma normale soddisfatta, e perciò indica il livello al quale è stata normalizzata. Quando vengono considerate *separatamente* da altri fattori, le forme normali non garantiscono una buona progettazione di basi di dati. Di solito non è sufficiente verificare separatamente che ogni schema di relazione nella base di dati sia, ad esempio, in BCNF o in 3NF. Il processo di normalizzazione tramite decomposizione deve piuttosto confermare anche l'esistenza di proprietà aggiuntive che gli schemi di relazione, considerati nel loro insieme, devono possedere, come:

- La **proprietà di join senza perdita (lossless join)** o di **join non-additivo (nonadditive join)**, che garantisce che negli schemi di relazione creati dopo una decomposizione non si presenti il problema di generazione di tuple spurie discusso in precedenza.
- La **proprietà di conservazione delle dipendenze (dependency preservation)**, che garantisce il mantenimento dei vincoli di integrità originari, con le relazioni decomposte che hanno la stessa capacità della relazione originaria di rappresentare i vincoli di integrità e quindi di rilevare aggiornamenti illeciti.

La proprietà di join non-additivo è estremamente critica e deve essere raggiunta a ogni costo, mentre la proprietà di conservazione delle dipendenze, sebbene desiderabile, viene qualche volta sacrificata.

Per soddisfare altri criteri desiderabili possono essere definite altre forme normali, basate su altri tipi di vincoli. L'utilità pratica delle forme normali diventa però discutibile quando i vincoli su cui sono basate sono difficili da capire o da rilevare dai progettisti della base di dati e dagli utenti che devono scoprirli. Perciò oggigiorno la progettazione di basi di dati nell'industria presta particolare attenzione alla normalizzazione fino a BCNF o alla 4NF.

Un altro punto degno di nota è che i progettisti di basi di dati *non sono obbligati* a normalizzare fino alla più alta forma normale possibile. Le relazioni possono essere lasciate a un livello più basso di normalizzazione per ragioni di prestazioni. Il processo di memorizzazione del join di relazioni in forma normale più alta come una relazione di base - in una forma normale più bassa - è noto come **denormalizzazione**.

Prima di procedere oltre, occorre riprendere le definizioni di chiavi di uno schema di relazione. Una **superchiave** di uno schema di relazione $R = \{A_1, A_2, \dots, A_n\}$ è un insieme di attributi $S \subseteq R$ con la proprietà che nessuna coppia di tuple t_1 e t_2 in un generico stato valido di relazione r di R avrà $t_1[S] = t_2[S]$.

Una **chiave** K è una superchiave con la *proprietà aggiuntiva* che la rimozione di un qualsiasi attributo da K fa sì che K cessi di essere superchiave. La differenza tra una chiave e una superchiave sta nel fatto che una chiave deve essere *minimale*; cioè, se si ha una chiave $K = \{A_1, A_2, \dots, A_n\}$ di R , allora $K - \{A\}$ non è una chiave di R , qualsiasi sia i , $1 \leq i \leq k$. In Figura 1 {SSN} è una chiave per IMPIEGATO, mentre {SSN}, {SSN, NOME_I}, {SSN, NOME_I, DATA_N} ecc.... sono tutte superchiavi.

Se uno schema di relazione ha più di una chiave, ognuna di esse è detta **chiave candidata**. Una delle chiavi candidate è *arbitrariamente* nominata **chiave primaria**, le altre sono dette chiavi secondarie. Ogni schema di relazione deve avere una chiave primaria. In Figura 1 {SSN} è la sola chiave candidata per IMPIEGATO, e pertanto essa è anche la chiave primaria.

Un attributo di uno schema di relazione R è detto **attributo primo** di R se esso è membro di una *qualche chiave candidata* di R . Un attributo è detto **non-primo** se non è un attributo primo – cioè se non è membro di nessuna chiave candidata. In Figura 1 sia SSN sia NUMERO_P sono attributi primi di LAVORA_SU, mentre gli altri attributi di LAVORA_SU sono non-primi.

Nello schema di relazione

$$\text{Stradario} = \{\text{Via, Comune, CAP}\}$$

Si ha:

$\{\text{Via, CAP}\}$ è una **chiave** dato che $\text{CAP} \rightarrow \text{Comune}$

Quindi $\{\text{Via, CAP}\}$ sono attributi **primi**. Tuttavia, anche $\{\text{Via, Comune}\}$ è **chiave**, quindi Comune è **primo**.

Verranno qui ora presentate le prime tre forme normali: 1NF, 2NF e 3NF, proposte da Codd come una sequenza per raggiungere lo stato di relazioni in 3NF attraversando, se necessario, gli stadi intermedi di 1NF e 2NF.

Prima forma normale

La **prima forma normale (1NF)** è ora considerata parte integrante della definizione formale di relazione nel modello relazionale di base (detto anche, in questo contesto, *flat*: piano, piatto); storicamente è stata definita per non permettere l'uso di attributi multivalue, di attributi composti e delle loro combinazioni. Essa richiede che il dominio di un attributo comprenda solo *valori atomici* (semplici, indivisibili) e che il valore di qualsiasi attributo in una tupla sia un *valore singolo* del dominio di quell'attributo. Perciò la 1NF non consente di avere un insieme di valori, una tupla di valori, o una combinazione di entrambi come valore di un attributo per una *tupla singola*. In altre parole, la 1NF non permette “relazioni entro relazioni” o “relazioni come attributi di tuple”. I soli valori di attributi consentiti dalla 1NF sono i singoli **valori atomici** (o **indivisibili**).

The diagram illustrates the normalization process for the DIPARTIMENTO relation. Part (a) shows the original relation with four columns: NOME_D, NUMERO_D, SSN_DIR_DIP, and SEDI_D. A dashed arrow points from the primary key NUMERO_D to the SEDI_D column, indicating that SEDI_D is not functionally dependent on the primary key. Part (b) shows an example instance of the relation with five tuples. Part (c) shows the relation normalized into 1NF, where the SEDI_D column has been removed and replaced by two new columns: SEDE_D and SEDI_D, which are functionally dependent on the primary key NUMERO_D.

(a) DIPARTIMENTO			
NOME_D	NUMERO_D	SSN_DIR_DIP	SEDI_D

(b) DIPARTIMENTO			
NOME_D	NUMERO_D	SSN_DIR_DIP	SEDI_D
Ricerca	5	333445555	{Bellaire, Sugarland, Houston}
Amministrazione	4	987654321	{Stafford}
Sede centrale	1	888665555	{Houston}

(c) DIPARTIMENTO			
NOME_D	NUMERO_D	SSN_DIR_DIP	SEDE_D
Ricerca	5	333445555	Bellaire
Ricerca	5	333445555	Sugarland
Ricerca	5	333445555	Houston
Amministrazione	4	987654321	Stafford
Sede centrale	1	888665555	Houston

Figura 8 - Normalizzazione in 1NF. (a) Schema di relazione che non è in 1NF. (b) Istanza di relazione d'esempio. (c) Relazione in 1NF con ridondanza

Ci sono due modi di vedere l'attributo SEDI_D:

- Il dominio di SEDI_D contiene valori atomici, ma alcune tuple possono avere un insieme di questi valori; in questo caso SEDI_D non è funzionalmente dipendente da NUMERO_D;
- Il dominio di SEDI_D contiene insiemi di valori e perciò è non-atomico; in questo caso $\text{NUMERO_D} \rightarrow \text{SEDI_D}$, perché ogni insieme è considerato un unico membro del dominio dell'attributo.

Si consideri lo schema di relazione DIPARTIMENTO mostrato in Figura 1, la cui chiave primaria è NUMERO_D, e si supponga di estenderlo inserendo l'attributo SEDI_D come mostrato in Figura 8(a). Si supponga che ogni dipartimento possa avere *un certo numero* di sedi. Lo schema DIPARTIMENTO e un'estensione d'esempio sono mostrati in Figura 8. Come si può vedere, essa non è in 1NF perché SEDI_D non è un attributo atomico, come illustrato dalla prima tupla in Figura 8 (b).

In entrambi i casi la relazione DIPARTIMENTO di Figura 8 non è in 1NF; di fatto essa non si potrebbe definire relazione. Ci sono tre tecniche principali per raggiungere la prima forma normale per una relazione di questo tipo.

1. Si rimuova l'attributo SEDI_D che viola la 1NF e lo si ponga in una relazione separata SEDI_DIP insieme con la chiave primaria NUMERO_D di DIPARTIMENTO. La chiave primaria di questa relazione è la combinazione {NUMERO_D, SEDE_D}, come mostrato in Figura 2. In SEDI_DIP esiste una tupla distinta per *ogni sede* di un dipartimento. Ciò decompone la relazione che non è in 1NF in due relazioni in 1NF.
2. Si espanda la chiave in modo tale che ci sia una tupla separata nella relazione DIPARTIMENTO originaria per ogni sede di un DIPARTIMENTO, come mostrato in Figura 8(c). In questo caso la chiave primaria diventa la combinazione {NUMERO_D, SEDE_D}. Questa soluzione ha lo svantaggio di introdurre *ridondanza* nella relazione.
3. Se è noto il *numero di massimo di valori* dell'attributo – ad esempio, se è noto che per un dipartimento possono esistere *al più tre sedi* – si sostituisca l'attributo SEDI_D con tre attributi atomici: SEDE_D1, SEDE_D2, SEDE_D3. Questa soluzione ha lo svantaggio di introdurre *valori nulli* se la maggior parte dei dipartimenti ha meno di tre sedi.

Delle tre soluzioni sopra proposte, la prima risulta essere la migliore perché non presenta ridondanza ed è completamente generale, non presentando limiti sul numero massimo di valori. Infatti, se si sceglie la seconda soluzione, essa verrà ulteriormente decomposta, durante i successivi passi di normalizzazione, nella prima soluzione.

La prima forma normale non permette neppure attributi multivalori che siano essi stessi composti. Queste relazioni sono dette **relazioni nidificate** perché ogni tupla può avere *al suo interno* una relazione. In Figura 9 viene mostrato come potrebbe apparire la relazione IMP_PROG se fosse concessa la nidificazione. Ogni tupla rappresenta un'entità impiegato, e una relazione PROGG (NUMERO_P, ORE) *all'interno di ogni tupla* rappresenta i progetti su cui lavora l'impiegato e le ore settimanali lavorate dall'impiegato su ogni progetto. Lo schema di questa relazione IMP_PROG può essere rappresentato come segue:

IMP_PROG (SSN, NOME_I, {PROGG (NUMERO_P, ORE)})

Le parentesi graffe {} indicano che l'attributo PROGG è multivale, mentre gli attributi componenti che formano PROGG sono posti tra parentesi (). È interessante notare che una recente ricerca sul modello relazionale sta tentando di consentire e formalizzare le relazioni nidificate, che erano state inizialmente proibite dalla 1NF.

Si noti che SSN è la chiave primaria della relazione IMP_PROG nelle Figure 9(a) e (b), mentre NUMERO_P è la chiave primaria **parziale** della relazione nidificata; cioè la relazione nidificata deve avere valori univoci per NUMERO_P, all'interno di ogni tupla. Per normalizzare questa relazione in 1NF si spostano gli attributi della relazione nidificata in una relazione e *si propaga la chiave primaria* al suo interno; la chiave primaria della nuova relazione combinerà la chiave parziale con la chiave primaria della relazione originaria. La decomposizione e la propagazione della chiave primaria producono gli schemi IMP_PROG1 e IMP_PROG2 di Figura 9(c).

IMP_PROG			
SSN	NOME_I	PROGG	
		NUMERO_P	ORE

IMP_PROG			
SSN	NOME_I	NUMERO_P	ORE
123456789	Smith,John B.	1	32.5
		2	7.5
666884444	Narayan,Ramesh K.	3	40.0
453453453	English,Joyce A.	1	20.0
		2	20.0
333445555	Wong,Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya,Alicia J.	30	30.0
		10	10.0
987987987	Jabbar,Ahmed V.	10	35.0
		30	5.0
987654321	Wallace,Jennifer S.	30	20.0
		20	15.0
888665555	Borg,James E.	20	null

IMP_PROG1	
SSN	NOME_I

IMP_PROG2		
SSN	NUMERO_P	ORE

Figura 9 – Normalizzazione in 1NF di relazioni nidificate. (a) Schema della relazione IMP_PROG con una relazione nidificata PROGG. (b) Estensione d'esempio della relazione IMP_PROG che mostra relazioni nidificate all'interno di ciascun tupla. (c) Decomposizione di IMP_PROG nelle relazioni in 1NF IMP_PROG1 e IMP_PROG2 tramite propagazione della chiave primaria

Seconda forma normale

La **seconda forma normale (2NF)** si basa sul concetto di **dipendenza funzionale completa**. Una dipendenza funzionale $X \rightarrow Y$ è una **dipendenza funzionale completa** se la rimozione di qualsiasi attributo A da X comporta che la dipendenza funzionale non sussista più; cioè per ogni attributo $A \in X$, $(X - \{A\}) \rightarrow Y$ non determina

funzionalmente Y . Una dipendenza funzionale $X \rightarrow Y$ è una **dipendenza parziale** se si possono rimuovere da X certi attributi $A \in X$ e la dipendenza continua a sussistere; cioè, per qualche $A \in X$, $(X - \{A\}) \rightarrow Y$. In Figura 3(b) $\{\text{SSN}, \text{NUMERO_P}\} \rightarrow \text{ORE}$ è una dipendenza completa (non sussiste né $\text{SSN} \rightarrow \text{ORE}$ né $\text{NUMERO_P} \rightarrow \text{ORE}$). La dipendenza $\{\text{SSN}, \text{NUMERO_P}\} \rightarrow \text{NOME_I}$ è invece parziale perché sussiste la $\text{SSN} \rightarrow \text{NOME_I}$.

Il test per la 2NF comporta l'esame di dipendenze funzionali i cui attributi di parte sinistra fanno parte della chiave primaria. Se la chiave primaria contiene un solo attributo è inutile effettuare il test. Uno schema di relazione R è in **2NF** se ogni attributo non-primo A di R dipende funzionalmente in modo completo dalla chiave primaria di R . la relazione

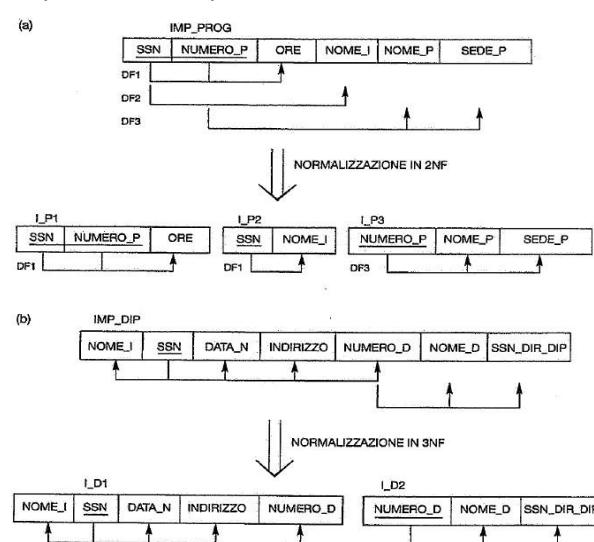


Figura 10 - Il processo di normalizzazione. (a) Normalizzazione di IMP_PROG in relazioni in 2NF. (b) Normalizzazione di IMP_DIP in relazioni in 3NF.

IMP_PROG di Figura 3(b) è in 1NF ma non è in 2NF. L'attributo non-primo NOME_I viola la 2NF a causa di DF2, come fanno gli attributi non-primi NOME_P e SEDE_P a causa di DF3. Le dipendenze funzionali DF2 e DF3 rendono NOME_I, NOME_P e SEDE_P parzialmente dipendenti dalla chiave primaria {SSN, NUMERO_P} di IMP_PROG, violando così il test di 2NF.

Se uno schema di relazione non è in 2NF, esso può essere “normalizzato in 2NF”, scomponendolo in un certo numero di relazioni in 2NF, nelle quali gli attributi non-primi sono associati solo alla parte della chiave primaria da cui sono funzionalmente dipendenti in modo completo. Le dipendenze funzionali DF1, DF2 e DF3 di Figura 3(b) portano perciò alla decomposizione di IMP_PROG nei tre schemi di relazione I_P1, I_P2 e I_P3 di Figura 10(a), ognuno dei quali è in 2NF.

Terza forma normale

La **terza forma normale (3NF)** è basata sul concetto di *dipendenza transitiva*. Una dipendenza funzionale $X \rightarrow Y$ in uno schema di relazione R è una **dipendenza transitiva** se esiste un insieme di attributi Z , che non è né una chiave candidata né un sottoinsieme di una chiave di R , per cui valgono contemporaneamente $X \rightarrow Z$ e $Z \rightarrow Y$. La dipendenza SSN \rightarrow SSN_DIR_DIP in IMP_DIP di Figura 3(a) è transitiva attraverso NUMERO_D perché sussistono entrambe le dipendenze SSN \rightarrow NUMERO_D e NUMERO_D \rightarrow SSN_DIR_DIP, e NUMERO_D non è né una chiave di per sé né un sottoinsieme della chiave di IMP_DIP. Intuitivamente si può vedere che la dipendenza di SSN_DIR_DIP da NUMERO_D in IMP_DIP è sgradita, dal momento che NUMERO_D non è una chiave di IMP_DIP.

Stando alla definizione originaria di Codd uno schema di relazione R è in **3NF** se soddisfa la 2NF e nessun attributo non-primo di R dipende in modo transitivo dalla chiave primaria. Lo schema di relazione IMP_DIP in Figura 3(a) è in 2NF, dal momento che non esistono dipendenze funzionali parziali da una chiave. IMP_DIP non è però in 3NF a causa della dipendenza transitiva di SSN_DIR_DIP (e anche NOME_D) da SSN attraverso NUMERO_D.

Si può normalizzare IMP_DIP decomponendolo nei due schemi di relazione in 3NF I_D1 e I_D2 mostrati in Figura 10(b). Intuitivamente si vede che ID_1 e ID_2 rappresentano fatti di entità indipendenti sugli impiegati e sui dipartimenti. Un’operazione di JOIN NATURALE su ID_1 e ID_2 recupererà la relazione originale IMP_DIP senza generare tuple spurie.

La Tabella 1 riassume informalmente le tre forme normali basate sulle chiavi primarie, i test usati nei vari casi e il corrispondente “rimedio” o normalizzazione per ottenere la forma normale.

Forma Normale	Test	Rimedio (Normalizzazione)
Prima (1NF)	La relazione non deve avere attributi non-atomici o relazioni nidificate.	Formare nuove relazioni per ogni attributo non-atomico o relazione nidificata.
Seconda (2NF)	Per relazioni in cui la chiave primaria contiene più attributi, nessun attributo non-chiave deve essere funzionalmente dipendente da una parte della chiave primaria.	Decomporre e preparare una nuova relazione per ogni chiave parziale con i suoi attributi dipendenti. Assicurarsi di mantenere una relazione con la chiave primaria originale e tutti gli attributi funzionalmente dipendenti in modo completo da essa.
Terza (3NF)	La relazione non deve avere un attributo non-chiave determinato funzionalmente da un altro attributo non-chiave (o da un insieme di attributi non-chiave). Cioè non deve esserci nessuna dipendenza transitiva di un attributo non-chiave dalla chiave primaria.	Decomporre e preparare una relazione che comprenda gli attributi non-chiave che determinano funzionalmente altri attributi non-chiave.

Tabella 1 – Sommario delle forme normali basate su chiavi primarie e corrispondente normalizzazione.

Definizioni generali di seconda e terza forma normale

In generale, si vogliono progettare schemi di relazione in modo tale che non presentino né dipendenze parziali né dipendenze transitive, perché questi tipi di dipendenze provocano le anomalie di aggiornamento discusse. I passi per la normalizzazione in relazione in 3NF trattati non consentono dipendenze parziali e transitive sulla *chiave primaria*. Queste definizioni, però, non tengono conto delle altre chiavi candidate di una relazione, ammesso che ve ne siano. Ora si forniranno le definizioni più generali di 2NF e 3NF, che tengono conto di *tutte* le chiavi candidate di una relazione. Si noti che ciò non riguarda la definizione di 1NF, dal momento che essa non dipende da chiavi e dipendenze funzionali. Come definizione generale di **attributo primo**, verrà considerato primo un attributo che faccia parte di *una qualsiasi chiave candidata*. Le dipendenze funzionali parziali e totali nonché le dipendenze transitive saranno ora *relative a tutte le chiavi candidate* di una relazione.

Definizione generale di seconda forma normale

Uno schema di relazione R è in **seconda forma normale (2NF)** se ogni attributo non-primo A di R non è parzialmente dipendente da *nessuna* chiave di R . si consideri lo schema di relazione LOTTI di Figura 11(a), che descrive appezzamenti di terreno in vendita in varie contee di uno stato. Si supponga che ci siano due chiavi candidate: #ID_PROPRIETA' e {NOME_CONTEA, #LOTTO}; ciò significa che i numeri di lotto sono univoci solo all'interno di ogni contea, ma i numeri di ID_PROPRIETA' sono univoci per tutte le contee dell'intero stato. Sulla base delle due chiavi candidate #ID_PROPRIETA' e {NOME_CONTEA, #LOTTO}, si sa che sussistono le dipendenze funzionali DF1 e DF2 di Figura 11(a). Si sceglie poi #ID_PROPRIETA' come chiave primaria, e perciò essa è sottolineata in figura, ma non verrà riservato nessun riguardo particolare ad essa rispetto all'altra chiave candidata.

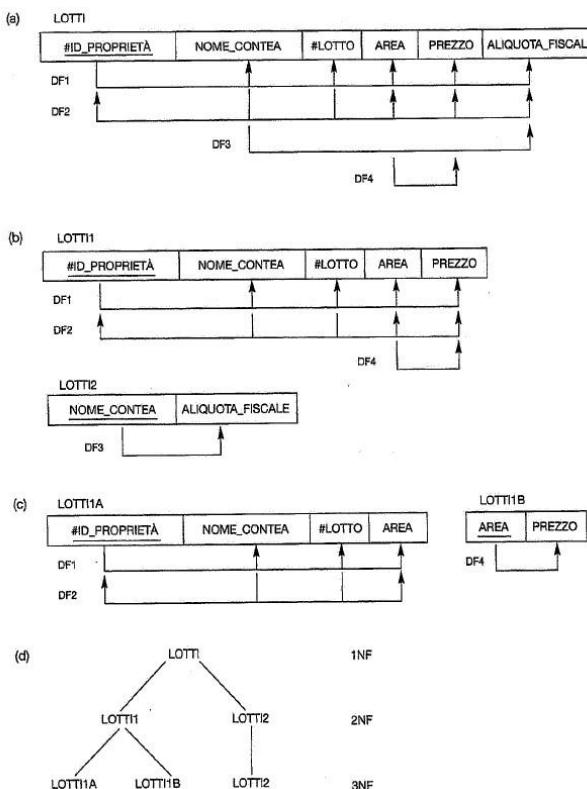


Figura 11 – Normalizzazione in 2NF e 3NF. (a) Lo schema di relazione LOTTI e le sue dipendenze funzionali da DF1 a DF4. (b) Decomposizione di LOTTI nelle relazioni in 2NF LOTTI1 e LOTTI2. (c) Decomposizione di LOTTI1 nelle relazioni in 3NF LOTTI1A e LOTTI1B. (d) Riassunto della normalizzazione di LOTTI

Definizione generale di terza forma normale

Uno schema di relazione R è in **terza forma normale (3NF)** se, ogni volta che sussiste in R una dipendenza funzionale *non-banale* $X \rightarrow A$, o (a) X è una superchiave di R , o (b) A è un attributo primo di R . Secondo questa definizione, LOTTI2 (Figura 11b) è in 3NF. Però DF4 in LOTTI1 viola la 3NF perché in questa relazione AREA non è una superchiave e PREZZO non è un attributo primo. Per normalizzare LOTTI1 in 3NF lo si decompone negli schemi di relazione LOTTI1A e LOTTI1B mostrati in Figura 11(c), LOTTI1A viene costruito rimuovendo l'attributo PREZZO che viola la 3NF da LOTTI1, e ponendolo con AREA (la parte sinistra di DF4 che provoca la dipendenza transitiva) in un'altra relazione LOTTI1B. Sia LOTTI1A sia LOTTI1B sono in 3NF. Ci sono due aspetti degni di nota al riguardo della definizione generale di 3NF:

- LOTTI1 viola la 3NF perché PREZZO è transitivamente dipendente da ognuna delle chiavi candidate di LOTTI1 tramite l'attributo non-primo AREA;
- Questa definizione può essere applicata *direttamente* per verificare se uno schema di relazione è in 3NF; *non* è necessario passare prima attraverso la 2NF. Se si applica la definizione di 3NF sopra vista a LOTTI con le dipendenze da DF1 a DF4, si trova che DF3 e DF4 violano *entrambe* la 3NF. Sarebbe perciò possibile decomporre direttamente LOTTI in LOTTI1A, LOTTI1B e LOTTI2. Quindi le dipendenze transitive e parziali che violano la 3NF possono essere rimosse *in qualsiasi ordine*.

Si supponga che in LOTTI sussistano le seguenti due dipendenze funzionali aggiuntive:

DF3: NOME_CONTEA \rightarrow ALIQUOTA_FISCALE

DF4: AREA \rightarrow PREZZO

A parole, la dipendenza DF3 dice che l'aliquote di tassazione è fissa per una data contea (non varia da lotto a lotto all'interno della stessa contea), mentre DF4 dice che il prezzo di un lotto è determinato dalla sua area indipendentemente dalla contea in cui esso si trova (si supponga che questo sia il prezzo del lotto a scopi fiscali). Lo schema di relazione LOTTI viola la definizione generale di 2NF perché ALIQUOTA_FISCALE dipende in modo parziale dalla chiave candidata (NOME_CONTEA, #LOTTO), per la presenza di DF3. Per normalizzare LOTTI in 2NF lo si decompone nelle due relazioni LOTTI1 e LOTTI2, mostrate in Figura 11(b). Si costruisce LOTTI1 rimuovendo da LOTTI l'attributo ALIQUOTA_FISCALE, che viola la 2NF, e ponendo con NOME_CONTEA (la parte sinistra di DF3 che causa la dipendenza parziale) in un'altra relazione LOTTI2. Sia LOTTI1 sia LOTTI2 sono in 2NF. Si noti che DF4 non viola la 2NF ed è riportata in LOTTI1.

Forma normale di Boyce e Codd

La **forma normale di Boyce e Codd (BCNF)** è stata proposta come una forma più semplice di 3NF, ma si è rivelata più restrittiva della 3NF; infatti, ogni relazione in BCNF è anche in 3NF, mentre una relazione in 3NF non è necessariamente in BCNF. Si può intuitivamente ravvisare la necessità di una forma normale più forte della 3NF tornando allo schema di relazione LOTTI di Figura 11(a), con le sue quattro dipendenze funzionali, da DF1 a DF4. Si supponga di avere migliaia di lotti nella relazione, ma che i lotti vengano solo da due contee: Dekalb e Fulton. Si supponga inoltre che le dimensioni possibili dei lotti nella contea di Dekalb siano ristrette a 0.5, 0.6, 0.7, 0.8, 0.9 e 1.0 acri, mentre le dimensioni possibili dei lotti nella contea di Fulton si limitano a 1.1, 1.2, ..., 1.9, 2.0 acri. In questa situazione si avrà la dipendenza funzionale aggiuntiva FD5: AREA → NOME_CONTEA. Se la si aggiunge alle altre dipendenze, lo schema di relazione LOTTI1A rimane in 3NF perché NOME_CONTEA è un attributo primo.

L'area di un lotto che determina la contea, come specificato da DF5, può essere rappresentata da 16 tuple in una relazione separata $R(\text{AREA}, \text{NOME_CONTEA})$, dal momento che ci sono solo 16 possibili valori per AREA. Questa rappresentazione riduce la ridondanza insita nel ripetere la stessa informazione in migliaia di tuple di LOTTI1A. La BCNF è una *forma normale più restrittiva*, che non permette LOTTI1A e fornisce indicazioni per decomporla.

La definizione formale di BCNF differisce solo leggermente dalla definizione di 3NF. Uno schema di relazione R è in **BCNF** se, ogni volta, che sussiste in R una dipendenza funzionale *non-banale* $X \rightarrow A$, X è una superchiave di R . La sola differenza tra le definizioni di BCNF e di 3NF è che la condizione (b) della 3NF, che consente ad A di essere primo, è assente dalla BCNF.

Nel nostro esempio DF5 viola la BCNF in LOTTI1A, perché AREA non è una superchiave di LOTTI1A. Si noti che DF5 soddisfa la 3NF in LOTTI1A perché NOME_CONTEA è un attributo primo (condizione b), ma in questa condizione non esiste nella definizione di BCNF. Si può decomporre LOTTI1A nelle due relazioni in BCNF LOTTI1AX e LOTTI1AY di Figura 12(a). Questa decomposizione perde la dipendenza funzionale DF2 perché i suoi attributi non coesistono più nella stessa relazione.

In pratica, la maggior parte degli schemi di relazione in 3NF è anche in BCNF. Solo se in uno schema di relazione R sussiste $X \rightarrow A$, con X che non è una superchiave e A attributo primo, R sarà in 3NF ma non in BCNF. Lo schema di relazione R di Figura 12(b) illustra il caso generale di una relazione di questo tipo. Idealmente la progettazione di una base di dati relazione dovrebbe sforzarsi di raggiungere la BCNF o la 3NF per ogni schema di relazione. Il raggiungimento dello stato di normalizzazione a livello della sola 1NF o 2NF

non è considerato sufficiente, dal momento che esse sono state storicamente sviluppate come punti di partenza verso la 3NF e la BCNF.

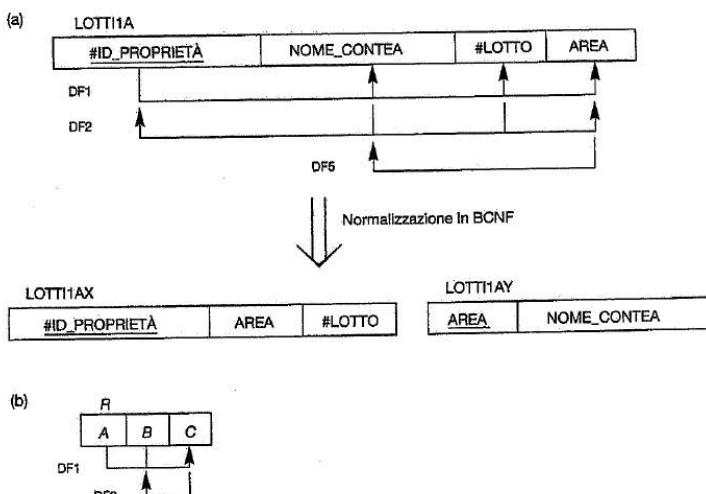


Figura 12 – Forma normale di Boyce e Codd. (a) Normalizzazione in BCNF con la dipendenza DF2 che viene “persa” nella decomposizione. (b) Una relazione in R in 3NF ma non in BCNF.

INSEGNA		
STUDENTE	INSEGNAMENTO	DOCENTE
Narayan	Basi di dati	Mark
Smith	Basi di dati	Navathe
Smith	Sistemi operativi	Ammar
Smith	Informatica teorica	Schulman
Wallace	Basi di dati	Mark
Wallace	Sistemi operativi	Ahamad
Wong	Basi di dati	Omieciński
Zelaya	Basi di dati	Navathe

Figura 13 – Una relazione INSEGNA che è in 3NF ma non in BCNF

In Figura 13 è mostrata una relazione INSEGNA con le seguenti dipendenza:

DF1: {STUDENTE, INSEGNAMENTO} → DOCENTE

DF2: DOCENTE → INSEGNAMENTO

Si noti che {STUDENTE, INSEGNAMENTO} è una chiave candidata per questa relazione e che le dipendenze viste seguono il modello illustrato in Figura 12(b). Pertanto, questa relazione è in 3NF ma non in BCNF. La decomposizione di questo schema di relazione in due schemi distinti non è immediata, dal momento che esso può essere decomposto in una delle tre coppie possibili:

1. {STUDENTE, DOCENTE} e {STUDENTE, INSEGNAMENTO};
2. {INSEGNAMENTO, DOCENTE} e {INSEGNAMENTO, STUDENTE};
3. {DOCENTE, INSEGNAMENTO} e {DOCENTE, STUDENTE}.

Tutte e tre le decomposizioni “perdonano” la dipendenza funzionale DF1. La decomposizione preferibile tra le tre sopra viste è la terza, perché non genererà tuple spurie dopo un join. Un test per determinare se una decomposizione è non-additiva (senza-perdita) sarà esaminato successivamente. In generale una relazione non in BCNF dovrebbe essere decomposta senza perdita, rinunciando eventualmente a preservare tutte le dipendenze funzionali nelle relazioni decomposte, come accade in questo esempio.

Algoritmi per la progettazione di DB Relazionali e ulteriori dipendenze

Come precedentemente visto, esistono fondamentalmente due approcci per la progettazione di basi di dati relazionali. Il primo consiste nella **progettazione top-down**, una tecnica ampiamente usata per le applicazioni commerciali di basi di dati che prevede la progettazione di uno schema concettuale in un modello di dati di alto livello, come il modello EER, e quindi una sua traduzione in un insieme di relazioni, attraverso varie procedure. Seguendo tale approccio ogni relazione viene analizzata sulla base delle dipendenze funzionali e delle chiavi primarie assegnati, applicando la procedura di normalizzazione, precedentemente descritta, per rimuovere dipendenze parziali e transitive, se ne rimangono. L'analisi di dipendenze indesiderate può anche essere svolta durante la progettazione concettuale, tramite un esame delle dipendenze funzionali presenti fra gli attributi di tipi di entità e tipi di associazione, ovviando così alla necessità di un'ulteriore normalizzazione successiva alla traduzione.

Il secondo approccio consiste nella **progettazione bottom-up**, una tecnica più purista che vede la progettazione di uno schema di base di dati relazionale rigorosamente in termini di dipendenze, funzionali e di altro tipo, specificate sugli attributi della base di dati. Dopo che i progettisti della base di dati hanno specificato le dipendenze, viene applicato un **algoritmo di normalizzazione** per sintetizzare gli schemi di relazione. Ogni singolo schema di relazione dovrebbe raggiungere il livello di qualità associato alla 3NF o alla BCNF, se non a qualche forma normale superiore. Successivamente verranno descritti alcuni di questi algoritmi. Si descriverà anche in maggior dettaglio le due proprietà desiderate di join non-additivo (senza perdita) e di conservazione delle dipendenze. Gli algoritmi di normalizzazione cominciano tipicamente sintetizzando uno schema di relazione gigante, detto **relazione universale**, che comprende tutti gli attributi della base di dati. Si eseguono quindi decomposizioni ripetute, basandosi sulle dipendenze funzionali specificate dal progettista di basi di dati, finché non diventino non più possibili o non più desiderabili. Si comincerà col descrivere le due **proprietà desiderabili delle decomposizioni** – vale a dire la proprietà di conservazione delle dipendenze e la proprietà di join senza perdita (o non-additivo), entrambe usante dagli algoritmi di progettazione per ottenere decomposizioni desiderabili. Si mostrerà anche come le forme normali siano *di per sé insufficienti* a garantire una buona progettazione di schemi di basi di dati relazioni: per poter essere considerate frutto di buona progettazione, infatti, le relazioni devono nell'insieme soddisfare queste due ulteriori proprietà.

Algoritmi per la progettazione di schemi di basi di dati relazionali

Decomposizione delle relazioni e insufficienza delle forme normali

Gli algoritmi di progettazione di basi di dati relazionali qui presentati prendono inizio da un unico **schema di relazione universale** $R = \{A_1, A_2, \dots, A_n\}$ contenente *tutti* gli attributi della base di dati. Si farà implicitamente l'**assunzione di relazione universale**, la quale stabilisce che ogni nome di attributo sia unico. L'insieme di F di dipendenze funzionali che deve sussistere sugli attributi di R è specificato dai progettisti della base di dati ed è fornito agli algoritmi di progettazione. Basandosi sulle dipendenze funzionali, gli algoritmi decompongono lo schema di relazione universale R in un insieme di schemi di relazione $D = \{R_1, R_2, \dots, R_m\}$ che diventerà lo schema della base di dati relazione; D è detto **decomposizione** di R .

È necessario assicurarsi che ogni attributo presente in R sia anche presente in almeno uno schema di relazione R_i nella decomposizione, così che non ci siano attributi “persi”; formalmente si ha

$$\bigcup_{i=1}^m R_i = R$$

Questa condizione è detta condizione di **conservazione degli attributi** di una decomposizione.

Un altro obiettivo è quello che ogni singola relazione R_i nella decomposizione D sia in BCNF (o in 3NF). Questa condizione non è però sufficiente a garantire di per sé una buona progettazione di basi di dati. Occorre infatti considerare la decomposizione nel complesso; oltre a esaminare le singole relazioni. Per illustrare questo punto, si consideri la relazione IMP_SEDI (NOME_I, SEDE_P) di Figura 5, che è sia in 3NF sia in BCNF. In realtà ogni schema di relazione con due soli attributi è automaticamente in BCNF. Anche se IMP_SEDI è in BCNF, essa dà comunque origine a tuple spurie quando viene unita tramite join con IMP_PROG1 (SSN, NUMERO_P, ORE, NOME_P, SEDE_P), che non è in BCNF (si veda il risultato del join naturale in Figura 6). IMP_SEDI rappresenta perciò uno schema di relazione particolarmente mal definito, a causa della sua semantica contorta in cui SEDE_P fornisce la sede di *uno dei progetti* sui quali lavora un impiegato. L'unione tramite join in IMP_SEDI con PROGETTO (NOME_P, NUMERO_P, SEDE_P, NUM_D) di Figura 2 – che è in BCNF – dà pure origine a tuple spurie. Si ha quindi bisogno di altri criteri che, insieme alle condizioni di 3NF e BCNF, evitino progetti di cattiva qualità di questo tipo. Nei tre paragrafi seguenti verranno esaminate le condizioni aggiuntive che devono sussistere su una decomposizione D considerata nel suo insieme.

Decomposizione e conservazione delle dipendenze

Sarebbe utile che ogni dipendenza funzionale $X \rightarrow Y$ specificata in F apparisse direttamente in uno degli schemi di relazione R_i della decomposizione D , oppure potesse essere inferita dalle dipendenze presenti in qualche R_i . Informalmente, quella enunciata è la *condizione di conservazione delle dipendenze*: si desidera conservare le dipendenze perché ogni dipendenza in F rappresenta un vincolo sulla base di dati. Se una delle dipendenze non è rappresentata in una singola relazione R_i della decomposizione, non è possibile imporre questo vincolo considerando una sola relazione; occorre piuttosto unire tramite join due o più relazioni della decomposizione e quindi verificare che nel risultato dell'operazione di join sussistano le dipendenze funzionali. Chiaramente questa è una procedura inefficiente e poco pratica.

Non è necessario che le dipendenze specificate in F si presentino esattamente nelle singole relazioni della decomposizione D . È sufficiente che l'unione delle dipendenze che sussistono sulle singole relazioni in D siano equivalenti a F . si darà ora definizione formale di questi concetti.

Dato un insieme di dipendenze F su R , la **proiezione** di F su R_i , denotata con $\pi_{R_i}(F)$, dove R_i è un sottoinsieme di R , è l'insieme di dipendenze $X \rightarrow Y$ di F^+ tali che gli attributi di $X \cup Y$ siano tutti contenuti in R_i . Perciò la proiezione di F su ogni schema di relazione R_i della decomposizione D è l'insieme delle dipendenze funzionali in F^+ , chiusura di F , tali che tutti i loro attributi di parte sinistra e di parte destra siano in R_i . Si dirà che una decomposizione $D = \{R_1, R_2, \dots, R_m\}$ di R **conserva le dipendenze** rispetto a F se l'unione delle proiezioni di F su ogni R_i in D è equivalente a F ; cioè

$$((\pi_{R_1}(F)) \cup \dots \cup (\pi_{R_m}(F)))^+ = F^+$$

Se una decomposizione non conserva le dipendenze, qualche dipendenza viene **persa** nella decomposizione. Come detto prima, per verificare se una dipendenza persa sussiste comunque, occorre effettuare il JOIN di due o più relazioni presenti nella decomposizione, fino a ottenere una relazione che presenti tutti gli attributi di parte sinistra e di parte destra nella dipendenza persa, e quindi verificare se nel risultato del JOIN sussiste la dipendenza – un'operazione poco pratica.

Un esempio di decomposizione che non conserva le dipendenze è presentato in Figura 12(a), in cui la dipendenza funzionale DF2 è persa quando LOTTI1A viene decomposta in {LOTTI1AX, LOTTI1AY}. Le decomposizioni di Figura 11, invece, conservano le dipendenze. Analogamente, per l'esempio di Figura 13, indipendentemente da quale delle tre decomposizioni mostrate venga scelta per la relazione INSEGNA{STUDENTE, INSEGNAMENTO, DOCENTE}, una o entrambe le dipendenze originariamente presenti vengono perse. Si enuncia qu una proposizione relativa a tale proprietà senza darne la dimostrazione.

Proposizione 1: È sempre possibile trovare una decomposizione D che conserva le dipendenze rispetto a F e tale che ogni relazione R_i di D sia in 3NF.

L'algoritmo 1.1 crea, a partire da un insieme di dipendenze funzionali F , una decomposizione $D = \{R_1, R_2, \dots, R_m\}$ di una relazione universale R , che conserva le dipendenze e tale che ogni R_i di D sia in 3NF. Esso garantisce solo la proprietà di conservazione delle dipendenze, *non* la proprietà di join senza perdita. Il primo passo dell'algoritmo 1.1 consiste nel trovare una copertura minimale G di F .

Input: Una relazione universale R e un insieme di dipendenze funzionali F sugli attributi di R .

1. Si trovi una copertura minimale G per F (si usi l'Algoritmo 10.2);
2. Per ogni parte sinistra X di una dipendenza funzionale presente in G , si crei uno schema di relazione in D con attributi $\{X \cup \{A_1\} \cup \{A_2\} \dots \cup \{A_k\}\}$, dove $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_k$ sono le sole dipendenze in G con X come parte sinistra (X è la CHIAVE di questa relazione);
3. Si pongano tutti gli attributi rimanenti (che non sono stati posti in nessuna relazione) in un solo schema di relazione per garantire la proprietà di conservazione degli attributi.

Algoritmo 1.1 - Algoritmo di sintesi relazionale con conservazione delle dipendenze.

Proposizione 1A: Ogni schema di relazione creato dall'Algoritmo 1.1 è in 3NF.

È ovvio che l'algoritmo conserva tutte le dipendenze presenti in G , perché ogni dipendenza compare in una delle relazioni R_i della decomposizione D . Dal momento che G è equivalente a F , tutte le dipendenze in F sono direttamente conservative nella decomposizione oppure sono derivabili da quelle presenti nelle relazioni risultati, assicurando così la proprietà di conservazione delle dipendenze. L'Algoritmo 1 è detto **algoritmo di sintesi relazionale**, perché ogni schema di relazione R_i nella decomposizione è *sintetizzato* (costruito) a partire dall'insieme di dipendenze funzionale in G con la stessa parte sinistra X .

Copertura minimale

Un insieme F di dipendenze funzionali è **minimale** se soddisfa le seguenti condizioni:

1. Ogni dipendenza presente in F ha come parte destra un solo attributo.
2. Non è mai possibile sostituire una dipendenza $X \rightarrow A$ di F con una dipendenza $Y \rightarrow A$, dove Y è un sottoinsieme proprio di X , e avere ancora un insieme di dipendenze equivalenti a F ;
3. Non è mai possibile rimuovere una dipendenza da F e avere ancora un insieme di dipendenze equivalenti a F .

Una **copertura minimale** di un insieme F di dipendenze funzionali è un insieme minimale di dipendenze F_{min} equivalenti a F . Purtroppo per uno stesso insieme di dipendenze funzionali ci possono essere molte coperture minimali. Usando l'Algoritmo 2 si può sempre trovare *almeno una* copertura minimale G per ogni insieme F di dipendenze.

1. Poni $G := F$.
2. Sostituisci ogni dipendenza funzionale $X \rightarrow \{A_1, A_2, \dots, A_n\}$ in G con le n dipendenze funzionali $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n$.
3. Per ogni dipendenza funzionale $X \rightarrow A$ in G
per ogni attributo B che sia un elemento di X
se $((G - \{X \rightarrow A\}) \cup \{(X - \{B\}) \rightarrow A\})$ è equivalente a G ,
allora sostituisci $X \rightarrow A$ con $(X - \{B\}) \rightarrow A$ in G .
4. Per ogni dipendenza funzionale rimanente $X \rightarrow A$ in G
se $(G - \{X \rightarrow A\})$ è equivalente a G ,
allora rimuovi $X \rightarrow A$ da G .

Algoritmo 2 - Ricerca di una copertura minimale G di F .

Decomposizione e Join senza perdita (non-additivi)

Un'altra proprietà che una decomposizione D deve soddisfare è quella di join senza perdita o join non-additivo, che assicura che non vengano generate tuple spurie quando alle relazioni della decomposizione viene applicata un'operazione di JOIN NATURALE. Il problema è già stato illustrato in precedenza, con l'esempio delle Figure 5 e 6. Dato che si tratta della proprietà di una decomposizione di *schemi* di relazione, la condizione di assenza di tuple spurie deve sussistere per *ogni stato valido di relazione* – cioè per ogni stato di relazione che soddisfa le dipendenze funzionali in F . La proprietà di join senza perdita è perciò sempre definita rispetto a un insieme specifico F di dipendenze. Formalmente, una decomposizione $D = \{R_1, R_2, \dots, R_m\}$ di R soddisfa la **proprietà di join senza perdita (non-additivo)** rispetto all'insieme di dipendenze F di R se, per *ogni* stato di relazione r di R che soddisfa F , sussiste quanto segue, dove $*$ è il JOIN NATURALE di tutte le relazioni in D :

$$*(\pi_{R_1}(r), \dots, \pi_{R_m}(r)) = r$$

La parola perdita in *senza perdita* si riferisce a *perdita di informazione*, non a perdita di tuple. Se una decomposizione non soddisfa la proprietà di join senza perdita, è possibile che si presentino tuple spurie aggiuntive, dopo che sono state eseguire le operazioni di PROIEZIONE (π) e di JOIN NATURALE (*); queste tuple aggiuntive rappresentano un'informazione errata. Si preferisce qui la locuzione **join non-additivo** perché essa descrive più fedelmente la situazione; se per una decomposizione sussiste questa proprietà è sicuro che, dopo che sono state eseguite le operazioni di PROIEZIONE e JOIN NATURALE, non verranno *aggiunte* al risultato tuple spurie, che portano informazioni sbagliate.

Ovviamente la decomposizione di IMP_PROG(SSN, NUMERO_P, ORE, NOME_I, NOME_P, SEDE_P), di Figura 3, in IMP_SEDI(NOME_I, SEDE_P) e IMP_PROG1(SSN, NUMERO_P, ORE, NOME_P, SEDE_P), presente in Figura 5, non soddisfa la proprietà di join senza perdita, come illustrato in Figura 6. Per verificare se una data decomposizione D soddisfa la proprietà di join senza perdita, rispetto a un insieme F di dipendenze funzionali, si può usare l'algoritmo 1.2:

Input: una relazione universale R , una decomposizione $D = \{R_1, R_2, \dots, R_m\}$ di R , e un insieme F di dipendenze funzionali.

1. Si costruisca una matrice iniziale S con una riga i per ogni relazione R_i di D , e una colonna j per ogni attributo A_j di R .
2. Si ponga $S(i, j) := B_{ij}$ per ogni elemento della matrice.
(* ogni B_{ij} è un simbolo distinto associato agli indici (i, j) *)
3. Per ogni riga i che rappresenta lo schema di relazione R_i
{per ogni colonna j che rappresenta l'attributo A_j ,
{se (la relazione R_i contiene l'attributo A_j) allora si
ponga $S(i, j) := A_j$;};};
(* ogni A_j è un simbolo distinto associato all'indice (j) *)
4. Si ripeta il ciclo seguente finché l'*ESECUZIONE DI UN CICLO COMPLETO* non dà luogo ad alcun cambiamento in S
{per ogni dipendenza funzionale $X \rightarrow Y$ in F
{per tutte le righe di S CHE PRESENTANO GLI STESSI SIMBOLI nelle colonne corrispondenti ad attributi presenti in X
{si faccia sì che i simboli in ogni colonna corrispondente a un attributo di Y siano gli stessi per tutte queste righe nel modo seguente:
se una riga presenta un simbolo "A" in quella colonna, allora si pongano le altre righe uguali, nella colonna, allo STESSO simbolo "A". Se non esiste alcun simbolo "A" per l'attributo in nessuna riga, si scelga per l'attributo uno dei simboli "B" presenti in una delle righe e si ponga nelle altre righe lo stesso simbolo "B" per quella colonna};};};
5. Se una riga è costituita per intero di simboli "A", allora la decomposizione ha la proprietà di join senza perdita; altrimenti non ha questa proprietà.

Algoritmo 1.2 – Verifica della proprietà di join senza perdita (non-additivo).

Data una relazione R che è decomposta in un certo numero di relazioni R_1, R_2, \dots, R_m , l'Algoritmo 1.2 comincia col costruire uno stato di relazione r nella matrice S . La riga i in S rappresenta una tupla t_i (relativa alla relazione R_i) che ha simboli "a" nelle colonne corrispondenti agli attributi di R_i e simboli "b" nelle altre colonne. L'algoritmo quindi trasforma le righe di questa matrice (durante il ciclo del passo 4) in modo tale che esse rappresentino tuple che soddisfano tutte le dipendenze funzionali in F . Alla fine del ciclo di applicazione delle dipendenze funzionali, ogni coppia di righe in S – che rappresentano due tuple di r – che si accordano sui valori degli attributi di parte sinistra X di una dipendenza funzionale $X \rightarrow Y$ di F , si accorderanno anche sui valori degli attributi di parte destra Y . Si può dimostrare che se, dopo aver applicato il ciclo del passo 4, una qualsiasi riga di S termina con tutti simboli "a", allora la decomposizione D soddisfa la proprietà di join senza perdita rispetto a F . Se, da altro lato, nessuna riga ha alla fine tutti i simboli "a", allora D non soddisfa la proprietà di join senza perdita. In quest'ultimo caso lo stato di relazione r rappresentato da S alla fine dell'algoritmo sarà un esempio di stato di relazione r di R che soddisfa le dipendenze in F ma non soddisfa la condizione di join senza perdita; perciò questa relazione serve come controesempio che prova che D non

ha la proprietà di join senza perdita rispetto a F . Si noti che i simboli "a" e "b" alla fine dell'algoritmo non hanno alcun significato particolare.

La Figura 1.1(a) mostra come si applica l'Algoritmo 1.2 alla decomposizione dello schema di relazione IMP_PROG di Figura 3(b) nei due schemi di relazione IMP_PROG1 e IMP_SEDI di Figura 5(a). Il ciclo al passo 4 dell'algoritmo non può cambiare nessun simbolo "b" in un simbolo "a"; perciò la matrice S risultante non ha una riga di soli simboli "a" e quindi la decomposizione non gode della proprietà di join senza perdita.

La Figura 1.1(b) mostra un'altra decomposizione IMP_PROG in IMP, PROGETTO e LAVORA_SU che gode della proprietà di join senza perdita, e la Figura 1.1(c) mostra come si può applicare l'algoritmo a questa decomposizione.

- (a) $R = \{\text{SSN}, \text{NOME_I}, \text{NUMERO_P}, \text{NOME_P}, \text{SEDE_P}, \text{ORE}\}$ $D = \{R_1, R_2\}$
 $R_1 = \text{IMP_SEDI} = \{\text{NOME_I}, \text{SEDE_P}\}$
 $R_2 = \text{IMP_PROG1} = \{\text{SSN}, \text{NUMERO_P}, \text{ORE}, \text{NOME_P}, \text{SEDE_P}\}$

$$F = \{\text{SSN} \rightarrow \text{NOME_I}; \text{NUMERO_P} \rightarrow \{\text{NOME_P}, \text{SEDE_P}\}; \{\text{SSN}, \text{NUMERO_P}\} \rightarrow \text{ORE}\}$$

	SSN	NOME_I	NUMERO_P	NOME_P	SEDE_P	ORE
R_1	b ₁₁	a ₂	b ₁₃	b ₁₄	a ₅	b ₁₆
R_2	a ₁	b ₂₂	a ₃	a ₄	a ₅	a ₆

(nessun cambiamento alla matrice dopo aver applicato le dipendenze funzionali)

(b)

IMP		PROGETTO			LAVORA_SU		
SSN	NOME_I	NUMERO_P	NOME_P	SEDE_P	SSN	NUMERO_P	ORE

- (c) $R = \{\text{SSN}, \text{NOME_I}, \text{NUMERO_P}, \text{NOME_P}, \text{SEDE_P}, \text{ORE}\}$ $D = \{R_1, R_2, R_3\}$
 $R_1 = \text{IMP} = \{\text{SSN}, \text{NOME_I}\}$
 $R_2 = \text{PROG1} = \{\text{NUMERO_P}, \text{NOME_P}, \text{SEDE_P}\}$
 $R_3 = \text{LAVORA_SU} = \{\text{SSN}, \text{NUMERO_P}, \text{ORE}\}$

$$F = \{\text{SSN} \rightarrow \text{NOME_I}; \text{NUMERO_P} \rightarrow \{\text{NOME_P}, \text{SEDE_P}\}; \{\text{SSN}, \text{NUMERO_P}\} \rightarrow \text{ORE}\}$$

	SSN	NOME_I	NUMERO_P	NOME_P	SEDE_P	ORE
R_1	a ₁	a ₂	b ₁₃	b ₁₄	b ₁₅	b ₁₆
R_2	b ₂₁	b ₂₂	a ₃	a ₄	a ₅	b ₂₆
R_3	a ₁	b ₃₂	a ₃	b ₃₄	b ₃₅	a ₆

(matrice originale S all'inizio dell'algoritmo)

	SSN	NOME_I	NUMERO_P	NOME_P	SEDE_P	ORE
R_1	a ₁	a ₂	b ₁₃	b ₁₄	b ₁₅	b ₁₆
R_2	b ₂₁	b ₂₂	a ₃	a ₄	a ₅	b ₂₆
R_3	a ₁	b ₃₂	a ₃	b ₃₄	b ₃₅	a ₆

(matrice S dopo l'applicazione delle prime due dipendenze funzionali – l'ultima riga è costituita da tutti simboli "a", e pertanto ci fermiamo)

Figura 1.1 – L'algoritmo di test per il join senza perdita. (a) Applicazione dell'algoritmo per controllare la decomposizione di IMP_PROG in IMP_PROG1 e IMP_SEDI. (b) Un'altra decomposizione di IMP_PROG. (c) Applicazione dell'algoritmo alla decomposizione in Figura 1.1 (b)

Una volta che una riga consista solo di simboli “a”, si sa che la decomposizione gode della proprietà di join senza perdita, ed è possibile smettere di applicare le dipendenze funzionali (passo 4 dell’algoritmo) alla matrice S . L’Algoritmo 1.2 ci consente di controllare se una specifica decomposizione D soddisfa la proprietà di join senza perdita rispetto a un insieme F di dipendenze funzionali. La domanda successiva è se esiste un algoritmo per decomporre uno schema di relazione universale $R = \{ A_1, A_2, \dots, A_n \}$ in $D = \{ R_1, R_2, \dots, R_m \}$, in modo che ogni R_i sia in BCNF e la decomposizione D gode della proprietà di join senza perdita rispetto a F . La risposta è sì, ma prima di descrivere l’algoritmo occorre presentare alcune proprietà generali delle decomposizioni con join senza perdita. La prima proprietà riguarda le **decomposizioni binarie** – decomposizioni di una relazione R in due relazioni. Essa fornisce un test più semplice da eseguire rispetto all’Algoritmo 1.2, ma è *limitata* alle sole decomposizioni binarie.

PROPRIETÀ LJ1.

Una decomposizione $D = \{ R_1, R_2 \}$ di R soddisfa la proprietà di join senza perdita rispetto a un insieme di dipendenze funzionali F di R se e solo se

- la DF $((R_1 \cap R_2)) \rightarrow (R_1 - R_2)$ è in F^* , oppure
- la DF $((R_1 \cap R_2)) \rightarrow (R_2 - R_1)$ è in F^* .

La seconda proprietà tratta l’esecuzione di decomposizioni successive.

PROPRIETÀ LJ2.

Una decomposizione $D = \{ R_1, R_2, \dots, R_m \}$ di R gode della proprietà di join senza perdita rispetto a un insieme di dipendenze funzionali F su R , e se una decomposizione $D_1 = \{ Q_1, Q_2, \dots, Q_k \}$ di R_i gode della proprietà di join senza perdita rispetto alla proiezione di F su R_i , allora la decomposizione

$D_2 = \{ R_1, R_2, \dots, R_{i-1}, Q_1, Q_2, \dots, Q_k, R_{i+1}, \dots, R_m \}$ di R gode della proprietà di join senza perdita rispetto a F .

La proprietà LJ2 sostiene che, se una decomposizione D soddisfa già la proprietà di join senza perdita – rispetto a F – e noi decomponiamo ulteriormente uno degli schemi di relazione R_i di D in un’altra decomposizione D_1 che gode della proprietà di join senza perdita – rispetto a $\pi_{R_i}(F)$ – allora la sostituzione di R_i in D con D_1 avrà come risultato una decomposizione che gode ancora della proprietà di join senza perdita – rispetto a F . Tale proprietà è stata qui assunta implicitamente negli esempi informali di normalizzazione presenti in precedenza. Ad esempio, in Figura 11, quando si è normalizzata la relazione LOTTI in LITT1 e LOTTI2, si è assunto che questa decomposizione fosse senza perdita. Decomponendo ulteriormente LOTTI1 in LOTTI1A e LOTTI1B si ottengono tre relazioni: LOTTI1A, LOTTI1B e LOTTI2; quest’ultima decomposizione continua a essere senza perdita in virtù della proprietà LJ2 sopra esposta.

L’Algoritmo 1.3 utilizza le proprietà LJ1 e LJ2 per creare una decomposizione con join senza perdita di una relazione universale R , $D = \{ R_1, R_2, \dots, R_m \}$, basata su un insieme di dipendenze funzionali F e tale che ogni R_i di D sia in BCNF.

Input: Una relazione universale R e un insieme di dipendenze funzionali F sugli attributi di R .

1. Poni $D := \{R\};$
2. Fino a che c’è uno schema di relazione Q in D non in BCNF esegui


```

      {
        scegli uno schema di relazione Q in D che non sia in BCNF;
        trova una dipendenza funzionale X → Y in Q che viola la BCNF;
        sostituisci, in D, Q con due schemi di relazione (Q - Y) e (X ∪ Y);
      };
    
```

Algoritmo 1.3 – Decomposizione relazionale in relazioni in BCNF con la proprietà di join senza perdita.

Ogni volta che si attraversa il ciclo presente nell'Algoritmo 1.3 si decomponete uno schema di relazione Q che non è in BCNF in due schemi di relazione. secondo la LJ1 e la LJ2, la decomposizione D ha la proprietà di join senza perdita. Al termine dell'algoritmo tutti gli schemi di relazione in D saranno in BCNF. Si può verificare che l'esempio di normalizzazione delle Figure 11 e 12 segue sostanzialmente questo algoritmo. Le dipendenze funzionali DF3, DF4 e, più tardi, DF5, violano la BCNF e inoltre la decomposizione soddisfa la proprietà di join senza perdita. Analogamente, se si applica l'algoritmo allo schema di relazione INSEGNA di Figura 13, esso viene decomposto in INSEGNA1 (DOCENTE, STUDENTE) e INSEGNA2 (DOCENTE, INSEGNAMENTO) dato che la dipendenza DF2: DOCENTE → INSEGNANTE viola la BCNF.

Al passo 2 dell'Algoritmo 1.3 è necessario determinare se uno schema di relazione Q è in BCNF oppure no. Un metodo per fare ciò consiste nel verificare, per ogni dipendenza funzionale $X \rightarrow Y$ in Q , se X^+ non comprende tutti gli attributi di Q . Se le cose stanno così, allora $X \rightarrow Y$ viola la BCNF perché X in questo caso non può essere una (super)chiave di Q . Un'altra tecnica è basata sull'osservazione che, ogni volta che uno schema di relazione Q viola la BCNF, esiste una coppia di attributi A e B di Q tale che $\{Q - \{A - B\}\} \rightarrow A$; calcolando la chiusura $\{Q - \{A - B\}\}^+$ per ogni coppia di attributi $\{A, B\}$ di Q , e verificando se la chiusura comprende A (o B), è possibile determinare se Q è in BCNF.

Se si vuole che una decomposizione goda della proprietà di join senza perdita e conservi le dipendenze, occorre accontentarsi di schemi di relazione in 3NF anziché in BCNF. Una semplice variazione dell'Algoritmo 1.1 (si veda l'Algoritmo 1.4), porta a una decomposizione D di R con le seguenti proprietà:

- conserva le dipendenze,
- gode della proprietà di join senza perdita,
- è tale che ogni schema di relazione risultante nella decomposizione sia in 3NF.

Input: Una relazione universale R e un insieme di dipendenze funzionali F sugli attributi di R .

1. Si trovi una copertura minimale G di F (si usi l'Algoritmo 10.2).
2. Per ogni parte sinistra X di una dipendenza funzionale che compare in G si costruisca uno schema di relazione in D con attributi $\{X \cup \{A_1\} \cup \{A_2\} \dots \cup \{A_k\}\}$, dove $X \rightarrow A_1$, $X \rightarrow A_2$, ..., $X \rightarrow A_k$ sono le sole dipendenze di G con X come parte sinistra (X è la CHIAVE di questa relazione).
3. Se nessuno degli schemi di relazione in D contiene una chiave di R , allora si costruisca un altro schema di relazione in D che contenga attributi che formano una chiave di R .

Algoritmo 1.4 - Algoritmo di sintesi relazionale con conservazione delle dipendenze e proprietà di join senza perdita

Si può dimostrare che la decomposizione formata dall'insieme di schemi di relazione costruiti dall'algoritmo precedente conserva le dipendenze e gode della proprietà di join senza perdita. Inoltre, ogni schema di relazione nella decomposizione è in 3NF. Questo algoritmo è un miglioramento dell'Algoritmo 1.1, dato che quest'ultimo garantiva solo la conservazione delle dipendenze.

Il passo 3 dell'Algoritmo 1.4 prevede l'individuazione di una chiave K di R . Per individuare una chiave K di R sulla base dell'insieme dato F di dipendenze funzionali può essere usato l'Algoritmo 1.4a. Si comincia ponendo K uguale a tutti gli attributi di R ; quindi si rimuove un attributo alla volta e si verifica se i restanti attributi formano ancora una superchiave. Si noti che l'insieme di dipendenze funzionali usato per determinare una chiave nell'Algoritmo 1.4a può essere F o G , dal momento che essi sono equivalenti. Si noti, inoltre, che l'Algoritmo 1.4a determina solo *una chiave* fra le possibili chiavi candidate di R ; la chiave fornita dipende dall'ordine con cui gli attributi sono rimossi da R al passo 2.

1. Si ponga $K := R$.
2. Per ogni attributo A in K
 {si calcoli $(K - A)^+$ rispetto a F ;
 Se $(K - A)^+$ contiene tutti gli attributi di R , allora si
 ponga $K := K - \{A\}$ };

Algoritmo 1.4a - Ricerca di una chiave K per lo schema di relazione R sulla base di un insieme F di dipendenze funzionali

Non è sempre possibile trovare una decomposizione in schemi di relazione che conservi le dipendenze e che consenta a ogni schema di relazione della decomposizione di essere in BCNF (anziché in 3NF come nell'Algoritmo 1.4). Si può pensare di controllare gli schemi di relazione in 3NF della decomposizione uno a uno per vedere se soddisfano la BCNF. Se qualche schema di relazione R_i non è in BCNF si può scegliere di decomporlo ulteriormente o di lasciarlo così com'è in 3NF (con qualche possibile anomalia di aggiornamento). Il fatto che non si possa sempre trovare una decomposizione in schemi di relazione in BCNF che conservi le dipendenze può essere evidenziato dagli esempi in Figura 12. Le relazioni LOTTI1A (Figura 12a) e INSEGNA (Figura 13) non sono in BCNF, ma sono in 3NF. Ogni tentativo di decomporre ulteriormente queste relazioni in relazioni in BCNF ha come risultato una perdita della dipendenza DF2:

$\{\text{NOME_CONTEA}, \#\text{LOTTO}\} \rightarrow \{\#\text{ID_PROPRIETA'}, \text{AREA}\}$ in LOTTI1A

o una perdita di DF1:

$\{\text{STUDENTE}, \text{INSEGNAMENTO}\} \rightarrow \text{DOCENTE}$ in INSEGNA.

È importante sottolineare che la teoria delle decomposizioni con join senza perdita si basa sull'assunzione che *non sono consentiti valori nulli per gli attributi di join*. Il prossimo paragrafo affronta alcuni problemi che nelle decomposizioni relazionali possono essere causati da valori nulli.

Problemi con valori nulli e tuple dangling

Quando si progetta uno schema di base di dati relazionale bisogna valutare attentamente i problemi associati alla presenza di valori nulli. A tutt'oggi non è stata presentata nessuna teoria di progettazione relazionale pienamente soddisfacente che comprenda i valori nulli. Un problema si verifica quando alcune tuple presentano valori nulli per attributi che saranno usati per effettuare il JOIN di singole relazioni della decomposizione. Per illustrare ciò si consideri la base di dati mostrata in Figura 1.2(a), in cui sono presenti le due relazioni IMPIEGATO e DIPARTIMENTO. Le ultime due tuple impiegato – Berger e Benitez – rappresentano impiegati appena assunti che non sono stati ancora assegnati a un dipartimento (si dia per scontato che ciò non violi alcun vincolo di integrità). Si supponga ora di voler recuperare un elenco di valori di $(\text{NOME_I}, \text{NOME_D})$ per tutti gli impiegati. Se si esegue l'operazione di JOIN NATURALE su IMPIEGATO e DIPARTIMENTO (Figura 1.2b), le due tuple suddette *non* appariranno nel risultato. Con l'operazione di JOIN ESTERNO si può affrontare questo problema. Si ricordi che, se si considera il JOIN ESTERNO SINISTRO di IMPIEGATO con DIPARTIMENTO, le tuple di IMPIEGATO che presentano un valore nullo per l'attributo di join appariranno comunque nel risultato, congiunte (*joined*) con una tupla "immaginaria" di DIPARTIMENTO che presenta valori nulli per tutti i suoi attributi. La Figura 1.2(c) mostra il risultato.

In generale, ogni volta che viene progettato uno schema di base di dati relazionale in cui due o più relazioni sono collegate tramite chiavi esterne, deve essere dedicata particolare attenzione a ricercare potenziali valori nulli nelle chiavi esterne. Ciò può infatti causare un'inaspettata perdita d'informazione nelle interrogazioni che prevedono join su quella chiave esterna. Inoltre, se si presentano valori nulli in altri attributi, come STIPENDIO, il loro effetto su funzioni built-in(integrate) come SUM e AVERAGE deve essere attentamente valutato.

Un problema collegato è quello delle **tuple dangling** (dondolanti), che possono presentarsi se si spinge troppo oltre una decomposizione. Si supponga di decomporre ulteriormente la relazione IMPIEGATO di Figura 1.2(a)

Lezione 02 - Algoritmi per la progettazione di DB Relazionali e ulteriori dipendenze

nelle relazioni IMPIEGATO_1 e IMPIEGATO_2, mostrate un Figura 1.3(a) e 1.3(b). Se si applica l'operazione di JOIN NATURALE a IMPIEGATO_1 e IMPIEGATO_2, si ottiene la relazione IMPIEGATO originale. Si può però usare la rappresentazione alternativa mostrata in Figura 1.3(c) , in cui *non si inserisce una tupla* in IMPIEGATO_3 se all'impiegato non è stato assegnato un dipartimento (anziché inserire una tupla con valore nullo per NUM_D come in IMPIEGATO_2). Se si usa IMPIEGATO_3 anziché IMPIEGATO_2 e si esegue un JOIN NATURALE su IMPIEGATO_1 e IMPIEGATO_3, le tuple relative a Berger e Benitez non saranno presenti nel risultato; queste sono dette **tuple dangling** perché sono rappresentate in una sola delle due relazioni che rappresentano impiegati e pertanto vengono perse se si esegue un'operazione di join (interno).

(a) **IMPIEGATO**

NOME_I	SSN	DATA_N	INDIRIZZO	NUM_D
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4
Narayan, Ramesh K.	666884444	1982-09-15	975 Fire Oak, Humble, TX	5
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1
Berger, Anders C.	999776555	1965-04-26	6530 Braes, Bellaire, TX	null
Benitez, Carlos M.	888664444	1963-01-09	7654 Beech, Houston, TX	null

DIPARTIMENTO

NOME_D	NUM_D	SSN_DIR_DIP
Ricerca	5	333445555
Amministrazione	4	987654321
Sede centrale	1	888665555

(b)

NOME_I	SSN	DATA_N	INDIRIZZO	NUM_D	NOME_D	SSN_DIR_DIP
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Ricerca	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Ricerca	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Amministrazione	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Amministrazione	987654321
Narayan, Ramesh K.	666884444	1982-09-15	975 Fire Oak, Humble, TX	5	Ricerca	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Ricerca	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Amministrazione	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Sede centrale	888665555

(c)

NOME_I	SSN	DATA_N	INDIRIZZO	NUM_D	NOME_D	SSN_DIR_DIP
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Ricerca	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Ricerca	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Amministrazione	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Amministrazione	987654321
Narayan, Ramesh K.	666884444	1982-09-15	975 Fire Oak, Humble, TX	5	Ricerca	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Ricerca	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Amministrazione	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Sede centrale	888665555
Berger, Anders C.	999776555	1965-04-26	6530 Braes, Bellaire, TX	null	null	null
Benitez, Carlos M.	888664444	1963-01-09	7654 Beech, Houston, TX	null	null	null

Figura 1.2 - Illustrazione dei problemi relativi al join con valori nulli. (a) Una base di dati con valori nulli per alcuni attributi di join. (b) Risultato dell'applicazione del JOIN NATURALE alle relazioni IMPIEGATO e DIPARTIMENTO. (c) Risultato dell'applicazione del JOIN ESTERNO a IMPIEGATO e DIPARTIMENTO

(a) **IMPIEGATO_1**

NOME_I	SSN	DATA_N	INDIRIZZO
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX
Wallace, Jennifer S.	987654321	1941-08-20	291 Berry, Bellaire, TX
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX
Benitez, Carlos M.	888664444	1963-01-09	7654 Beech, Houston, TX

(b) **IMPIEGATO_2**

SSN	NUM_D
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1
999775555	null
888664444	null

(c) **IMPIEGATO_3**

SSN	NUM_D
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1

Figura 1.3 - Il problema della "tupla dangling". (a) La relazione IMPIEGATO_1 (comprende tutti gli attributi di IMPIEGATO a eccezione di NUMERO_D). (b) La relazione IMPIEGATO_2 (comprende l'attributo NUMERO_D con valori nulli). (c) La relazione IMPIEGATO_3 (comprende l'attributo NUMERO_D ma non comprende tuple per le quali NUMERO_D assume valori nulli).

Analisi degli algoritmi di normalizzazione

Uno dei problemi con gli algoritmi di normalizzazione descritti è che il progettista della base di dati deve prima di tutto specificare *tutte* le dipendenze funzionali rilevanti fra gli attributi della base di dati. Questo non è un compito agevole per una grande base di dati con centinaia di attributi. Una dimenticanza nello specificare una o due dipendenze importanti può avere come risultato un progetto inadeguato. Un altro problema è che questi algoritmi non sono in generale deterministici. Ad esempio, gli *algoritmi di sintesi* (Algoritmi 1.1 e 1.4) richiedono la specificazione di una copertura minimale G per l'insieme di dipendenze funzionali Y . Dato che in generale ci possono essere molte coperture minimali corrispondenti a F , l'algoritmo può fornire diversi progetti a seconda della specifica copertura minimale usata. Alcuni di questi progetti possono non essere desiderabili. L'*algoritmo di decomposizione* (Algoritmo 1.3) dipende dall'ordine con cui le dipendenze funzionali sono fornite all'algoritmo; anche qui è possibile che si derivino molti progetti diversi relativi allo stesso insieme di dipendenze funzionali, a seconda dell'ordine con cui queste dipendenze sono considerate per la violazione della BCNF. E ancora, alcuni progetti possono essere significativamente superiori mentre altri possono essere inadeguati.

Algoritmi di normalizzazione – Sommario

Algoritmo	Input	Output	Proprietà/Scopo	Note
Algoritmo 1.1	Insieme F di dipendenze funzionali	Un insieme di relazioni in 3NF	Conservazione delle dipendenze	Nessuna garanzia sulla verifica della proprietà di join senza perdita
Algoritmo 1.2	Una decomposizione D di R e un insieme F di dipendenze funzionali	Risultato booleano, sì o no per la proprietà di join non-additivo	Test per la decomposizione non-additiva	Esiste un test più semplice
Algoritmo 1.3	Insieme F di dipendenze funzionali	Un insieme di relazioni in BCNF	Decomposizione non-additiva	Nessuna garanzia della conservazione delle dipendenze
Algoritmo 1.4	Insieme F di dipendenze funzionali	Un insieme di relazione in 3NF	Decomposizione non-additiva e con conservazione delle dipendenze	Può capitare di non avere una BCNF, ma si ottiene la 3NF e tutte le proprietà auspicabili
Algoritmo 1.4a	Schema di relazione R con un insieme F di dipendenze funzionali	Chiave K di R	Per trovare una chiave K (cioè un sottoinsieme di R)	L'intera relazione R è sempre una superchiave di default

Dipendenze multivalue e quarta forma normale

Finora abbiamo analizzato solo il concetto di dipendenza funzionale, che rappresenta il tipo più importante di dipendenza nella teoria della progettazione di basi di dati relazionali. In molti casi, però, le relazioni hanno vincoli che non possono essere specificati sotto forma di dipendenze funzionali. In questo paragrafo si esaminerà il concetto di dipendenza *multivalue* (MVD, *multivalued dependency*) e si definirà la *quarta forma normale* che si basa su questa dipendenza. Le dipendenze multivalue sono una conseguenza della prima forma normale (1NF), che impedisce a un attributo in una tupla di conoscere un *insieme di valori*. Se vi sono due o più attributi multivalue *indipendenti* nello stesso schema di relazione, si ha il problema di dover ripetere ogni valore di uno degli attributi per ciascun valore dell'altro attributo per mantenere consistente lo stato della relazione e per conservare l'indipendenza tra gli attributi coinvolti. Questo vincolo viene espresso mediante una dipendenza multivalue.

Si consideri ad esempio la relazione IMP mostrata in Figura 1.4. Una tupla in questa relazione IMP rappresenta il fatto che un impiegato il cui nome è NOME_I lavora al progetto denominato NOME_P e ha un dipendente il cui nome è NOME_D. Un impiegato può lavorare a tanti progetti, può avere molti dipendenti, e i progetti e i dipendenti dell'impiegato sono indipendenti l'uno dall'altro. Per mantenere consistente lo stato della relazione si dovrà avere una tupla distinta per ogni possibile combinazione di dipendente e progetto di ciascun impiegato. Questo vincolo viene specificato tramite una dipendenza multivalue sulla relazione IMP. In modo informale può verificarsi una MVD ogni volta che due associazioni *indipendenti* A:B e A:C di tipo 1:N sono fuse nella stessa relazione.

(a) IMP			(b) PROGETTI_IMP	
NOME_I	NOME_P	NOME_D	NOME_I	NOME_P
Smith	X	John	Smith	X
Smith	Y	Anna	Smith	Y
Smith	X	Anna	Brown	W
Smith	Y	John	Brown	X
Brown	W	Jim	Brown	Y
Brown	X	Jim	Brown	Z
Brown	Y	Jim		
Brown	Z	Jim		
Brown	W	Jean		
Brown	X	Jean		
Brown	Y	Jean		
Brown	Z	Jean		
Brown	W	Bob		
Brown	X	Bob		
Brown	Y	Bob		
Brown	Z	Bob		

DIPENDENTI_IMP	
NOME_I	NOME_D
Smith	Anna
Smith	John
Brown	Jim
Brown	Jean
Brown	Bob

Figura 1.4 – Decomposizione di uno stato della relazione IMP che non è in 4NF. (a) La relazione IMP con ulteriori tuple. (b) Le due relazioni in 4NF corrispondenti: PROGETTI_IMP e DIPENDENTI_IMP

Definizione formale di dipendenza multivalore

Una dipendenza multivalore $X \twoheadrightarrow Y$ specificata sullo schema di relazione R , dove X e Y sono sottoinsiemi di R , specifica il seguente vincolo su qualsiasi stato di relazione r di R ; se in r esistono due tuple t_1 e t_2 tali che $t_1[X] = t_2[Y]$, allora in r devono esistere anche due tuple, t_3 e t_4 con le seguenti proprietà, dove viene usato Z per indicare $(R - (X \cup Y))$:

- $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
- $t_3[Y] = t_1[Y] = t_4[Y] = t_2[Y]$.
- $t_3[Z] = t_2[Z] = t_4[Z] = t_1[Z]$.

Ogni volta che sussiste $X \twoheadrightarrow Y$, diciamo che X **multidetermina** Y . A causa della simmetria nella definizione, quando $X \twoheadrightarrow Y$ sussiste in R , lo stesso fa $X \twoheadrightarrow Z$; quindi $X \twoheadrightarrow Y$ implica $X \twoheadrightarrow Z$ e perciò talvolta viene scritto come $X \twoheadrightarrow Y|Z$.

La definizione formale specifica che, dato un particolare valore di X , l'insieme di valori di Y determinato da questo valore di X è determinato completamente solo da X e *non dipende* dai valori dei restanti attributi in Z di R . Ogni volta che esistono due tuple che hanno valori distinti di Y ma lo stesso valore di X , questi valori di Y devono essere ripetuti in tuple separate con *ogni valore distinto di Z* che si presenta con quello stesso valore di X . Questo corrisponde informalmente al fatto che Y è un attributo multivalore delle entità rappresentate dalle tuple in R .

(a) IMP		
NOME_I	NOME_P	NOME_D
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

(b) PROGETTI_IMP		DIPENDENTI_IMP	
NOME_I	NOME_P	NOME_I	NOME_D
Smith	X	Smith	John
Smith	Y	Smith	Anna

Nella relazione valgono: $\text{NOME_I} \rightarrow \text{NOME_P}$ e $\text{NOME_I} \twoheadrightarrow \text{NOME_D}$ (oppure $\text{NOME_I} \twoheadrightarrow \text{NOME_P} | \text{NOME_D}$).

Una MVD $X \twoheadrightarrow Y$ in R è detta **MVD banale** (trivial) se:

- Y è un sottoinsieme di X , oppure
- $X \cup Y = R$

In PROGETTI_IMP, $\text{NOME_I} \twoheadrightarrow \text{NOME_P}$ è una MVD banale.

Figura 1.5 – Quarta forma normale. (a) La relazione IMP con due MVD: $\text{NOME_I} \rightarrow \text{NOME_P}$ e $\text{NOME_I} \twoheadrightarrow \text{NOME_D}$. (b) Decomposizione della relazione IMP in due relazioni PROGETTI_IMP e DIPENDENTI_IMP in

Regole di inferenza per FD e MVD

Come per le dipendenze funzionali, sono state sviluppate regole di inferenza per le dipendenze multivalore; tuttavia è meglio fornire un contesto unificato che includa le DF e le MVD così che ambedue i tipi di vincolo possano essere considerati insieme. Le regole di inferenza seguenti, da RI1 a RI8, formano un insieme completo e corretto per inferire le dipendenze funzionali e multivalore da un insieme di dipendenze dato. Si supponga che tutti gli attributi siano inclusi in uno schema di relazione “universale” $R = \{A_1, A_2, \dots, A_n\}$ e che X, Y, Z e W siano sottoinsiemi di R .

RI1 (regola riflessiva per le DF): se $X \supseteq Y$, allora $X \rightarrow Y$.

RI2 (regola di arricchimento per le DF): $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.

RI3 (regola transitiva per le DF): $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.

RI4 (regola di complemento per le MVD): $\{X \twoheadrightarrow Y\} \models X \twoheadrightarrow (R - (X \cup Y))$.

RI5 (regola di arricchimento per le MVD): Se $X \twoheadrightarrow Y$ e $W \supseteq Z$, allora $WX \twoheadrightarrow YZ$.

RI6 (regola transitiva per le MVD): $\{X \twoheadrightarrow Y, Y \twoheadrightarrow Z\} \models X \twoheadrightarrow (Z - Y)$.

RI7 (regola di replicazione per DF in MVD): $\{X \rightarrow Y\} \models X \twoheadrightarrow Y$.

RI8 (regola di unione per le DF e le MVD): se $X \twoheadrightarrow Y$ ed esiste W con le proprietà che (a) $W \cap Y$ è vuota, (b) $W \rightarrow Z$ e (c) $Y \supseteq Z$, allora $X \rightarrow Z$.

Le regole di inferenza di Armstrong per le DF sono quelle da RI1 a RI3, mentre le regole da RI4 a RI6 sono regole di inferenza che riguardano solo le MVD. Le RI7 e RI8 sono correlate sia alle DF sia alle MVD. In particolare, la RI7 afferma che una dipendenza funzionale è un *caso speciale* di una dipendenza multivalore; cioè ogni DF è anche una MVD perché soddisfa la definizione formale di una MVD. Questa equivalenza, però, sottende che: una DF $X \rightarrow Y$ è una MVD $X \twoheadrightarrow Y$ con l'*ulteriore limitazione implicita* che al massimo un valore di Y è associato a ogni valore di X . Dato un insieme F di dipendenze funzionali e multivalore specificato su $R = \{A_1, A_2, \dots, A_n\}$, si possono usare le regole da RI1 a RI8 per inferire l'insieme (completo) di tutte le dipendenze (funzionali o multivalore) F^+ che sussisterà in ogni stato di relazione r di R che soddisfa F . Viene chiamato F^+ la **chiusura** di F .

Quarta forma normale

Ora viene presentata la definizione di quarta forma normale 4NF (*fourth normal form*), che viene violata quando una relazione ha dipendenze multivalore non volute, e quindi può essere usata per identificare e decomporre tali relazioni.

Definizione: Uno schema di relazione R è in 4NF rispetto a un insieme di dipendenze F (che include le dipendenze funzionali e multivalore) se, per ogni dipendenza multivalore *non banale* $X \twoheadrightarrow Y$ in F^+ , X è una superchiave di R .

La relazione IMP della Figura 1.4(a) non è in 4NF perché nelle MVD non banali $NOME_I \twoheadrightarrow NOME_P$ e $NOME_I \twoheadrightarrow NOME_D$, $NOME_I$ non è una superchiave di IMP. Si decompone allora IMP in PROGETTI_IMP e DIPENDENTI_IMP, come mostrato nella Figura 1.4(b). Sia PROGETTI_IMP sia DIPENDENTI_IMP sono in 4NF, perché le MVD: $NOME_I \twoheadrightarrow NOME_P$ in PROGETTI_IMP e $NOME_I \twoheadrightarrow NOME_D$ in DIPENDENTI_IMP sono MVD banali. In PROGETTI_IMP o DIPENDENTI_IMP non è contenuta alcun'altra MVD non banale. In questi schemi di relazione non sussiste neppure alcuna DF.

NOTA sulle MVD non-banali: Relazioni contenenti MVD non-banali tendono ad essere relazioni “*tutta chiave*”, nel senso che la chiave è formata da tutti gli attributi.

Decomposizione di join non-additivo in schemi in 4NF

Ogni volta che si decomponete uno schema di relazione R in $R_1 = (X \cup Y)$ e $R_2 = (R - Y)$ sulla base di una MVD $X \rightarrow\!\!\!> Y$ che sussiste in R , la decomposizione gode della proprietà di join non-additivo. Si può dimostrare che questa è una *condizione necessaria e sufficiente* per decomporre uno schema in due schemi che hanno la proprietà di join non-additivo, come specificato dalla proprietà LJ1' che è un'ulteriore generalizzazione della proprietà LJ1 mostrata in precedenza. La proprietà LJ1 è relativa solo alle DF, mentre LJ1' gestisce sia le DF sia le MVD (da ricordare che una DF è anche una MVD).

PROPRIETÀ LJ1'.

Gli schemi di relazione R_1 e R_2 costituiscono una decomposizione non-additiva di R rispetto a un insieme F di dipendenze funzionali e multivalore se e solo se:

$$(R_1 \cap R_2) \rightarrow\!\!\!> (R_1 - R_2)$$

oppure, per simmetria, se e solo se

$$(R_1 \cap R_2) \rightarrow\!\!\!> (R_2 - R_1).$$

Con una piccola modifica dell'Algoritmo 1.3 si ottiene l'Algoritmo 1.5, che crea una decomposizione non-additiva che produce schemi di relazione che sono in 4NF (anziché in BCNF). Come l'Algoritmo 1.3, l'Algoritmo 1.5 *non* produce necessariamente una decomposizione che conserva le DF.

Input: Una relazione universale R e un insieme di dipendenze funzionali e multivalore F .

1. Si ponga $D := \{R\}$;
2. Fintantoché vi è uno schema di relazione Q in D che non è in 4NF;
 - { si scelga uno schema di relazione Q in D che non è in 4NF;
 - si trovi una MVD non banale $X \rightarrow\!\!\!> Y$ in Q che viola la 4NF;
 - si sostituisca Q in D con due schemi di relazione $(Q - Y)$ e $(X \cup Y)$;

Algoritmo 1.5 – Decomposizione relazionale in relazioni in 4NF con la proprietà di join non-additivo.

Applicazioni di design e Tuning di database

Progettazione di un database

In precedenza, sono stati analizzati in dettaglio gli aspetti **teorici** della progettazione di un database. L'attività di design di un database nel suo complesso è un processo sistematico che segue una metodologia ben definita: la metodologia di design, ed è spesso legata al tool di design fornito dalla casa produttrice (Oracle, Sybase, ecc...). Non verrà analizzata una specifica metodologia, ma piuttosto si vedrà come viene svolta la fase di progettazione di un database in una grande azienda.

In generale, i **Large Database** sono quei database con diverse decine di gigabyte di dati e 30-40 o più distinti tipi di entità (grandi database). Tra questi figurano i sistemi di gestione delle transazioni, attivi 24 ore al giorno con grandi volumi di transazioni e centinaia o migliaia di utenti.

Il ruolo dei Sistemi Informativi in un'azienda

I database sono una parte fondamentale di ogni sistema informativo aziendale; il riconoscimento dell'importanza strategica di un'accurata gestione dei dati ha portato alla creazione di nuove figure professionali:

- Un DBA (Database Administrator) o un dipartimento per l'amministrazione di database,
- Un management per la gestione delle risorse informative.

L'importanza della gestione dei dati

Un'accurata gestione dei dati è fondamentale in quanto i dati sono una risorsa dell'azienda e una loro corretta gestione facilita e rende più efficiente il lavoro. In un'azienda poi, sempre più funzionalità sono informatizzate, aumentando la richiesta di memorizzazione di grandi quantità di informazioni che siano reperibili al loro valore attuale. All'aumentare dei dati e delle applicazioni, aumentano anche le relazioni tra i dati da modellare e memorizzare.

L'importanza dei database

L'utilizzo di sistemi di basi di dati soddisfa pienamente i punti precedentemente elencati. Un sistema di base di dati ha anche altre due caratteristiche preziose in ambiti aziendali:

- L'indipendenza dai dati protegge i programmi applicativi da cambiamenti sia della logica aziendale, sia della memorizzazione fisica.
- Gli schemi esterni (o *viste*) permettono l'uso agli stessi dati da parte di diverse applicazioni.

Le **caratteristiche chiave**, quindi si individuano nell'integrazione dei dati tra diverse applicazioni in un singolo DB, facilità di sviluppo di nuove applicazioni usando linguaggi ad alto livello tipo SQL e la possibilità da parte dei manager di interrogare i dati ed avere risultati aggiornati.

L'evoluzione e la diffusione

Dai primi anni '70 fino alla metà degli anni '80 c'era la tendenza di creare grosse repository, gestite da un singolo DBMS centralizzato, ma negli ultimi 15 anni la situazione si è invertita per i seguenti motivi:

1. La diffusione di **Personal Database** (Access, Excel, FoxPro, SQL, Anywhere) che permette a molte categorie di utenti di definire dei database personali. È possibile scaricare porzioni di DB dal server, lavorarci sopra e ri-memorizzare il tutto nuovamente sul server.
2. L'avvento di **DBMS distribuiti** e client/server permette di allocare i dati su più computer per un migliore controllo ed una elaborazione locale più veloce. Gli utenti locali possono accedere ai dati remoti come client del DBMS o tramite web.

Strutturazione dei database system aziendali

Molte organizzazioni utilizzano dizionari dei dati, cioè dei mini-DBMS per gestire i metadati, quali:

- Strutture di database,
 - Descrizione degli schemi,
 - Descrizione del design fisico (strutture di accesso, file, taglia dei record, ecc...)
- Informazioni sugli utenti (responsabilità, diritti di accesso)
- Descrizioni di alto livello di transazioni ed applicativi e relazioni utente-transazioni.
- Relazioni tra transazioni e dati
- Statistiche sull'utilizzo di porzioni di database

Sistemi di gestione delle transazioni

Sono sistemi **business critical**, attivi 24 ore su 24, che servono centinaia di transazioni al minuto da terminali remoti e locali. Il tempo di risposta medio e massimo ed il numero medio di transazioni per minuto sono dei fattori critici. Per questo tipo di sistemi la **progettazione fisica** del database è un elemento di vitale importanza.

Il ciclo di vita di un sistema informativo

In grosse aziende, un database system è solo una parte di un **sistema informativo**. Esso è composto da dati, DBMS, hardware, media di memorizzazione, applicativi che interagiscono con i dati, il personale che gestisce o usa il sistema, gli applicativi che gestiscono l'aggiornamento dei dati, i programmatore che sviluppano tali applicativi.

La progettazione di una base di dati costituisce solo una delle componenti del processo di sviluppo di un sistema informativo complesso e va quindi inquadrata in un contesto più ampio, quello del *ciclo di vita* dei sistemi informativi. Tale ciclo di vita di un sistema informativo (risorse per raccolta, gestione uso e disseminazione) è detto **macro-ciclo di vita**. Quindi, il ciclo di vita di un sistema di base di dati è detto **macro-ciclo di vita**; con l'aumentare della complessità e delle funzioni svolte dai DBMS, questa suddivisione diventa sempre più sfumata.

Le fasi del macro-ciclo di vita

1. **Analisi di fattibilità**: dove si analizzano le potenziali aree di applicazione, si effettuano degli studi di *costi/benefici*, si determina la complessità di dati e processi, e si impostano le priorità tra le applicazioni.
2. **Raccolta ed analisi dei requisiti**: consiste nell'individuazione e nello studio delle proprietà e delle funzionalità che il sistema informativo dovrà avere. Questa fase richiede un'interazione con gli utenti del sistema e produce una descrizione completa, ma generalmente informale, dei dati coinvolti e delle operazioni su di essi.
3. **Progettazione**: si divide generalmente in *progettazione dei dati* e *progettazione delle applicazioni*. Nella prima si individua la struttura e l'organizzazione che i dati dovranno avere, nell'altra si definiscono le caratteristiche dei programmi applicativi.
4. **Implementazione**: consiste nella realizzazione del sistema informativo secondo la struttura e le caratteristiche definite nella fase di progettazione. Viene costruito e popolato il database e viene prodotto il codice dei programmi, testando le transazioni.
5. **Validazione e Testing**: serve a certificare il corretto funzionamento e la qualità del sistema informativo. La sperimentazione deve prendere, per quanto possibile, tutte le condizioni operative.

6. **Rilascio e Manutenzione:** in questa fase il sistema informativo diventa operativo ed esegue i compiti per i quali era stato originariamente progettato. Il rilascio può essere preceduto da una fase di addestramento del personale al nuovo sistema. Se emergono nuove funzionalità da implementare, si ripetono i passi precedenti, per includerle nel sistema. Se non si verificano malfunzionamenti o revisioni delle funzionalità del sistema, questa attività richiede solo operazioni di gestione e manutenzione.

Metodologia di progettazione

Nell'ambito delle basi di dati, si è consolidata negli anni una metodologia di progetto che ha dato prova di soddisfare pienamente alcune proprietà che una buona base di dati dovrebbe possedere. Tale metodologia è articolata in tre fasi principali da effettuare in cascata, separando in maniera netta le decisioni relativa a "cosa" rappresentare in una base di dati (prima fase), da quelle relative a "come" farlo (seconda e terza fase).

- **Progettazione concettuale.** Il suo scopo è quello di rappresentare le specifiche informali della realtà d'interesse in termini di una descrizione formale e completa, ma indipendente dai criteri di rappresentazione utilizzati nei sistemi di gestione di basi di dati. Il prodotto di questa fase viene chiamato *schema concettuale* e fa riferimento a un *modello concettuale* dei dati. I modelli concettuali consentono di descrivere l'organizzazione dei dati a un alto livello di astrazione, senza tenere conto degli aspetti implementativi. In questa fase infatti, il progettista deve cercare di rappresentare il *contenuto informativo* della base di dati, senza preoccuparsi né della modalità con le quali queste informazioni verranno codificate in un sistema reale, né dell'efficienza dei programmi che faranno uso di queste informazioni.
- **Progettazione logica.** Consiste nella traduzione dello schema concettuale definito nella fase precedente, in termini del modello di rappresentazione dei dati adottato dal sistema di gestione di base di dati a disposizione. Il prodotto in questa fase viene denominato *schema logico* della base di dati e fa riferimento a un *modello logico* dei dati. Come noto, un modello logico permette di descrivere i dati secondo una rappresentazione ancora indipendente da dettagli fisici, ma concreta perché disponibile nei sistemi di gestione di base di dati. In questa fase, le scelte progettuali si basano, tra l'altro, su criteri di ottimizzazione delle operazioni da effettuare sui dati. Si fa comunemente uso anche di tecniche forma di verifica della qualità dello schema logico ottenuto e nel caso del modello relazionale dei dati, la tecnica comunemente utilizzata è la normalizzazione.
- **Progettazione fisica.** In questa fase lo schema logico viene completato con la specifica dei parametri fisici di memorizzazione dei dati (organizzazione dei file e degli indici). Il prodotto di questa fase viene denominato *schema fisico* e fa riferimento a un *modello fisico* dei dati. Tale modello dipende dallo specifico sistema di gestione di basi di dati scelto e si basa sui criteri di organizzazione fisica dei dati in quel sistema.

Le fasi del micro-ciclo di vita

Le attività del ciclo di vita di un database system includono le seguenti 8 fasi:

1. **Definizione del sistema:** dove viene definito l'ambito del sistema di base di dati, i suoi utenti e se le funzionalità. Inoltre, si identificano le interfacce per le categorie di utenti, i vincoli sui tempi di risposta ed i requisiti hardware.
2. **Progettazione della base di dati:** si realizza la progettazione logica e fisica per il DBMS scelto.
3. **Implementazione della base di dati:** si specificano le definizioni concettuali, esterne ed interne, si creano i file del database vuoti e si implementa dell'eventuale software applicativo di supporto.

4. **Caricamento / conversione dei dati:** si popola il database, o inserendo direttamente i dati o convertendo file esistenti nel nuovo formato.
5. **Conversione delle applicazioni:** si convertono le vecchie applicazioni software al nuovo sistema.
6. **Test e validazione:** si effettuano test e validazione del nuovo sistema.
7. **Operation:** il sistema di base di dati e le sue applicazioni diventano operativi. In genere, per un certo tempo vengono utilizzati in parallelo il vecchio ed il nuovo sistema.
8. **Controllo e manutenzione:** il sistema è sottoposto a costante monitoraggio. Eventualmente si possono gestire aggiunte nei dati o nelle funzionalità presenti.

Il processo di progettazione di un database

La fase più interessante nel micro-ciclo di vita è quella di progettazione. Il problema della progettazione può essere riformulato come: *progettare la struttura logica e fisica di uno o più database, per soddisfare i requisiti degli utenti di un'organizzazione su un determinato insieme di operazioni*. Gli scopi della fase di progettazione sono:

1. Soddisfare i requisiti sui dati che interessano gli utenti e a cui accedono le applicazioni.
2. Fornire una strutturazione delle informazioni naturale e di facile comprensione.
3. Soddisfare i requisiti di elaborazione e di prestazioni (tempo di risposta, spazio di memorizzazione ecc....).

È difficile raggiungere tutti gli scopi, in quanto alcuni sono in contrasto tra loro ed un modello più comprensibile può comportare un costo in termini di prestazioni.

Il processo di progettazione è costituito da due attività parallele:

- Progettazione di strutture e contenuti dei dati (*progettisti di basi di dati*).
- Progettazione delle applicazioni che usano la base di dati (*ingegneri del software*).

Le metodologie di progettazione di database si sono focalizzate sulla prima attività (**approccio data-driven** vs **progettazione process-driven**). È ormai riconosciuto, però, che progettisti di database e ingegneri del software debbano collaborare quanto più possibile, servendosi di tools di design per combinarle.

Fasi della progettazione di un database

La progettazione di un database può essere vista come composta da sei fasi principali:

1. Raccolta ed analisi dei requisiti.
2. Progettazione dello schema concettuale.
3. Scelta del DBMS.
4. Mapping del data model (*design logico*).
5. Progettazione dello schema fisico.
6. Implementazione e tuning del database system.

Queste sei fasi non sono eseguite in sequenza, in quanto spesso delle modifiche ad un livello devono essere propagate al livello superiore, creando dei **cicli di feedback**.

Fase 1: Raccolta ed analisi dei requisiti

Per progettare un database è necessario conoscere ed analizzare le aspettative degli utenti nel modo più dettagliato possibile; questo processo è chiamato, appunto, **raccolta ed analisi dei requisiti**. Per specificare i requisiti, è necessario individuare tutte le componenti che interagiranno col database. Tipicamente queste sono: gli utenti e le applicazioni. La fase 1 comprende di quattro attività:

1. Identificare le principali aree di applicazione, gli utenti che useranno il database e quelli il cui lavoro sarà influenzato dal database stesso. Individuare in ogni gruppo di persone un rappresentante per poter portare avanti la raccolta delle specifiche.
2. Analizzare la documentazione già esistente riguardante le applicazioni. Esaminare anche altri tipi di documentazione (form, report, grafici aziendali, ecc...) che in qualche modo possono influenzare i requisiti.
3. Esaminare il contesto operativo e l'utilizzo pianificato delle informazioni. Questa attività include l'analisi delle transazioni, dei flussi di informazioni e la specifica dei dati di input e output per ogni transazione.
4. Intervistare gli utenti finali per determinare priorità ed importanza previste per le varie applicazioni.

Successivamente i requisiti andranno sviluppati in quanto, spesso all'inizio i requisiti sono informali, incompleti, inconsistenti e parzialmente incorretti. È quindi necessario molto lavoro per trasformarli in specifiche da fornire a programmatore e tester. Poiché i requisiti si riferiscono ad un sistema non ancora esistente, inevitabilmente vengono spesso modificati. Per trasformare i requisiti in una forma strutturata, si utilizzano delle **tecniche di specifica dei requisiti**. Le principali tecniche comprendono:

- Analisi orientata agli oggetti (OOA).
- Diagramma di flusso dei dati (DFD).
- Raffinamento degli obiettivi dell'applicazione.

Esistono anche altre tecniche che producono una specifica formale dei requisiti che consentono verifiche matematiche di consistenza (*analisi simboli "what-if"*); sebbene più difficili da usare, queste tecniche sono fondamentali per applicazioni *mission-critical*.

È possibile utilizzare dei **tool CASE (Computer Aided Software Engineering)** per controllare la completezza e la consistenza delle specifiche, detti Upper Case tool. Altri tool permettono di evidenziare i collegamenti tra requisiti ed altre entità di progetto, quali moduli di codice, casi di test, ecc.... (basi di dati di tracciabilità).

La fase di raccolta ed analisi dei requisiti richiede un grande sforzo in termini di tempo, ma è cruciale per il successo del sistema informativo. Un errore dovuto a requisiti errati può essere estremamente costoso poiché può comportare la re-implementazione di buona parte del lavoro; il sistema potrebbe non rispondere alle richieste del cliente e non essere usato affatto.

Fase 2: Design del Database Concettuale

La seconda fase del progetto di un database consta di due attività parallele:

1. **Progettazione dello schema concettuale.** Si esaminano i requisiti per produrre lo schema concettuale del database.
2. **Progettazione di transazioni ed applicazioni.** Si esaminano le applicazioni del database per produrre specifiche di alto livello delle applicazioni.

Progettazione dello schema concettuale

Lo schema concettuale deve essere indipendente dal DBMS per i seguenti motivi:

- ✓ Lo scopo dello schema concettuale è fornire una comprensione completa di struttura, semantica, relazioni e vincoli del database. Legarsi ad un DBMS porterebbe a delle restrizioni che influenzerebbero lo schema.
- ✓ Lo schema concettuale è una descrizione stabile dei contenuti del database. La scelta del DBMS e le decisioni di progettazione successive possono cambiare senza che questo debba essere modificato.
- ✓ L'utilizzo di una data model di alto livello è più espressivo e generale dei modelli di dati utilizzati dai singoli DBMS, ed è quindi più comprensibile per gli utenti.
- ✓ La descrizione diagrammatica dello schema concettuale può essere usata molto efficacemente come mezzo di comunicazione tra gli utenti del database, i progettisti e gli analisti. I modelli di dati dei DBMS, essendo di livello più basso, spesso mancano di un tale livello di espressività.

Caratteristiche di un modello concettuale dei dati di alto livello

Un data model di alto livello deve godere delle seguenti proprietà:

1. **Espressività.** Il data model deve permettere una facile distinzione tra tipi di dati, relazioni e vincoli.
2. **Semplicità e comprensibilità.** Il modello deve essere semplice, per consentire a utenti non esperti di comprendere ed utilizzare i suoi concetti.
3. **Minimalità.** Il modello dovrebbe avere pochi concetti di base, non sovrapponibili.
4. **Rappresentazione diagrammatica.** Il modello dovrebbe avere una notazione diagrammatica per rappresentare schemi concettuali di facile comprensione.
5. **Formalità.** Il modello deve fornire dei formalismi per specificare in modo non ambiguo i dati.

Approcci alla progettazione di uno schema concettuale

Per progettare uno schema concettuale, è necessario individuare le componenti di base di uno schema. Queste sono:

- Le entità.
- Le relazioni.
- Gli attributi.
- I vincoli di cardinalità e partecipazione.
- Le chiavi.
- Le gerarchie di specializzazione/generalizzazione.
- Le entità deboli.

Esistono due approcci alla progettazione di uno schema concettuale:

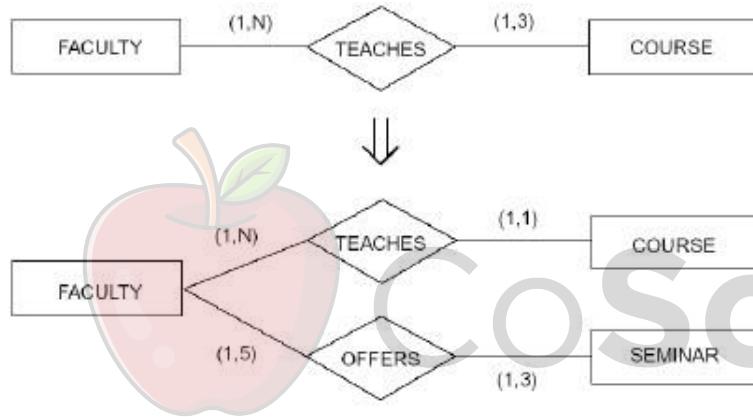
1. **Progetto di schema centralizzato (o one-shot).** Tutti i requisiti di applicazioni differenti e diversi gruppi di utenti sono fusi in un singolo insieme di requisiti prima di iniziare la progettazione. Ai Database Administrator spetta il compito di decidere come unire i requisiti e di progettare l'intero schema. Una volta progettato l'intero schema concettuale si progettano gli schemi esterni.
2. **Integrazione di viste.** I requisiti non vengono fusi: Inizialmente si progetta uno schema (o vista) per ogni gruppo di utenti. Successivamente, in una fase di integrazione, le viste sono fuse in un unico schema concettuale globale.

Strategie per la progettazione di schemi

Esistono differenti strategie per creare uno schema concettuale partendo dai requisiti:

- **Strategia Top-Down.** Si parte da uno schema con astrazioni di alto livello, che vengono successivamente raffinate. Vengono specificate le entità ad alto livello; e quando si vanno a specificare gli attributi, le entità si spezzano in entità di livello inferiore e si introducono relazioni (ad esempio, specializzazione di un tipo di entità in sottoclassi).
- **Strategia Bottom-Up.** Si parte da uno schema contenente astrazioni di base, che vengono via via combinate e messe in relazione tra loro (ad esempio, generalizzare tipi di entità in superclassi di alto livello).
- **Strategia Inside-Out.** È una specializzazione del bottom-up, in cui l'attenzione è focalizzata su un nucleo centrale di operazioni. La modellazione, quindi, si allarga verso l'esterno, inglobando nuovi concetti collegati a quelli già considerati.
- **Strategia Mixed.** Non prevede una strategia uniforme per tutto il progetto, ma dà la disponibilità di usare metodi diversi per le varie componenti, che vengono combinate successivamente.

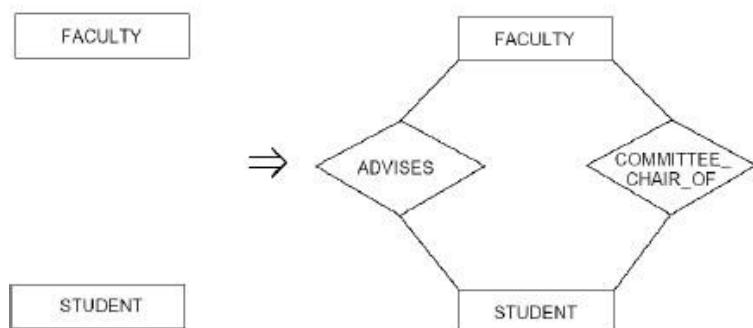
Strategia Top-Down: Esempio



Un raffinamento Top-Down:

L'entità Course è raffinata in Course e Seminar e la relazione TEACHES è spezzata in TEACHES e OFFERS.

Strategia Bottom-Up: Esempio



Un raffinamento Bottom-Up:

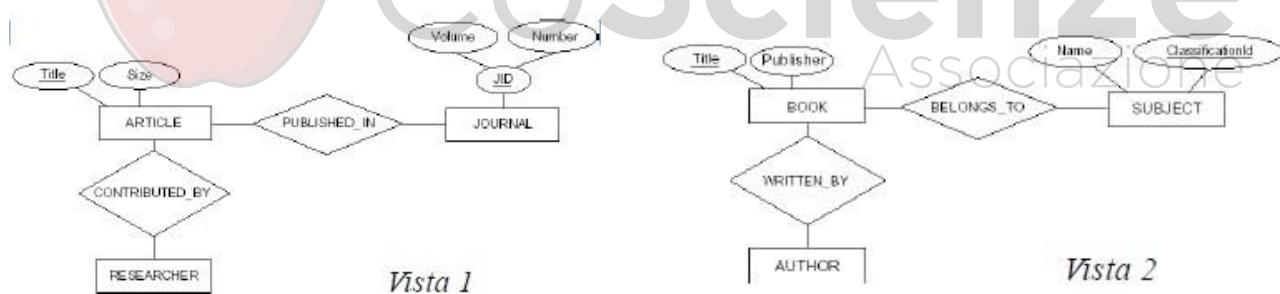
Dopo la definizione delle entità, si ha un raffinamento, definendo le relazioni esistenti.

Integrazione di schemi

Si analizza una **metodologia per l'integrazione di schemi in uno schema globale di database** (necessaria per grossi database). L'integrazione di schemi può essere divisa in quattro sottocompiti:

1. **Identificazione di corrispondenze e conflitti tra gli schemi.** In questa fase è possibile individuare quattro possibili tipi di conflitti:
 - a. **Conflitti di nome.** Che possono essere di due tipi:
 - i. **Sinonimi:** due schemi utilizzano termini diversi per identificare lo stesso concetto (ad esempio, in due schemi differenti esistono il tipo di entità CUSTOMER e CLIENT che descrivono lo stesso concetto)
 - ii. **Omonimi:** due schemi utilizzano lo stesso termine per identificare concetti diversi.
 - b. **Conflitti di tipo.** Lo stesso concetto può essere espresso in schemi diversi con costrutti di modellazione diversi (ad esempio, un attributo in uno schema e un tipo di entità in un altro schema).
 - c. **Conflitti di dominio.** Un attributo può avere domini differenti in schemi diversi (ad esempio, intero in uno schema e carattere in un altro). Conflitti di unità di misura (un attributo è descritto in metri in uno schema e in chilometri in un altro).
 - d. **Conflitti tra vincoli.** Due schemi possono imporre vincoli differenti (ad esempio, la chiave di un tipo di entità può risultare diversa in due schemi differenti).
2. **Modifica delle viste per renderle conformi.**
3. **Fusione delle viste.** Si crea uno schema globale fondendo tutte le viste. I concetti corrispondenti devono essere rappresentati solo una volta. Non è un compito automatizzabile, e richiede una notevole esperienza.
4. **Ristrutturazione.** Lo schema può essere analizzato per eliminare ridondanze.

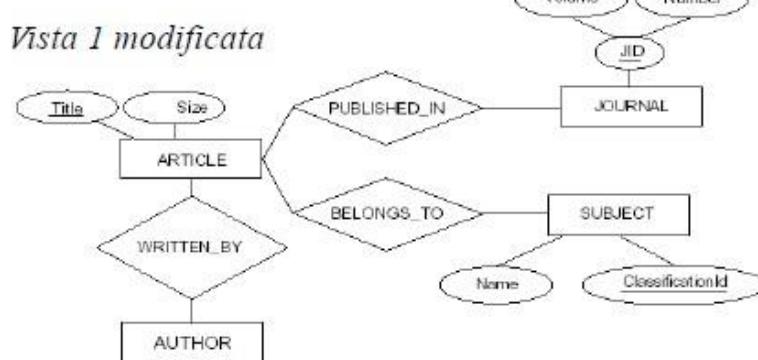
Esempio: Due viste di un database bibliografico



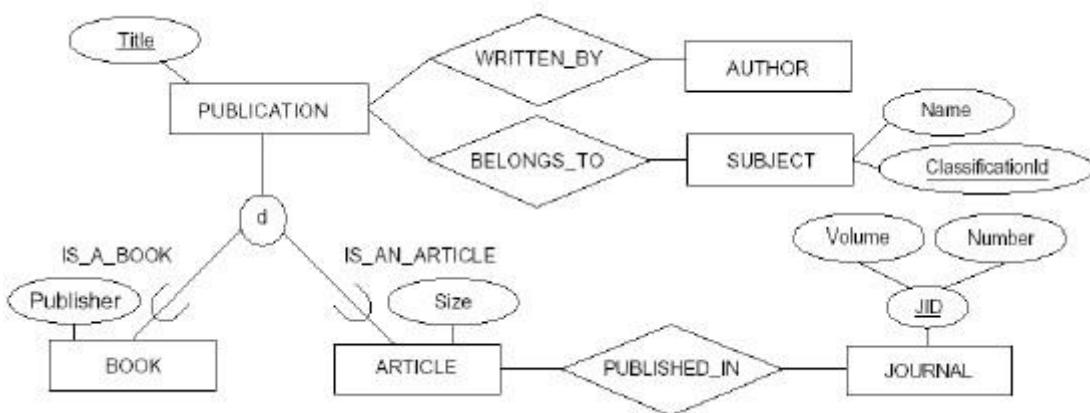
La vista 1 viene modificata per rendere i contenuti conformi alla vista 2:

- Researcher e Author,
- CONTRIBUTED_BY e WRITTEN_BY sono sinonimi.

Vista 1 modificata



Schema finale ottenuto dall'integrazione delle due viste con la generalizzazione di **book** e **article** in **publication**.



Progettazione di transazioni

Lo scopo di questa fase è di progettare le transazioni (o applicazioni) del database in modo indipendente dal DBMS. Specificare le caratteristiche funzionali delle transazioni assicura che lo schema del database includerà le informazioni che esse richiedono. Difficilmente nella fase di progettazione si ha una visione completa di tutte le transazioni da implementare: molte saranno identificate e realizzate al termine dell'implementazione del database. Una tecnica usata per specificare le transazioni prevede l'identificazione di: Input, Output e Comportamento funzionale.

Definendo i parametri di *input/output* ed il flusso di *controllo funzionale* interno, è possibile specificare una transazione in un modo concettuale indipendente dal sistema.

È possibile raggruppare le transazioni in tre categorie:

1. **Transazioni di retrieval.** Usate per recuperare dati da visualizzare o su schermo o su report.
2. **Transazioni di update.** Usate per inserire o modificare i dati.
3. **Transazioni miste.** Usate per applicazioni più complesse che richiedono sia retrieval che update.

Il design di transazioni è considerato parte dell'Ingegneria del software.

Fase 3: Scelta del DBMS

La scelta del DBMS è influenzata da tre fattori:

- Fattori tecnici.
- Fattori economici.
- Fattori organizzativi/aziendali.

Fattori tecnici

Tra i fattori tecnici che possono influenzare la scelta del DBMS troviamo:

- Data model del DBMS (relazionale, O-O, ecc...).
- Strutture di memorizzazione offerte.
- Disponibilità di interfacce per utenti e programmatore.
- Disponibilità di tool di sviluppo.
- Linguaggio di query supportato.
- Possibilità di interagire con altri DBMS.

Fattori economici

Tra i fattori economici che possono influenzare la scelta del DBMS troviamo:

- Costi di acquisto del software (inclusi tool per GUI, recovery/backup, design, linguaggi di programmazione, ecc...).
- Costi di manutenzione (assistenza del rivenditore e costi di aggiornamento).
- Costi di acquisizione nuovo hardware (memoria, terminali, driver di disco, ecc...).
- Costi di creazione e conversione database.
- Costi del personale.
- Costi del training.
- Costi operativi.

Fattori aziendali

Tra i fattori aziendali che possono influenzare la scelta del DBMS troviamo:

- Adozione di una “filosofia” in tutta l’azienda. Accettare un DBMS implica accettarne il data model, il paradigma di programmazione, i tool di sviluppo, ecc....
- Familiarità del personale con il sistema. Scegliere un sistema già conosciuto diminuisce i costi di training.
- Disponibilità di servizi post-vendita. La disponibilità di assistenza post-vendita può risultare molto importante.

Ulteriori fattori da considerare nella scelta del DBMS sono:

- portabilità su più piattaforme,
- disponibilità di tool di backup, recovery, security, ecc...
- integrazione con “soluzioni complete” per il sistema informativo.

I DBMS si stanno evolvendo sempre più, integrando un gran numero di pacchetti software. Tra questi troviamo:

- Browser e editor di testo.
- Generatori di report.
- Software di comunicazione (monitor TP).
- Soluzioni di data entry e data display, quali form, menù, ecc...
- Tool di interrogazione via Web.
- Tool di progettazione visuale.

Fase 4: Data Model Mapping

La creazione di schemi concettuali ed esterni nel data model specifico del DBMS selezionato avviene in due passi:

1. **Mapping indipendente dal sistema.** Il mapping non considera nessuna caratteristica specifica del DBMS (ad esempio, la traduzione da ER a Relazionale)
2. **Mapping per lo specifico DBMS.** Si modifica lo schema ottenuto al passo 1 per adeguarlo alle caratteristiche ed ai vincoli dello specifico DBMS.

Risultato: le istruzioni DDL (Data Definition Language) per specificare gli schemi concettuali ed esterni del database.

Fase 5: Progettazione fisica

Comprende la definizione di strutture di memorizzazione e di access path per i file del database. Esistono tre parametri che guidano la progettazione fisica di un database:

1. **Tempo di risposta.** È il tempo medio trascorso dalla sottomissione di una transazione alla ricezione dei risultati.
2. **Utilizzazione di spazio.** È lo spazio totale utilizzato dal database, compresi gli indici.
3. **Throughput delle transazioni.** È il numero medio di transazioni completate al minuto (*parametro critico per sistemi transazionali quali banche, prenotazioni su linee aeree*).

A causa dell'importanza di tali parametri, spesso nei requisiti si includono i limiti per il caso medio e per il caso pessimo. Le prestazioni dipendono dalla taglia dei record e dal numero di record nei file, essi sono parametri che vanno stimati. Spesso si utilizzano dei prototipi del sistema per valutarne le prestazioni effettive. Devono essere considerati gli attributi usati per accedere ai record e gli indici primari e secondari necessari.

Fase 6: Implementazione e Tuning del database system

L'implementazione del database è a carico del Database Administrator e dei Database Designer. Le istruzioni DDL sono compilete ed eseguite per creare gli schemi ed i file vuoti. Il database viene quindi popolato con i dati; se essi sono già esistenti in un altro formato, può essere necessario implementare delle routine di conversione. Infine, i programmatore implementano le transazioni utilizzando comandi DML (Data Manipulation Language) del DBMS.

Tuning del database

La maggior parte dei DBMS includono tool di monitoraggio. In base ai dati collezionati, è possibile modificare tabelle, access path, query, ecc... alcune query o transazioni possono essere riscritte per migliorare le prestazioni. Il processo di tuning continua durante tutta l'operatività del database system.

La progettazione fisica nei database relazionali

La progettazione fisica di un database si propone non solo di fornire delle strutture dati appropriate, ma anche di garantire delle buone performance del database system. Per effettuare una buona progettazione, è necessario conoscere le query, le transazioni e gli applicativi eseguiti sul database, analizzarne la frequenza di esecuzione, e gli eventuali vincoli posti nei requisiti.

Fattori che influenzano il design fisico

- **Analisi di query.** Per ogni query si deve specificare:
 1. I file a cui accede la query.
 2. Gli attributi su cui sono specificate le condizioni di selezione.
 3. Gli attributi su cui sono specificate le condizioni di join.
 4. Gli attributi di cui la query recupera i valori.

Gli attributi identificati nei punti 2) e 3) sono candidati per la definizione di access path.

- **Analisi di transazioni.** Per ogni transazione di update si specificano:
 1. I file aggiornati dalla transazione.
 2. Il tipo di operazioni per ogni file (*lettura, modifica, cancellazione*).
 3. Gli attributi su cui sono specificate condizioni di cancellazione o di modifica.
 4. Gli attributi i cui valori sono modificati dalla transazione.

Gli attributi identificati nel punto 3) sono candidati per la definizione di access path. Invece, gli attributi identificati nel punto 4) **NON** dovrebbero essere utilizzati in access path.

- **Analisi sulla frequenza di esecuzione di query e transazioni.** Si determina la frequenza con cui le query sono eseguite; in generale, vale la regola “80-20” dove l’80% del processing è effettuato dal 20% delle query.
- **Analisi sui vincoli temporali.** Alcune query e transazioni possono avere dei vincoli temporali che impongono priorità nella definizione di access path (ad esempio, una transazione deve terminare entro 5s dalla sua invocazione). Questo vincolo fornisce una priorità maggiore a degli attributi che dovranno essere usati per access path.
- **Analisi sulla frequenza di operazioni di update.** Il numero di access path su file aggiornati frequentemente deve essere minimizzato, in quanto produce un overhead per aggiornare anche gli access path.
- **Analisi dei vincoli di univocità degli attributi.** Andrebbero specificati degli access path per ogni chiave candidata.

Decisioni progettuali sugli indici

Sebbene le performance di query migliorino fortemente in presenza di indici o schemi hash, le operazioni di inserimento, modifica e cancellazione sono rallentate dagli indici. Le decisioni sulle indicizzazioni ricadono in una delle cinque categorie seguenti:

1. **Quando indicizzare un attributo.** Un attributo deve essere indicizzato se è chiave o se è utilizzato in una condizione di select (*uguaglianza o range di valori*) o join da una query.
2. **Quale attributo indicizzare.** Un indice può essere definito su uno o più attributi. Se più attributi sono coinvolti in varie query, è necessario definire un indice multi-attributo. L’ordine degli attributi nell’indice deve corrispondere a quello nella query.
3. **Quando creare un indice clustered.** Al più un indice per tabella può essere primario o clustering. Le query su range di valori si avvantaggiano di tali indici, mentre le query di ricerca su indici, che non restituiscono dati, non hanno miglioramenti con indici clustering.
4. **Quando usare indici hash invece di indici ad albero.** I database in genere usano i B⁺-Tree, utilizzabili sia con condizioni di uguaglianza sia con query su range di valori. Gli indici hash, invece, funzionano solo con condizioni di uguaglianza.
5. **Quando utilizzare hashing dinamico.** Con i file di dimensioni molto variabili è consigliabile utilizzare tecniche di hashing dinamico (*non offerte dai DBMS più commercializzati*).

Denormalizzare uno schema

Lo scopo della normalizzazione è di separare attributi in relazione logica, per minimizzare la ridondanza ed evitare le anomalie di aggiornamento. Tali concetti a volte possono essere sacrificati per ottenere delle performance migliori su alcuni tipi di query che occorrono frequentemente. Questo processo è detto **denormalizzazione**.

Il progettista aggiunge degli attributi ad uno schema per rispondere a delle query o a dei report per ridurre gli accessi a disco, evitando operazioni di join. Ad esempio, si potrebbe scegliere di denormalizzare uno schema di relazione da 4NF a 2NF, una forma più debole, ma che ridurrebbe gli accessi a disco in quanto eviterebbe la necessità di effettuare un’operazione di join.

Memorizzazione di record ed Organizzazione dei file

Le basi di dati sono fisicamente memorizzate come file di record, che a loro volta sono tipicamente memorizzati su dischi magnetici. Verrà esaminato come vengono organizzati i database fisicamente e si descriveranno i metodi e le tecniche per rendere più efficienti le operazioni di manipolazione dei dati.

Introduzione

La collezione di dati che costituisce una base di dati computerizzata deve essere fisicamente memorizzata su un qualche **supporto di memoria** del calcolatore. Il software del DBMS potrà poi recuperare, aggiornare ed elaborare questi dati quando necessario. I supporti di memoria del calcolatore formano una *gerarchia di memoria* che comprende due categorie fondamentali:

- **Memoria principale.** Questa categoria comprende supporti di memoria su cui può operare direttamente l'*unità centrale di elaborazione* (CPU) del calcolatore, come la memoria centrale del calcolatore e le meno capienti ma più veloci memorie cache. La memoria principale di solito consente un rapido accesso ai dati ma ha una limitata capacità di memorizzazione.
- **Memoria secondaria.** Questa categoria comprende dischi magnetici, dischi ottici e nastri. Questi dispositivi hanno solitamente una maggiore capacità, costano meno ma forniscono un accesso più lento ai dati rispetto ai dispositivi di memorizzazione principale. I dati nella memoria secondaria non possono essere elaborati direttamente dalla CPU, ma devono essere prima copiati nella memoria principale.

Gerarchie di memoria e dispositivi di memorizzazione

In un moderno sistema di elaborazione i dati risiedono in e vengono trasferiti fra supporti di memorizzazione organizzati in gerarchia. La memoria più veloce è anche la più costosa ed è pertanto disponibile con minore capacità; la memoria meno veloce è costituita dai nastri magnetici ed è sostanzialmente disponibile con capacità indefinita.

A *livello di memoria principale* la **memoria cache**, costituita da una RAM statica è la più costosa. La memoria cache viene tipicamente usata dalla CPU per accelerare l'esecuzione dei programmi. Il livello successivo di memoria principale è costituito dalla DRAM (Dynamic RAM), che fornisce l'area di lavoro principale della CPU per memorizzare programmi e dati ed è comunemente della **memoria centrale**. Il vantaggio della DRAM consiste nel suo basso costo, che continua a diminuire; lo svantaggio è la sua volatilità e la minore velocità se confrontata con la RAM statica. A *livello di memorizzazione secondaria* la gerarchia comprende dischi magnetici come pure **memoria di massa** nella forma di dispositivi CD-ROM, e infine nastri, i meno costosi. La **capacità di memoria** è misurata in kilobyte ($K\text{byte}$ o 2^{10} byte), megabyte ($M\text{byte}$ o 2^{20} byte), gigabyte ($G\text{byte}$ o 2^{30}) e anche terabyte (2^{40} byte).

I programmi risiedono e vengono eseguiti nella DRAM. In genere basi di dati permanenti di grandi dimensioni risiedono in memoria secondaria, e quando necessario porzioni della base di dati sono poste in, e trascritte da, buffer di memoria centrale. Ora che i personal computer e le workstation hanno decine di megabyte di dati in DRAM sta diventando possibile caricare un'ampia frazione della base di dati in memoria centrale. In alcuni casi intere basi di dati possono essere memorizzate in memoria centrale (con un copia di backup su disco magnetico), portando a **basi di dati in memoria centrale**; esse sono particolarmente utili in applicazioni in tempo reale che richiedono tempi di risposta estremamente brevi. Tra la memoria DRAM e il disco magnetico, un'altra forma di memoria, la **memoria flash**, sta diventando comune, in particolare perché non è volatile. Le memorie flash sono memorie ad alta densità e ad alte prestazioni che usano la tecnologia EEPROM. Il vantaggio della memoria flash risiede nell'elevata velocità di accesso; lo svantaggio nel fatto che deve essere cancellato e riscritto un intero blocco alla volta.

I CD-ROM memorizzano i dati otticamente e sono letti da un laser; contengono dati preregistrati che non possono essere sovrascritti. I dischi WORM (Write Once Read Many) costituiscono una forma di

memorizzazione ottica usata per archiviare dati: essi consentono che i dati vengano scritti una volta e letti un numero qualsiasi di volte senza avere la possibilità di cancellarli. Memorizzano circa mezzo gigabyte di dati per disco e durano molto più a lungo dei dischi magnetici. I **juke-box di memorie ottiche** usano una schiera di CD-ROM, che vengono caricati nei drive (unità di lettura) su richiesta. Anche se i juke-box ottici hanno capacità nell'ordine di centinaia di gigabyte, i loro tempi di recupero sono nell'ordine di centinaia di millisecondi, e sono quindi sensibilmente più lenti dei dischi magnetici. Questo tipo di memoria non è diventato così popolare come si aspettava a causa della rapida diminuzione di costo e del rapido aumento di capacità dei dischi magnetici. Il DVD è uno standard recente per i dischi ottici che consente da quattro a quindici gigabyte di memoria per disco.

Infine, i **nastri magnetici** sono usati per archiviate e salvare grandi quantità di dati. I **juke-box di nastri** – contenenti una banca di nastri che sono catalogati e che possono essere caricati automaticamente nei drive per i nastri – stanno diventando popolari come **memoria terziaria** per conservare terabyte di dati. La locuzione **very large database** (base di dati molto ampia) non può più essere definita precisamente, perché le capacità di memorizzazione su disco stanno aumentando e i costi stanno diminuendo; potrebbe perciò essere ben presto riservata alle basi di dati che contengono decine di terabyte.

Memorizzazione di database

Le basi di dati tipicamente memorizzano grandi quantità di dati, i quali devono *persistere* per lunghi periodi di tempo. Durante questi periodi si accede ai dati e li si elabora ripetutamente. Ciò è in contrasto con la nozione di strutture dati *transitorie* che persistono solo per un tempo limitato durante l'esecuzione del programma. La maggior parte delle basi di dati è memorizzata permanentemente (o persistentemente) su memoria secondaria a disco magnetico, per le seguenti ragioni:

- Di solito le basi di dati sono troppo ampie per poter essere interamente contenute in memoria centrale;
- Le circostanze che causano una perdita permanente di dati memorizzati si verificano meno frequentemente per la memorizzazione secondaria su disco che per la memorizzazione primaria; per questo motivo ci si riferisce al disco – e ad altri dispositivi di memoria secondaria – come a **memoria non volatile**, mentre la memoria centrale è spesso chiamata **memoria volatile**;
- Il costo di memorizzazione per unità di dato è di un ordine di grandezza inferiore per il disco che per la memoria centrale.

È verosimile che alcune delle più recenti tecnologie – come i dischi ottici, i DVD e i juke-box di nastri – forniranno valide alternative all'uso dei dischi magnetici. Inoltre, in futuro le basi di dati potranno risiedere, nella gerarchia di memoria, a livelli diversi rispetto a quelli descritti in precedenza. Per il momento, tuttavia, è importante capire le proprietà e le caratteristiche dei dischi magnetici e il modo in cui i file di dati possono essere organizzati su disco, per progettare vasi di dati reali con prestazioni accettabili.

I nastri magnetici sono frequentemente usati come supporto di memoria per fare una copia di backup della base di dati, dato che la memorizzazione su nastro costa ancor meno della memorizzazione su disco. Però l'accesso ai dati su nastro è piuttosto lento. I dati memorizzati su nastro sono **off-line**; cioè è necessario l'intervento di un operatore – o un dispositivo automatico di caricamento – per caricare un nastro prima che questi dati si rendano effettivamente disponibili. Al contrario, i dischi sono dispositivi **on-line** a cui si può accedere direttamente in ogni momento.

Le tecniche usate per memorizzare grandi quantità di dati strutturati su disco sono importanti per i progettisti di una base di dati, il DBA e gli implementatori di un DBMS. I progettisti della base di dati e il DBA devono conoscere i vantaggi e gli svantaggi di ciascuna tecnica di memorizzazione quando progettano, implementano e gestiscono una base di dati su uno specifico DBMS. Di solito il DBMS fornisce molte opzioni per organizzare i dati, e il processo di **progettazione fisica di una base di dati** prevede la scelta, fra le opzioni, delle particolari tecniche di organizzazione dei dati che meglio si adattano ai requisiti propri dell'applicazione. Gli

implementatori di sistema DBMS devono studiare le tecniche di organizzazione dei dati per poterle implementare efficientemente e perciò fornire a DBA e utenti un numero sufficiente di opzioni.

Le applicazioni tipiche di basi di dati hanno bisogno per l'elaborazione solo di una piccola porzione alla volta della basi di dati. Tutte le volte che c'è bisogno di una certa porzione dei dati, essa deve essere localizzata su disco, copiata in memoria centrale per l'elaborazione, e quindi riscritta su disco se i dati sono cambiati. I dati memorizzati su disco sono organizzati come **file di record**. Ogni record è costituito da una collezione di valori di dati che possono essere interpretati come fatti relativi alle entità, ai loro attributi e alle loro associazioni. I record dovrebbero essere memorizzati su disco in modo da rendere possibile individuarne efficientemente la collocazione ogni volta che se ne ha bisogno.

Ci sono molte **organizzazioni primarie dei file** che determinano come sono *collocati fisicamente* sul disco i record di un file, e perciò *come si può accedere ad essi*. Un *file heap (file non ordinato)* colloca i record su disco senza un ordine particolare, semplicemente aggiungendo nuovi record alla fine del file, mentre un *file sequenziale (file ordinato)* tiene i record in ordine sulla base del valore di un campo particolare (detto chiave di ordinamento). Un *file hash* usa una funzione hash applicata a un campo particolare (detto chiave hash) per determinare la collocazione di un record su disco. Altre organizzazioni primarie dei file, come i *B-Tree*, usano per l'appunto strutture ad albero. Un'**organizzazione secondaria, o struttura di accesso ausiliaria**, consente un accesso efficiente ai record di un file basata su *campi alternativi* rispetto a quelli che sono stati usati per l'organizzazione primaria dei file. La maggior parte di questi esistono come indici.

Dispositivi di memoria secondaria

Verranno descritte alcune caratteristiche dei dischi e dei nastri magnetici.

Descrizione dell'hardware di dispositivi a disco

I dischi magnetici sono usati per memorizzare grandi quantità di dati. La più elementare unità dati su disco è il singolo **bit** di informazione. Magnetizzando un'area su disco in certi modi, si può far sì che essa rappresenti un valore di bit 0 o 1. Per codificare le informazioni, i bit sono raggruppati in **byte**. Le dimensioni dei byte vanno da 4 a 8 bit, a seconda del computer e del dispositivo. Successivamente, verrà supposto che un carattere sia memorizzato in un singolo byte, e si useranno i termini *byte* e *carattere* indifferentemente. Per **capacità** di un disco si intende il numero di byte che esso può memorizzare, che di solito è molto grande. I

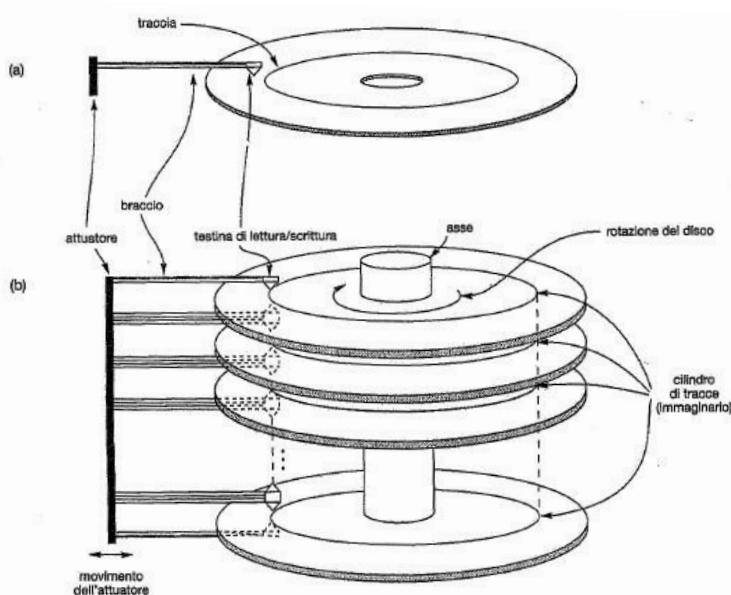


Figura 2.1 – (a) Un disco a singola faccia con l'hardware di lettura/scrittura.
(b) Una pila di dischi con l'hardware di lettura/scrittura

piccoli floppy disk usati nei microcomputer tipicamente contengono da 400 Kbyte a 1,5 MByte; i dischi rigidi per questi elaboratori tipicamente contengono da molte centinaia di Mbyte a qualche Gbyte; e le grosse pile di dischi (*disk packs*) usate nei minicomputer e nei mainframe hanno capacità che vanno da qualche decina a centinaia di Gbyte. Le capacità dei dischi continuano ad aumentare man mano che la tecnologia migliora. Qualunque sia la loro capacità, i dischi sono tutti fatti di materiale magnetico della forma di sottile disco circolare (Figura 2.1a) e protetto tramite una copertura di plastica o di acrilico. Un disco è a **singola faccia (single-sided)** se memorizza informazioni su uno solo dei

suoi lati e a **doppia faccia (double-sided)** se sono usati entrambi i lati. Per aumentare la capacità di memorizzazione i dischi sono assemblati in una **pila di dischi** (Figura 2.1b), che può comprendere molti dischi e perciò molti lati. L'informazione è memorizzata sulla faccia di un disco su circonferenze concentriche di **piccola larghezza**, ciascuna delle quali ha un diverso diametro. Ogni circonferenza è detta **traccia (track)**. Per le pile di dischi le tracce con lo stesso diametro sulle varie facce sono dette **cilindro** per la struttura che formerebbero se connesse nello spazio. Il concetto di cilindro è importante perché i dati memorizzati in un cilindro possono essere recuperati molto più velocemente se fossero distribuiti tra cilindri diversi.

Il numero di tracce per un disco va da qualche centinaio a qualche migliaio, e la capacità di ciascuna traccia va tipicamente dalle decine di Kbyte fino a 150 Kbyte. Dato che una traccia di solito contiene un grande quantità di informazione, essa viene suddivisa in blocchi o settori più piccoli. La suddivisione di una traccia in **settori** è codificata nell'hardware sulla faccia del disco e non può essere cambiata. Un tipo di organizzazione a settori chiama settore una porzione di traccia che sottende un fissato angolo al centro (Figura 2.2a). Sono possibili molte altre organizzazioni a settori, una delle quali consiste nell'avere settori che sottendono angoli al centro più piccoli man mano che ci si allontana dal centro stesso, mantenendo così una densità di memorizzazione costante (Figura 2.2b). Non tutti i dischi hanno le tracce suddivise in settori.

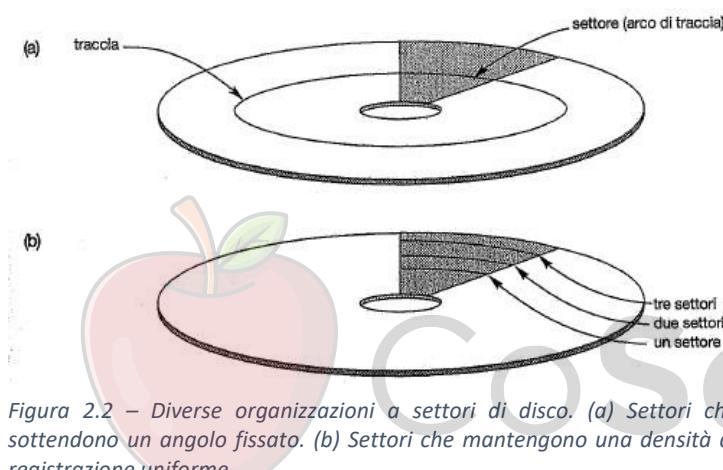


Figura 2.2 – Diverse organizzazioni a settori di disco. (a) Settori che sottendono un angolo fisso. (b) Settori che mantengono una densità di registrazione uniforme.

La divisione di una traccia in **blocchi di disco** (o **pagine**) di uguale dimensione è fissata dal sistema operativo durante la **formattazione** (o **inizializzazione**) del disco. La dimensione di un blocco è fissata durante l'inizializzazione e non può essere cambiata dinamicamente. Tipiche dimensioni di un blocco di disco vanno da 512 a 4096 byte. Un disco con settori codificati nell'hardware ha spesso i settori che vengono suddivisi in blocchi durante l'inizializzazione del disco. I blocchi sono separati da **spazi tra blocchi** di dimensioni fisse, che comprendono informazioni di controllo codificate

appositamente, scritte durante l'inizializzazione del disco. Queste informazioni sono usate per determinare quale blocco della traccia segue ciascuno spazio tra blocchi.

C'è un continuo miglioramento nelle capacità di memorizzazione e nei tassi di trasferimento associati ai dischi; la memorizzazione su disco sta anche diventando via via più economica – costando attualmente solo una frazione di dollaro per megabyte. I costi stanno diminuendo così rapidamente che si prevedono costi dell'ordine di un centesimo di dollaro per megabyte.

Un disco è un dispositivo indirizzabile ad *accesso casuale*. Il trasferimento dati tra la memoria centrale e il disco avviene in unità di blocchi di disco. L'**indirizzo hardware** di un blocco – una combinazione di numero di faccia, numero di traccia (nella faccia) e numero di blocco (nella traccia) – è fornito all'hardware dell'input/output (I/O) del disco. Viene anche fornito l'indirizzo di un **buffer** – un'area riservata di locazioni contigue in memoria centrale che può contenere un blocco. Con un comando **read** il blocco dal disco è copiato nel buffer, mentre con un comando **write** il contenuto del buffer è copiato nel blocco di disco. Talora possono essere trasferiti come una cosa sola molti blocchi contigui, detti **cluster**. In questo caso la dimensione del buffer è adattata per corrispondere al numero di byte presenti nel cluster.

L'effettivo meccanismo hardware che legge o scrive un blocco è la **testina di lettura/scrittura**, che fa parte di un sistema detto **unità disco (disk drive)**. Un disco o una pila di dischi sono inseriti nell'unità disco, che ha un motore che fa ruotare i dischi. Una testina di lettura/scrittura è costituita da un componente elettronico fissato a un **braccio meccanico**. Le pile di dischi con più facce sono controllate da molte testine di lettura/scrittura – una per ogni faccia (Figura 2.1b) Tutti i bracci sono collegati a un **attuatore** unito a un altro

motore elettrico, che muove le testine di lettura/scrittura all'unisono e le posiziona esattamente sopra il cilindro di tracce specificato nell'indirizzo di un blocco.

Le unità disco per i dischi rigidi fanno ruotare la pila di dischi ininterrottamente a una velocità costante. Una volta che la testina di lettura/scrittura è stata posizionata sulla traccia corretta e il blocco specificato nell'indirizzo di blocco si muove sotto di essa, il componente elettronico della testina di lettura/scrittura viene attivato per trasferire i dati. Alcune unità a disco hanno testine di lettura/scrittura fisse, con tante testine quanto sono le tracce. Queste unità sono dette **dischi a testina fissa**, mentre le unità a disco con un attuatore sono dette **dischi a testina mobile**. Per i dischi a testina fissa, una traccia o cilindro è selezionata tramite la commutazione elettronica all'appropriata testina di lettura/scrittura piuttosto che tramite un effettivo movimento meccanico; di conseguenza questa unità è molto più veloce. Tuttavia, il costo delle testine di lettura/scrittura aggiuntive è piuttosto alto, e pertanto i dischi a testina fissa non sono usati comunemente. Un **disk controller** (controllore di disco), tipicamente inserito nell'unità disco, controlla l'unità disco e la interfaccia al sistema di elaborazione. Una delle interfacce standard usate oggi per le unità disco nei PC e nelle workstation è detta **SCSI** (Small Computer Storage Interface). Il controller accetta comandi di I/O di alto livello e intraprende azioni appropriate per posizionare il braccio, facendo sì che abbia luogo l'azione di lettura/scrittura. Per trasferire un blocco di disco, dato il suo indirizzo, il controller deve prima di tutto posizionare meccanicamente la testina di lettura/scrittura sulla traccia corretta. Il tempo necessario per fare ciò è detto **tempo di posizionamento (seek time)**. Tempi di posizionamento tipici sono di 12-14 msec per i personal computer e di 8-9 msec per i server. Segue un ritardo – detto **latenza** – per attendere che l'inizio del blocco desiderato ruoti fino alla posizione sotto la testina di lettura/scrittura. Infine, c'è bisogno di un tempo aggiuntivo per trasferire i dati, detto **tempo di trasferimento di blocco**. Perciò il tempo totale necessario per localizzare e trasferire un blocco arbitrario, dato il suo indirizzo, è costituito dalla somma del tempo di posizionamento, ritardo di rotazione e tempo di trasferimento di blocco. Il tempo di posizionamento e il ritardo di rotazione sono solitamente molto più grandi del tempo di trasferimento di blocco. Per rendere più efficiente il trasferimento di più blocchi, è comune trasferire molti blocchi consecutivi sulla stessa traccia o cilindro. Ciò elimina il tempo di posizionamento e il ritardo di rotazione per tutti i blocchi tranne il primo e può avere come risultato un sostanziale risparmio di tempo quando vengono trasferiti numero blocchi contigui. Di solito i costruttori di dischi forniscono un **tasso di trasferimento di una molte di dati (bulk transfer rate)** per calcolare il tempo richiesto per trasferire blocchi consecutivi.

Il tempo necessario per localizzare e trasferire un blocco di disco è dell'ordine dei millisecondi, andando di solito dai 12 ai 60 msec, ma il trasferimento dei blocchi successivi può richiedere solo da 1 a 2 msec ciascuno. Molte tecniche di ricerca traggono profitto dal recupero di blocchi consecutivi quando cercano dati su disco. In ogni caso un tempo di trasferimento dell'ordine dei millisecondi è considerato piuttosto alto se confrontato con il tempo richiesto per elaborare dati in memoria centrale dalle CPU attuali. Perciò la localizzazione di dati su disco è un *collo di bottiglia importante* nelle applicazioni di basi di dati. Le strutture di file esaminate successivamente tentano di *ridurre al minimo il numero di trasferimenti di blocchi* necessari per localizzare e trasferire i dati richiesti da disco a memoria centrale.

Bufferizzazione di blocchi

Quando devono essere trasferiti da disco a memoria centrale numerosi blocchi, e sono noti tutti i loro indirizzi, per accelerare il trasferimento possono essere riservati in memoria centrale molti **buffer** (aree di memoria). Mentre viene letto o scritto un buffer, la CPU può elaborare i dati nell'altro buffer. Ciò è possibile perché esiste un processore indipendente per l'I/O di disco (controller) che, una volta avviato, può procedere nel trasferire un blocco di dati tra la memoria e il disco indipendentemente dall'operazione della CPU, e in parallelo con essa.

In Figura 2.3 è illustrato come due processi possano avanzare in parallelo. I processi A e B sono eseguiti **concorrentemente** e in modo **interleaved** (interfogliamento), mentre i processi C e D sono eseguiti **concorrentemente e in parallelo**. Quando una sola CPU controlla più processi, l'esecuzione parallela non è

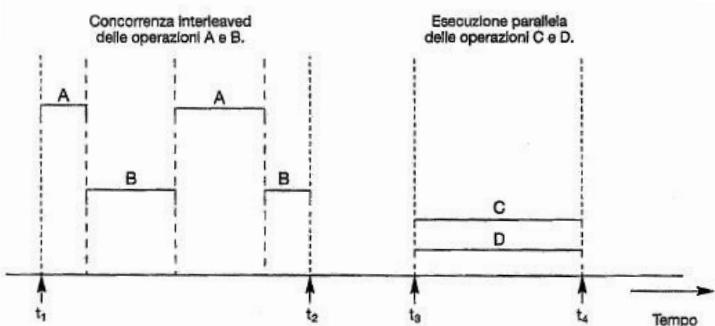


Figura 2.3 – Confronto tra la concorrenza interleaved e l'esecuzione in parallelo.

In Figura 2.4 è illustrato come la lettura e l'elaborazione possano procedere in parallelo quando il tempo richiesto per elaborare un blocco di disco in memoria è minore del tempo richiesto per leggere il blocco successivo e riempire un buffer. La CPU può cominciare a elaborare un blocco una volta che il suo trasferimento in memoria centrale è completato; allo stesso tempo il processore dell'I/O di disco può essere impegnato nella lettura e nel trasferimento del blocco successivo in un buffer diverso. Questa tecnica è detta **doppia bufferizzazione** e può essere usata anche per scrivere un flusso continuo di blocchi da memoria a disco. Essa consente una lettura o scrittura continua di dati su blocchi di disco consecutivi, che elimina il tempo di posizionamento e il ritardo di rotazione per i trasferimenti di tutti i blocchi escluso il primo. Inoltre, i dati sono tenuti pronti per essere elaborati, riducendo così il tempo di attesa nei programmi.

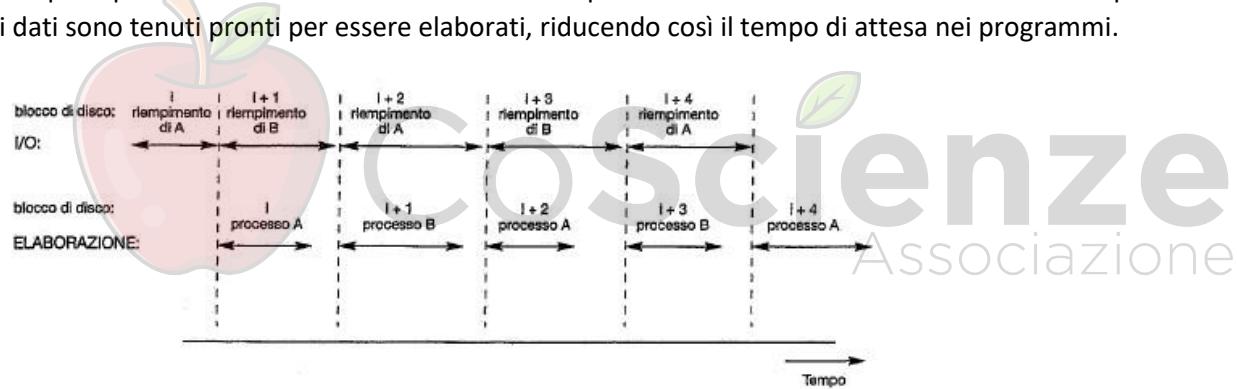


Figura 2.4 – Uso di due buffer, A e B, per la lettura da disco.

Collocazione su disco dei record di un file

I dati sono normalmente memorizzati sotto forma di **record**. Ogni record consiste di una collezione di **valori** collegati, dove ogni valore è formato da uno o più byte e corrisponde a un particolare **campo** del record. I record di solito descrivono le entità e i loro attributi. Per esempio, un record **IMPIEGATO** rappresenta un'entità impiegato, e il valore di ciascun campo del record specifica alcuni attributi di quell'impiegato, come **NOME**, **DATA_NASCITA**, **STIPENDIO** o **SUPERVISORE**. Una collezione di nomi di campi e dei tipi di dati corrispondenti costituisce una definizione di **tipo di record** o **formato di record**. Un **tipo di dati**, associato a ciascun campo, specifica il tipo di valori che possono essere assunti da quel campo.

Il tipo di dati di un campo è di solito uno dei tipi di dati standard usati in programmazione. Questi comprendono i tipi di dati numerici (integer, long integer o floating point), stringhe di caratteri (a lunghezza fissa o variabile), booleani (che assumono solo i valori 0 e 1 o VERO e FALSO), e talora i tipi di dati codificati in modo speciale **date** e **time**. Per ciascun sistema di elaborazione il numero di byte richiesti per ogni tipo di dati è fissato. Un integer può richiedere 4 byte, un long integer 8 byte, un numero reale 4 byte, un booleano

possibile. Però i processi possono ancora essere eseguiti concorrentemente in modo interleaved. La bufferizzazione è più utile quando i processi possono essere eseguiti concorrentemente in parallelo, o perché disponibile un processore di I/O di disco separato, o perché esistono più processori di CPU.

1 byte, una data 10 byte e una stringa a lunghezza fissa di k caratteri k byte. Le stringhe a lunghezza variabile possono richiedere tanti byte quanti sono i caratteri in ogni valore del campo.

In applicazioni di basi di dati recenti può sorgere la necessità di memorizzare voci di dati che consistono di grandi oggetti non strutturati, rappresentanti immagini, flussi digitalizzati audio o video, o testo libero. Queste sono indicate come **BLOB** (Binary Large Object). Una voce di dati BLOB è tipicamente memorizzata separatamente dal suo record in un pool di blocchi di disco, e ci si limita a inserire nel record un puntatore al BLOB.

File, record a lunghezza fissa e record a lunghezza variabile

Un **file** è una sequenza di record. In molti casi tutti i record presenti in un file fanno parte dello stesso tipo di record. Se ogni record nel file ha esattamente la stessa dimensione (in byte), si dice che il file è costituito da **record a lunghezza fissa**. Se record diversi nel file hanno dimensioni diverse, si dice che il file è costituito da **record a lunghezza variabile**. Un file può avere record a lunghezza variabile per molte ragioni:

- I record del file sono dello stesso tipo di record, ma uno o più campi sono di dimensione variabile (**campi di lunghezza variabile**), ad esempio, il campo NOME di IMPIEGATO può essere un campo di lunghezza variabile;
- I record del file sono dello stesso tipo di record, ma uno o più dei campi può avere valori multipli per singoli record; un tale campo è detto **repeating field** (campo che si ripete) e un gruppo di valori per quel campo è spesso detto **repeating group** (gruppo che si ripete);
- I record del file sono dello stesso tipo di record, ma uno o più campi sono **opzionali**, cioè possono assumere valori per alcuni ma non per tutti i record del file (**campi opzionali**);
- Il file contiene record di *diversi tipi di record* e perciò di dimensione variabile (**file misto**). Ciò potrebbe verificarsi se record collegati di tipi diversi fossero *raggruppati (clustered)* su blocchi di disco; ad esempio i record VOTAZIONE di uno specifico studente possono essere collocati di seguito al record di quello STUDENTE.

I record a lunghezza fissa IMPIEGATO in Figura 2.5(a) presentano una dimensione di record di 71 byte. Ogni record ha gli stessi campi, e le lunghezze dei campi sono fisse, cosicché il sistema può individuare la posizione del byte di inizio di ciascun campo relativamente alla posizione iniziale del record. Ciò facilita la localizzazione dei valori del campo da parte dei programmi che accedono a questi file. Si noti che è possibile rappresentare un file che da un punto di vista logico dovrebbe avere record a lunghezza variabile come un file di record a lunghezza fissa. Ad esempio, nel caso di campi opzionali sarebbe possibile aver inserito *tutti i campi* in ogni

record del file, ma poi memorizzare uno speciale valore nullo se non esiste alcun valore per quel campo. Per un campo che si ripete, si potrebbero allocare in ogni record tanti spazi quant'è il *massimo numeri di valori* che il campo può assumere. In entrambi i casi viene sprecato spazio quando certi record non presentano valori per tutti gli spazi fisici forniti in ciascun record. Si considerino ora altre opzioni per formattare record di un file di record a lunghezza variabile. Per *campi di lunghezza variabile* ogni record presenta un valore in corrispondenza a ogni campo, ma non è nota la lunghezza esatta dei valori di

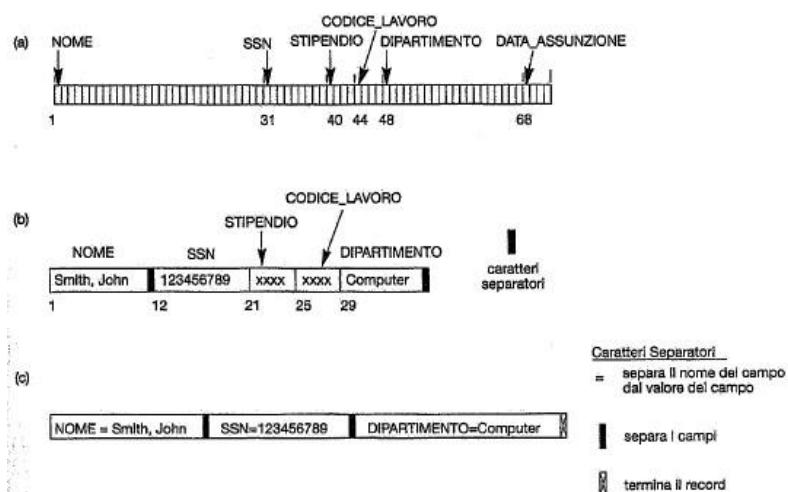


Figura 2.5 – Tre formati di memorizzazione di record. (a) Un record a lunghezza fissa con sei campi e dimensione di 71 byte. (b) Un record con due campi di lunghezza variabile e tre campi di lunghezza fissa. (c) Un record a campi variabili con tre tipi di caratteri separatori.

alcuni campi. Per determinare i byte che all'interno di un particolare record rappresentano ogni campo, si possono usare speciali caratteri **separatori** (come ? o % o \$) – che non si presentano in nessun valore del campo – per terminare campo di lunghezza variabile (Figura 2.5b), oppure memorizzare la lunghezza in byte del campo in quel record, prima del valore del campo.

Un file di record con *campi opzionali* può essere formattato in diversi modi. Se il numero totale di campi per il tipo di record è grande, ma il numero di campi che si presentano effettivamente in un record tipico è piccolo, è possibile inserire in ciascun record una sequenza di coppie <nome-campo, valore-campo> piuttosto che solo i valori del campo. In Figura 2.5(c) sono usati tre tipi di caratteri separatori, anche se si potrebbe usare lo stesso carattere separatore per i primi due scopi – separare il nome del campo dal valore del campo è separare un campo dal successivo. Un'opzione più pratica è quella di assegnare un breve codice **tipo di campo** – ad esempio un numero intero – a ciascun campo e inserire in ciascun record una sequenza di coppie <tipo-campo, valore-campo> piuttosto che coppie <nome-campo, valore-campo>.

Un ha bisogno di un carattere separatore per separare i valori del campo che si ripetono e di un altro carattere separatore per indicare la fine del campo. Infine, per un file che comprende *record di tipi diversi*, ogni record è preceduto da un indicatore di **tipo di record**. Comprensibilmente i programmi che elaborano file di record a lunghezza variabile – che sono di solito parte del file system e perciò nascosti ai programmatore comuni – devono essere più complessi di quelli per record a lunghezza fissa, dove la posizione di partenza e la dimensione di ogni campo sono note e fissate.

Ripartizione dei record in blocchi e confronto tra record con spanning e record senza spanning

I record di un file devono essere ripartiti su blocchi di disco perché un blocco è l'*unità di trasferimento dati* tra disco e memoria. Quando la dimensione del blocco è maggiore di quella del record, ogni blocco conterrà numerosi record, anche se alcuni file possono avere record inusualmente grandi che non possono trovar posto in un solo blocco. Si supponga che la dimensione del blocco sia di B byte. Per un file di record a lunghezza fissa della dimensione di R byte, con $B \geq R$, è possibile inserire $bfr = [B/R]$ record per blocco, dove la $[(x)]$ (*funzione di floor*) arrotonda per difetto il numero x ad un intero. Il valore bfr è detto **fattore di blocco (blocking factor)** per il file. In generale R può non dividere B esattamente, cosicché in ogni blocco si ha un certo spazio inutilizzato, pari a

$$B - (bfr * R) \text{ byte}$$

Per utilizzare questo spazio si può memorizzare parte di un record in un blocco e il resto in un altro. Un **puntatore** alla fine del primo blocco punta al blocco che contiene il resto del record nel caso in cui non sia il blocco consecutivo su disco. Questa organizzazione è detta **spanned** (estesa), perché i record possono estendersi su più di un blocco. Ogni volta che un record è più grande di un blocco *si deve* usare un'organizzazione spanned. Se ai record non è concesso di attraversare i confini di un blocco, l'organizzazione è detta **unspanned**. Essa è usata con record a lunghezza fissa che hanno $B > R$, perché consente a ciascun record di cominciare in una locazione nota del blocco, semplificando l'organizzazione dei record. Per record a lunghezza variabile può essere usata sia un'organizzazione spanned sia un'organizzazione unspanned. Se il record tipico è grande, è vantaggioso usare l'organizzazione con spanning per ridurre lo spazio perso in ciascun blocco. In Figura 2.6 sono messi a confronti i due tipi di organizzazione. Per record a lunghezza variabile che usano l'organizzazione spanned, ogni blocco può memorizzare un diverso numero di record. In questo caso il fattore di blocco bfr rappresenta il numero *medio* di record per blocco per il file. Si può usare bfr per calcolare il numero di blocchi b necessari per un file di r record:

$$b = [(r/bfr)] \text{ blocchi}$$

dove la $[(x)]$ (*funzione ceiling*) arrotonda per eccesso il valore di x al primo intero.

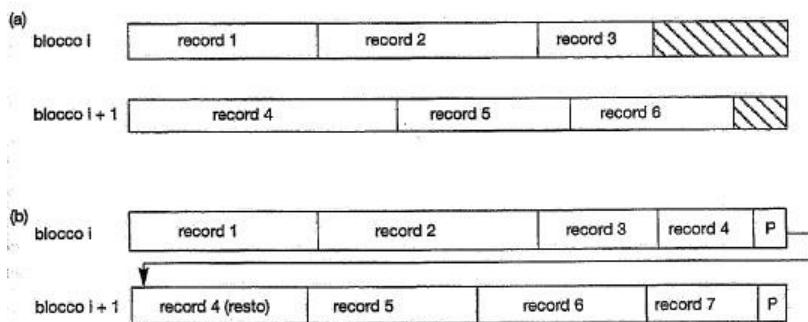


Figura 2.6 – Tipi di organizzazione dei record. (a) Senza spanning. (b) Con spanning.

Allocazione dei blocchi di un file su disco

Ci sono molte tecniche standard per allocare i blocchi di un file su disco. Nell'**allocazione continua** i blocchi del file sono allocati su blocchi di disco consecutivi: ciò rende molto veloce la lettura dell'intero file con l'uso della doppia bufferizzazione, ma allo stesso tempo rende difficile l'aumento delle dimensioni del file. Nell'**allocazione collegata** ogni blocco contiene un puntatore al successivo blocco del file: ciò facilita l'aumento delle dimensioni del file ma rallenta la lettura dell'intero file. Una combinazione delle due alloca **cluster** di blocchi di disco consecutivi, con i cluster collegati tra loro. I cluster sono talora detti **segmenti del file** o **estensioni del file**. Un'altra possibilità è quella di usare un'**allocazione indicizzata**, dove uno o più **blocchi di indici** contengono puntatori agli effettivi blocchi del file. È anche comune usare combinazioni di queste tecniche.

Header dei file

Un **header di file** (*intestazione*) o **descrittore di file** contiene informazioni su un file, necessarie ai programmi di sistema che accedono ai record del file. L'header comprende informazioni per determinare gli indirizzi di disco dei blocchi del file, nonché per le descrizioni dei formati dei record, che possono comprendere lunghezze di campo e ordine dei campi all'interno di un record per record a lunghezza fissa senza spanning, e codici di tipo di campo, caratteri separatori e codici di tipo di record per record a lunghezza variabile.

Per la ricerca di un record su disco, uno o più blocchi sono copiati nel buffer di memoria centrale. Quindi i programmi ricercano il record o i record desiderati dentro i buffer, usando le informazioni presenti nell'header del file. Se l'indirizzo del blocco che contiene il record desiderato non è noto, i programmi di ricerca devono effettuare una **ricerca lineare** attraverso i blocchi del file. Ogni blocco del file è copiato in un buffer ed esaminato fino a che il record è localizzato o tutti i blocchi del file sono stati esaminati senza successo. Ciò può essere molto dispendioso in termini di tempo per un file grande. Lo scopo di una buona riorganizzazione di file è quello di localizzare il blocco che contiene un record desiderato con il minimo numero di trasferimenti di blocco.

Operazioni sui file

Le operazioni su file sono di solito raggruppate in **operazioni di recupero (retrieval operations)** e **operazioni di aggiornamento (update operations)**. Le prime non cambiano alcun dato nel file, ma si limitano a localizzare certi record in modo tale che i valori dei loro campi possano essere esaminati ed elaborati. Le seconde cambiano il file con l'inserimento o la cancellazione di record o con la modifica dei valori di alcuni campi. In entrambi i casi è possibile che si debba **selezionare** uno o più record per il recupero, la cancellazione o la modifica basandosi su una **condizione di selezione** (o **condizione di filtraggio**), che specifica i criteri che il record o i record desiderati devono soddisfare.

Si consideri un file **IMPIEGATO** con campi **NOME**, **SSN**, **STIPENDIO**, **CODICE_LAVORO** e **DIPARTIMENTO**. Una **condizione di selezione semplice** può comportare un confronto di egualanza su un certo valore di campo – ad esempio (**SSN** = '123456789') o (**DIPARTIMENTO** = 'Ricerca'). Condizioni più complesse possono

coinvolgere altri tipi di operatori di confronto, come $>$ o \geq ; un esempio è ($STIPENDIO \geq 30000$). Il caso generale consiste nell'avere come condizione di selezione un'espressione booleana arbitraria sui campi del file.

Le operazioni di ricerca su file sono generalmente basate su condizioni di selezione semplici. Una condizione complessa deve essere decomposta dal DBMS (o dal programmatore) per estrarre una condizione semplice che possa essere usata per localizzare i record su disco. Ogni record localizzato viene poi esaminato per decidere se soddisfa l'intera condizione di selezione. Ad esempio, si può estrarre la condizione semplice ($DIPARTIMENTO = 'Ricerca'$) dalla condizione complessa ($(STIPENDIO \geq 30000) \text{ AND } (DIPARTIMENTO = 'Ricerca')$); ogni record che soddisfa ($DIPARTIMENTO = 'Ricerca'$) viene localizzato e poi esaminato per vedere se soddisfa anche ($STIPENDIO \geq 30000$).

Quando molti record di un file soddisfano una condizione di ricerca, viene inizialmente localizzato il *primo* record . relativamente alla sequenza fisica dei record del file – nominato **record corrente**. Le operazioni di ricerca seguenti cominciano da questo record e localizzano il *successivo* record del file che soddisfa la condizione.

Le operazioni effettive per localizzare e accedere ai record di un file variano da sistema a sistema. Si presenta qui di seguito un insieme di operazioni rappresentative. Tipicamente programmi di alto livello, come i programmi software del DBMS, accedono ai record usando questi comandi, e perciò talvolta nelle seguenti descrizioni ci si riferirà a **variabili di programma**.

- *Open*: prepara il file per la lettura o la scrittura. Alloca buffer appropriati (tipicamente almeno due) per contenere blocchi del file prelevati da disco, e recupera l'header del file. Imposta il file pointer (puntatore al file) all'inizio del file.
- *Reset*: imposta il file pointer di un file aperto all'inizio del file.
- *Find* (o *Locate*): cerca il primo record che soddisfa una condizione di ricerca. Trasferisce il blocco che contiene quel record in un buffer in memoria centrale (se non è già là). Il file pointer punta al record nel buffer ed esso diventa il *record corrente*. Talvolta vengono usati verbi diversi per indicare se il record localizzato deve essere recuperato o aggiornato.
- *Read* (o *Get*): copia il record corrente dal buffer in una variabile di programma del programma utente. Questo comando può anche far avanzare il puntatore dal record corrente al record successivo del file, il che può richiedere la lettura da disco del successivo blocco del file.
- *FindNext*: ricerca nel file il successivo record che soddisfa la condizione di ricerca. Trasferisce il blocco che contiene quel record in un buffer di memoria centrale (se non si trova già là). Viene localizzato il record nel buffer ed esso diventa il *record corrente*.
- *Delete*: cancella il record corrente e (alla fine) aggiorna il file su disco per rispecchiare la cancellazione.
- *Modify*: modifica i valori di alcuni campi del record corrente e (alla fine= aggiorna il file su disco per rispecchiare la modifica.
- *Insert*: inserisce un nuovo record nel file localizzando il blocco in cui deve essere inserito il record, trasferendo quel blocco in un buffer in memoria centrale (se non è già là), scrivendo il record nel buffer e (alla fine) scrivendo il contenuto del buffer su disco per rispecchiare l'inserimento.
- *Close*: completa l'accesso al file rilasciando i buffer ed eseguendo tutte le altre operazioni di pulizia necessarie.

Le precedenti operazioni (tranne Open e Close) sono dette operazioni **un-record-all-a-volta**, perché ogni operazione si applica a un solo record. È possibile snellire le operazioni Find, FindNext e Read in una sola operazione, Scan.

- *Scan*: se sul file sono appena state applicate le operazioni Open o Reset, restituisce il primo record; altrimenti fornisce il record successivo, se con l'operazione è specificata una condizione, il record restituito è il primo o il successivo record che soddisfa la condizione.

Nei sistemi di basi di dati possono essere applicate a un file operazioni aggiuntive di più alto livello **insieme-alla-volta**. Esempi di queste operazioni sono i seguenti.

- *FindAll*: localizza *tutti* i record nel file che soddisfano una condizione di ricerca.
- *FindOrdered*: recupera tutti i record nel file in un certo ordine specificato.
- *Reorganize*: comincia il processo di riorganizzazione. Come si avrà modo di vedere, alcune organizzazioni di file richiedono una riorganizzazione periodica. Un esempio consiste nel riordinare i record del file classificandoli sulla base di un campo specificato.

A questo punto vale la pena di notare la differenza tra i termini *organizzazione di file* e *metodo di accesso*. Un'**organizzazione di file** si riferisce all'organizzazione dei dati di un file in record, blocchi e strutture di accesso; ciò comprende il modo in cui i record e i blocchi sono posti nel supporto di memorizzazione il modo in cui sono collegati. Un **metodo di accesso**, d'altro lato, fornisce un gruppo di operazioni – come ad esempio quelle elencate in precedenza – che possono essere applicate a un file. In generale è possibile applicare molti metodi di accesso a una stessa organizzazione di file alcuni metodi di accesso, tuttavia, possono essere applicati solo a file organizzati in certi modi. Ad esempio, non è possibile applicare un metodo di accesso a indici a una file che non ha un indice (si vedrà in seguito).

Di solito ci si attende di usare alcune condizioni di ricerca piuttosto che altro. Alcuni file possono essere **statici**, nel senso che le operazioni di aggiornamento sono raramente eseguite su di essi; altri file, più **dinamici**, possono cambiare frequentemente, cosicché le operazioni di aggiornamento vengono costantemente applicate ad essi. Un'organizzazione di file di successo dovrebbe consentire di eseguire il più efficientemente possibile le operazioni che ci si attende di *eseguire frequentemente* sul file. Ad esempio, si consideri il file IMPIEGATO (Figura 2.6a) che contiene i record relativi agli attuali impiegati di un'azienda. Ciò che ci si attende è di dover inserire record (quando vengono assunti impiegati), cancellare record (quando impiegati lasciano l'azienda) e modificare record (ad esempio, quando cambia lo stipendio o il lavoro di un impiegato). La cancellazione o la modifica di un record richiede una condizione di selezione per identificare un particolare record o insieme di record. Anche il recupero di uno o più record richiede una condizione di selezione. Per applicazioni che entrano in conflitto sull'ordinamento di determinati record, alle volte non si riesce a porre una soluzione a tutte, ma viene scelto un compromesso che tenga conto dell'importanza della combinazione prevista di operazioni di recupero e aggiornamento.

File di record non ordinati (file heap)

In questo tipo di organizzazione, la più semplice e la più fondamentale, i record sono collocati nel file nell'ordine in cui sono inseriti, cosicché i nuovi record sono inseriti alla fine del file. Un'organizzazione di questo tipo è detta **file heap** (*file sequenziali*). Essa viene spesso usata con percorsi di accesso opzionali, come gli indici secondari (in seguito), o per raccogliere e memorizzare record di dati in vista di un uso futuro.

L'inserimento di un nuovo record è *molto efficiente*: l'ultimo blocco di disco del file viene copiato in un buffer, viene aggiunto il nuovo record e poi il blocco viene **riscritto** sul disco. L'indirizzo dell'ultimo blocco del file viene tenuto nell'header del file. Però la ricerca di un record attraverso una qualsiasi condizione di ricerca comporta una **ricerca lineare** sul file blocco per blocco – una procedura dispendiosa. Se un solo record soddisfa la condizione di ricerca, allora, in media, un programma leggerà (portandoli in memoria) e ispezionerà metà dei blocchi del file prima di trovare il record. Per un file costituito da b blocchi, ciò può richiedere in media l'ispezione di $(b/2)$ blocchi. Se nessun record o viceversa molti record soddisfano la condizione di ricerca, il programma deve leggere e ispezionare tutti i b blocchi del file.

Per cancellare un record, un programma deve dapprima trovare il suo blocco, copiare il blocco in un buffer, quindi cancellare il record dal buffer e infine **riscrivere il blocco** su disco. Ciò lascia spazio inutilizzato nel

blocco di disco. La cancellazione di un gran numero di record in questo modo ha come risultato uno spreco di spazio di memoria. Un'altra tecnica usata per la cancellazione di record consiste nel tenere memorizzato con ogni record un bit o byte supplementare, detto **indicatore di cancellazione**. Un record viene cancellato semplicemente ponendo l'indicatore di cancellazione a un certo valore. Un valore diverso dell'indicatore indica un record valido (cioè non cancellato). I programmi di ricerca considerano solo i record validi di ogni blocco, quando conducono la loro ricerca. Entrambe queste tecniche di cancellazione richiedono una **riorganizzazione** periodica del file per recuperare lo spazio inutilizzato dei record cancellati. Durante la riorganizzazione si accede consecutivamente ai blocchi del file, e i record vengono compattati rimuovendo i record cancellati. Dopo una riorganizzazione di questo tipo i blocchi sono di nuovo riempiti fino al limite delle loro capacità. Un'altra possibilità è quella di usare lo spazio lasciato libero dai record cancellati quando si inseriscono nuovi record, anche se ciò richiede una contabilità aggiuntiva per tener traccia delle locazioni vuote.

Per un file non ordinato è possibile usare sia un'organizzazione spanned sia un'organizzazione unspanned; inoltre esso può essere usato con record a lunghezza fissa o con record a lunghezza variabile. La modifica di un record a lunghezza variabile può richiedere la cancellazione del record vecchio e l'inserimento di un record modificato, perché il record modificato può non trovare posto nel suo vecchio spazio su disco.

Per leggere tutti i record ordinati secondo i valori di un certo campo si può creare una copia ordinata del file. L'ordinamento è un'operazione dispendiosa per un grande file su disco; vengono per queste usate tecniche speciali per l'**ordinamento esterno**.

Per un file di *record a lunghezza fissa* non ordinati, che usa *blocchi senza spanning* e *allocazione contigua*, viene naturale accedere a ogni record a partire dalla sua **posizione** nel file. Se i record del file sono numerati 0, 1, 2, ..., $r - 1$ e i record in ogni blocco sono numerati 0, 1, ..., $bfr - 1$, dove bfr è il fattore di blocco, allora l' i -esimo record del file è posto nel blocco $[(i/bfr)]$ ed è l' $(i \bmod bfr)$ -esimo record di quel blocco. Un file di questo tipo viene spesso chiamato **file relativo** o **diretto** perché si può agevolmente accedere direttamente ai record a partire dalle loro posizioni relative. L'accesso a un record a partire dalla sua posizione non aiuta a localizzare un record basandosi su una condizione di ricerca, però facilita la costituzione di percorsi di accesso al file.

File di record ordinati (file sorted)

È possibile ordinare fisicamente i record di un file su disco basandosi sui valori di uno dei loro campi, detto **campo di ordinamento**. Ciò porta a un **file ordinato**. Se il campo di riordinamento è anche un **campo chiave** del file – un campo per cui è garantita l'esistenza di un valore univoco in ogni record – allora esso è detto **chiave di ordinamento** del file. In Figura 2.7 è mostrato un file ordinato con NOME come campo chiave di ordinamento (supponendo che gli impiegati abbiano nomi distinti).

I file di record ordinati presentano alcuni vantaggi sui file di record non ordinati. In primo luogo, la lettura dei record secondo l'ordine dei valori della chiave di ordinamento diventa estremamente efficiente, perché non è richiesta alcuna azione di riordinamento. In secondo luogo, trovare il record successivo a quello corrente, secondo l'ordine della chiave di ordinamento, di solito non richiede accessi aggiuntivi ai blocchi, perché il record successivo è nello stesso blocco di quello corrente (a meno che il record corrente non sia l'ultimo del blocco). Infine, l'uso di una condizione di ricerca basata sul valore di un campo chiave di ordinamento ha come risultato un accesso più veloce quando viene usata la tecnica di ricerca binaria, che costituisce un miglioramento rispetto alle ricerche lineari, anche se non è usata spesso per i file su disco.

Una **ricerca binaria** per file su disco può essere fatta sui blocchi anziché sui record. Si supponga che il file abbia b blocchi numerati 1, 2, ..., b , e che i record siano ordinati per valore crescente del loro campo chiave di ordinamento e che si stia cercando un record il cui valore di campo chiave di ordinamento si K . Supponendo che gli indirizzi su disco dei blocchi del file siano disponibili nell'header del file, la ricerca binaria può essere descritta tramite l'Algoritmo 2.1. Una ricerca binaria di solito accede a $\log_2(b)$ blocchi, che il record sia trovato

	NOME	SSN	DATA_NASCITA	LAVORO	STIPENDIO	SESSO
blocco 1	Aaron, Ed					
	Abbott, Diana					
		⋮				
	Acosta, Marc					
blocco 2	Adams, John					
	Adams, Robin					
		⋮				
	Akers, Jan					
blocco 3	Alexander, Ed					
	Alfred, Bob					
		⋮				
	Allen, Sam					
blocco 4	Allen, Troy					
	Anders, Keith					
		⋮				
	Anderson, Rob					
blocco 5	Anderson, Zach					
	Angeli, Joe					
		⋮				
	Archer, Sue					
blocco 6	Arnold, Mack					
	Arnold, Steven					
		⋮				
	Atkins, Timothy					
	⋮					
blocco n - 1	Wong, James					
	Wood, Donald					
		⋮				
	Woods, Menny					
blocco n	Wright, Pam					
	Wyatt, Charles					
		⋮				
	Zimmer, Byron					

Figura 2.7 – Alcuni blocchi di un file ordinato di record IMPIEGATO, con NOME come campo chiave di ordinamento.

o meno – un miglioramento rispetto alle ricerche lineari, dove, in media, si accede a $(b/2)$ blocchi quando il record viene trovato e a b blocchi quando non viene trovato. Un criterio di ricerca che coinvolge le condizioni $>$, $<$, \geq e \leq sul campo di ordinamento è abbastanza efficiente, dal momento che l'ordinamento fisico dei record implica che tutti i record che soddisfano la condizione siano contigui nel file. Per esempio, relativamente alla Figura 2.7, se il criterio di ricerca è (*NOME < 'G'*) – dove *<* significa *alfabeticamente precedente* – i record che soddisfano il criterio di ricerca sono quelli che vanno dall'inizio del file fino al primo record che ha un valore *NOME* che comincia con la lettera G.

L'ordinamento non fornisce alcun vantaggio per un accesso casuale ed ordinato ai record basato sui valori degli altri campi *non di ordinamento del file*. In questi casi per l'accesso casuale si fa una ricerca lineare. Per accedere ai record in un ordine basato su un campo non di ordinamento è necessario creare un'altra copia ordinata – in ordine diverso – del file.

Algoritmo 2.1 – Ricerca binaria basata su una chiave di ordinamento di un file su disco.

```

l ← 1; u ← b; (*b è il numero di blocchi del file*)
while (u ≥ 1) do
    begin i ← (l + u) div 2;
    trasferisci il blocco i del file nel buffer;
    if K < (valore del campo chiave di ordinamento del PRIMO record
    nel blocco i)
    then u ← i - 1
    else if K > (valore del campo chiave di ordinamento dell'ULTIMO
    record nel blocco i)
    then l ← i + 1
    else if il record con valore del campo chiave di
    ordinamento = K è nel buffer
        then goto trovato
        else goto nontrovato;
    end;
    goto nontrovato;

```

Per un file ordinato l'inserimento e la cancellazione di record sono operazioni dispendiose perché i record devono rimanere fisicamente ordinati. Per inserire un record si deve trovare la sua posizione corretta nel file, basandosi sul valore del suo campo di ordinamento, e quindi far spazio nel file per inserire il record in quella posizione. Per un file di grosse dimensioni ciò può essere molto dispendioso in termini di tempo perché, in media, metà dei record del file devono essere spostati per fare spazio ai nuovi record. Ciò significa che metà dei blocchi del file devono essere letti e riscritti dopo che i record vengono spostati fra di loro. Per la

cancellazione di un record il problema è meno grave se vengono usati gli indicatori di cancellazione e una riorganizzazione periodica.

Un'opzione per rendere più efficiente l'inserimento consiste nel tenere in ogni blocco un certo spazio inutilizzato per nuovi record. Comunque, una volta che questo spazio è stato utilizzato, il problema si ripresenta. Un altro metodo frequentemente usato consiste nel creare un file *non ordinato* temporaneo detto file di **overflow** o di **transazione**. Con questa tecnica il file ordinato vero e proprio è detto file **principale** o file **master**. I nuovi record vengono inseriti alla fine del file di overflow anziché nella loro posizione corretta nel file principale. Periodicamente, durante la riorganizzazione del file, il file di overflow viene ordinato e quindi fuso con il file master. L'inserimento diviene molto efficiente, ma al costo di una complessità aggiuntiva nell'algoritmo di ricerca. Il file di overflow deve essere ispezionato usando una ricerca lineare se, dopo la ricerca binaria, il record non è stato trovato nel file principale. Per applicazioni che non richiedono le informazioni più recenti, nel corso di una ricerca i record di overflow possono essere ignorati.

La modifica del valore del campo di un record dipende da due fattori: la condizione di ricerca per localizzare il record e il campo che deve essere modificato. Se la condizione di ricerca coinvolge il campo chiave di ordinamento, allora si può localizzare il record usando una ricerca binaria, altrimenti occorre effettuare una ricerca lineare. Un campo non di ordinamento può essere modificato cambiando il record e riscrivendolo nella stessa locazione fisica su disco – supponendo di avere record a lunghezza fissa. La modifica del campo di ordinamento implica che il record possa cambiare la sua posizione nel file, il che richiederebbe la cancellazione del record vecchio seguita dall'inserimento del record modificato.

Leggere i record del file secondo l'ordine del campo di ordinamento è abbastanza efficiente se si trascurano i record in overflow, dal momento che i blocchi possono essere letti consecutivamente usando la doppia bufferizzazione. Per includere i record in overflow, bisogna inserirli nelle loro posizioni corrette; in questo caso si può dapprima riorganizzare il file, e poi leggere i suoi blocchi sequenzialmente. Per riorganizzare il file, prima di tutto si ordinano i record nel file di overflow, e poi li si fonde con il file master. Durante la riorganizzazione i record segnati per la cancellazione vengono rimossi.

I file ordinati sono usati raramente nelle applicazioni di basi di dati se non è usato un cammino di accesso aggiuntivo, detto **indice primario**; ciò ha come risultato un **file indicizzato sequenziale**. Questo migliora ulteriormente il tempo di accesso causale sul campo chiave di ordinamento.

Tipo di organizzazione	Metodo di accesso/ricerca	Tempo medio di accesso a un record specifico
Heap (non ordinata)	Scansione sequenziale (ricerca lineare)	$b/2$
Ordinata	Scansione sequenziale	$b/2$
Ordinata	Ricerca binaria	$\log_2 b$

Tabella 2 – Tempi medi di accesso.

Tecniche hash

Un altro tipo di organizzazione primaria di file è basato sull'hash, che fornisce un accesso molto rapido ai record sotto certe condizioni di ricerca. Questa organizzazione è detta di solito **file hash**. La condizione di ricerca deve essere una condizione di uguaglianza su un campo singolo, detto **campo hash** del file. Di solito il campo hash è anche un campo chiave del file; in questo caso è detto **chiave hash**. L'idea che sta dietro l'hash è quella di fornire una funzione h , detta **funzione hash** o **funzione di randomizzazione**, che è applicata al valore del campo hash di un record e fornisce l'*indirizzo* del blocco di disco in cui è memorizzato il record. Una ricerca del record all'interno del blocco può essere effettuata in un buffer in memoria centrale. Per la maggior parte dei record si ha bisogno solo di un accesso a un singolo blocco per recuperare quel record. L'hash è usato anche come una struttura di ricerca interna in un programma, ogni volta che a un gruppo di record si accede esclusivamente usando il valore di un campo.

Hash interno

Per file interni, l'hash è tipicamente implementato con una **tavella hash** costruita usando un vettore di record. Si supponga che il campo di valori possibili per l'indice del vettore vada da 0 a $M - 1$ (Figura 2.8a); si ha pertanto M slot i cui indirizzi corrispondono agli indici del vettore. Si sceglierà allora una funzione hash che trasforma il valore del campo hash in un intero compreso tra 0 e $M - 1$. Una funzione hash comune è la funzione $h(K) = K \bmod M$, che fornisce il resto di un valore interno di campo hash K dopo la divisione per M ; questo valore viene quindi usato per l'indirizzo del record.

Valori non interi del campo hash possono essere trasformati in valori interi prima che venga applicata la funzione mod. Per stringhe di caratteri possono essere usati nella trasformazione i codici numerici (ASCII) associati ai caratteri.

Figura 2.8 – Strutture dati per l'hash interno. (a) Vettore di M posizioni usato nell'hash interno. (b) Risoluzione delle collisioni tramite concatenazione di record.

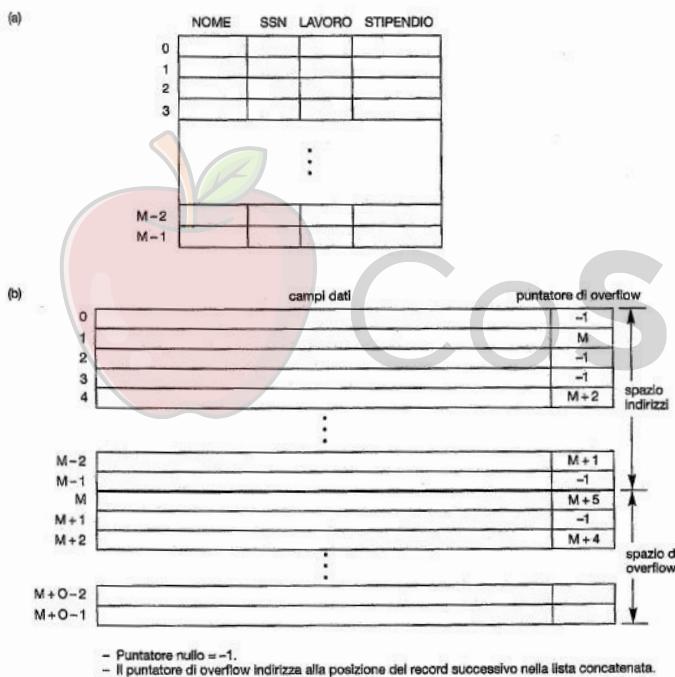


Figura 2.8 – Strutture dati per l'hash interno. (a) Vettore di M posizioni usato nell'hash interno. (b) Risoluzione delle collisioni tramite concatenazione di record.

Possono essere usate anche altre funzioni hash. Una tecnica, detta **folding**, prevede di applicare una funzione aritmetica come l'*addizione* o una funzione logica come l'*or esclusivo* a parti diverse del valore del campo hash per calcolare l'indirizzo hash. Un'altra tecnica prevede la raccolta di alcune cifre del valore del campo hash – ad esempio la terza, la quinta, l'ottava cifra – per formare l'indirizzo hash. Il problema con la maggior parte delle funzioni hash è che esse non garantiscono che valori distinti saranno trasformati in indirizzi distinti, perché lo **spazio del campo hash** – il numero di valori possibili che un campo hash può assumere – è di solito molto più grande dello **spazio indirizzi** – il numero di indirizzi disponibili per i record. La funzione hash mappa lo spazio del campo hash nello spazio indirizzi.

Una **collisione** si verifica quando il valore del campo hash di un record che sta per essere inserito è trasformato da una funzione hash in un indirizzo che contiene già un record diverso. In questa situazione occorre inserire il nuovo record in un'altra posizione, dal momento che il suo indirizzo hash è occupato. Il processo di ricerca di un'altra posizione è detto **risoluzione delle collisioni**. Ci sono molti metodi per la risoluzione delle collisioni, fra cui quelli qui di seguito ricordati.

- *Indirizzamento aperto*: procedendo dalla posizione occupata specificata dall'indirizzo hash, il programma verifica in ordine le posizioni successive fino a che non si trova una posizione inutilizzata (vuota).
- *Concatenamento*: per questo metodo si tengono varie locazioni di overflow, di solito estendendo il vettore con un certo numero di posizioni di overflow. Inoltre, a ogni locazione di record viene aggiunto un campo puntatore. Una collisione viene risolta ponendo il nuovo record in una locazione di overflow inutilizzata e imponendo il valore dell'indirizzo di tale locazione al puntatore presente nella locazione di indirizzo hash occupata. Si mantiene perciò una lista concatenata di record di overflow per ogni indirizzo hash, come mostrato in Figura 2.8(b).
- *Hash multiplo*: il programma applica una seconda funzione hash se la prima ha come risultato una collisione. Se ne risulta un'altra collisione, il programma usa l'indirizzamento aperto o applica una terza funzione hash e poi, se necessario, usa l'indirizzamento aperto.

Ogni metodo di risoluzione delle collisioni richiede il proprio algoritmo per l'inserimento, il recupero e la cancellazione di record. Gli algoritmi per il concatenamento sono i più semplici; gli algoritmi di cancellazione per l'indirizzamento aperto sono piuttosto complicati.

Lo scopo di una buona funzione hash è quello di distribuire i record uniformemente sullo spazio di indirizzi, in modo da minimizzare le collisioni senza lasciare molte locazioni inutilizzate. Studi di simulazione e di analisi hanno mostrato che di solito è meglio tenere una tabella hash piena tra il 70% e il 90%; in questo modo il numero di collisioni rimane basso e allo stesso tempo non si spreca troppo spazio. Perciò se si prevede di avere r record da memorizzare nella tabella, si dovranno scegliere M locazioni per lo spazio di indirizzi in modo tale che (r/M) sia compreso tra 0,7 e 0,9. Può essere anche utile scegliere un numero primo per M , dal momento che è stato dimostrato che ciò distribuisce meglio gli indirizzi hash sullo spazio di indirizzi quando viene usata la funzione mod hash. Altre funzioni hash possono invece richiedere che M sia una potenza di 2.

Hash esterno per file su disco

L'hash per file su disco è detto **hash esterno**. Per adattarsi alle caratteristiche della memorizzazione su disco, lo spazio indirizzi obiettivo è fatto di **bucket**, ciascuno dei quali contiene più record. Un bucket è un blocco di disco o un cluster di blocchi contigui. La funzione hash mappa una chiave in un numero relativo per il bucket, piuttosto di assegnare al bucket un indirizzo di blocco assoluto. Una tabella contenuta nell'header del file converte il numero del bucket nel corrispondente indirizzo di blocco di disco, come illustrato in Figura 2.9.

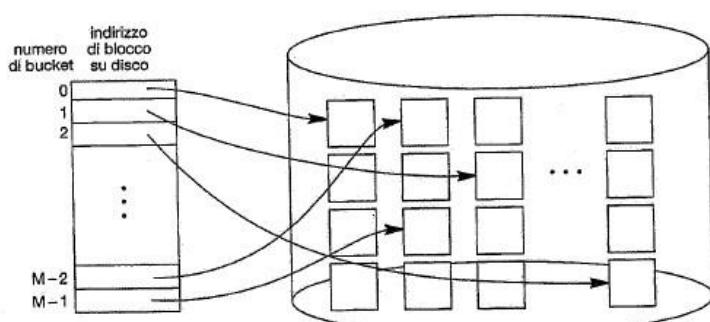


Figura 2.9 - Accoppiamento tra numeri di bucket e indirizzi di blocchi di disco.

con la concatenazione di record di overflow per il bucket, come mostrato in Figura 2.10. I puntatori nella lista

Con i bucket il problema delle collisioni è meno grave perché possono essere inviati nello stesso bucket senza causare problemi tanti record quanti ce ne stanno. Però occorre anche provvedere al caso in cui un bucket sia riempito fino al limite e un nuovo record che sta per essere inserito venga inviato dalla funzione hash in quel bucket. È possibile usare una variazione del concatenamento in cui in ogni bucket si mantiene un puntatore a una lista

concatenata dovrebbero essere **puntatori a record**, che comprendono sia un indirizzo di blocco sia una posizione relativa del record nel blocco.

L'hash fornisce il più rapido accesso possibile per il recupero di un record arbitrario dato il valore del suo campo hash. Anche se la maggior parte delle buone funzioni hash non mantengono i record nell'ordine fissato dai valori del campo chiave, ci sono alcune funzioni – dette **order preserving** (preservano l'ordine) – che lo fanno. Un semplice esempio di funzione hash order preserving è fornito dalla funzione hash che prende come indirizzo hash le tre cifre più a sinistra di un campo numero di fattura, e che tiene in ogni bucket i record ordinati secondo il numero di fattura.

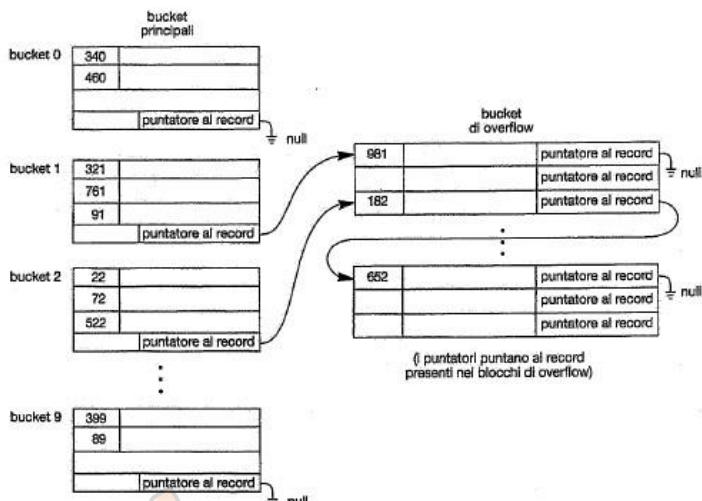


Figura 2.10 - Gestione tramite concatenamento dell'overflow dei bucket.

Un altro esempio consiste nell'usare una chiave hash intera direttamente come indice per un file relativo, se i valori della chiave hash riempiono un particolare intervallo; ad esempio, se i numeri di impiegato in un'azienda vengono assegnati come 1, 2, 3, Fino al numero totale di impiegati, si può usare la funzione hash identità che mantiene l'ordine. Purtroppo, ciò funziona solo se le chiavi sono generate in ordine da una certa applicazione.

Lo schema hash descritto è detto **hash statico** perché viene allocato un numero fisso M di bucket. Ciò può costituire un serio inconveniente per file dinamici. Si supponga di allocare M bucket per lo spazio di indirizzi, e sia m il numero massimo di record che possono trovar posto in un bucket; perciò nello spazio allocato troveranno posto al più $(m * M)$ record. Se succede che il numero di record è notevolmente minore di $(m * M)$, rimane molto spazio inutilizzato. D'altro lato, se il numero di record aumenta fino a molto più di $(m * M)$, si verificheranno molte collisioni e il recupero sarà rallentato a causa delle lunghe liste di record di overflow. In entrambi i casi ci si potrebbe trovare nella situazione di dover cambiare il numero di blocchi allocati M e quindi usare una nuova funzione hash (basata sul nuovo valore di M) per ridistribuire i record. Queste riorganizzazioni possono essere piuttosto dispendiose in termini di tempo per file di grandi dimensioni. Organizzazioni più recenti per file dinamici basate sull'hash consentono al numero di bucket di variare dinamicamente, limitandosi a una riorganizzazione localizzata.

Quando si usa l'hash esterno, la ricerca di un record, dato il valore di un certo campo diverso dal campo hash, è tanto dispendiosa quanto nel caso di un file non ordinato. La cancellazione di record può essere implementata rimuovendo il record dal suo bucket. Se il bucket ha una catena di overflow, si può spostare uno dei record di overflow nel bucket per sostituire il record cancellato. Se il record che deve essere cancellato è già in overflow, semplicemente lo si rimuove dalla lista concatenata. Si noti che la rimozione di un record in overflow implica di dover tenere ciccia delle posizioni di overflow vuote. Ciò può essere fatto facilmente mantenendo una lista concatenata di locazioni di overflow inutilizzate.

La modifica del valore di campo di un record dipende da due fattori: la condizione di ricerca per localizzare il record e il campo che deve essere modificato. Se la condizione di ricerca è un confronto di uguaglianza sul campo hash, è possibile localizzare il record efficientemente usando la funzione hash, altrimenti occorre fare

una ricerca lineare. Un campo non hash può essere modificato cambiando il record e riscrivendolo nello stesso bucket. La modifica del campo hash implica che il record possa spostarsi in un altro bucket, il che richiede la cancellazione del record vecchio seguita dall'inserimento del record modificato.

Tecniche di hashing per l'espansione dinamica dei file

Un grave inconveniente dello schema hash *statico* esaminato finora è che lo spazio indirizza hash è fisso, e quindi è difficile espandere o ridurre il file dinamicamente. Gli schemi descritti di seguito tentano di porre rimedio a questa situazione. Il primo schema – **hash estendibile** – memorizza una struttura d'accessi insieme al file, e perciò per certi versi simile all'indicizzazione. La differenza principale è che la struttura d'accesso è basata sui valori che risultano dopo l'applicazione della funzione hash al campo di ricerca. Nell'indicizzazione, invece, la struttura d'accesso è basata sui valori del campo di ricerca stesso. La seconda tecnica, detta **hash lineare**, non richiede strutture d'accesso addizionali.

Questi schemi hash traggono profitto dal fatto che il risultato dell'applicazione di una funzione hash è un intero non negativo, e perciò può essere rappresentato come un numero binario. La struttura d'accesso è costruita sulla **rappresentazione binaria** del risultato della funzione hash, che è una stringa di **bit**. Tale rappresentazione sarà detta **valore hash** di un record. I record sono distribuiti fra i bucket sulla base dei valori dei *bit iniziali* (*leading bits*) presenti nei loro valori hash.

Hash estendibile. Nell'hash estendibile si mantiene una specie di **directory** – un vettore di 2^d indirizzi di bucket, dove d è detta **profondità globale (global depth)** della directory. Il valore intero corrispondente ai primi d bit (i più significativi) di un valore hash è usato come un indice per il vettore per determinare un elemento della directory, e l'indirizzo in corrispondenza a quell'elemento determina il bucket in cui sono memorizzati i record corrispondenti. Però non ci deve necessariamente essere un bucket distinto per ciascuna delle 2^d locazioni della directory. Molte locazioni della directory con gli stessi d' primi bit per i loro valori hash possono contenere lo stesso indirizzo di bucket, se tutti i record che vengono inviati dalla funzione hash in queste locazioni trovano posto in un bucket singolo. Una **profondità locale (local depth) d'** – memorizzata con ogni bucket – specifica il numero di bit su cui è basato il contenuto del bucket. In Figura 2.11 viene mostrata una directory con profondità globale $d = 3$. Il valore di d può essere incrementato o decrementato di un'unità alla volta, raddoppiando o dimezzando così il numero di elementi nel vettore directory. Il raddoppio è necessario se un bucket, la cui profondità locale d' è uguale alla profondità globale d , va in overflow. Il dimezzamento si verifica se $d > d'$ per tutti i bucket dopo che si verifica qualche cancellazione. La maggior parte dei recuperi di record richiede due accessi ai blocchi, uno alla directory e

l'altro al bucket.

Per illustrare lo sdoppiamento dei bucket, si supponga che l'inserimento di un nuovo record causi un overflow nel bucket i cui valori hash cominciano con 01 – il terzo bucket in Figura 2.11. I record saranno distribuiti tra due bucket: il primo contiene tutti i record i cui valori hash cominciano con 010, il secondo tutti quelli i cui valori hash cominciano con 011. Ora le due locazioni della directory per 010 e 011 puntano ai due nuovi bucket distinti. Prima dello sdoppiamento esse puntavano allo stesso bucket. La profondità locale d' dei due nuovi bucket è 3, che è di un'unità superiore alla profondità locale del bucket vecchio.

Se un bucket che va in overflow e deve essere sdoppiato, aveva originariamente una profondità

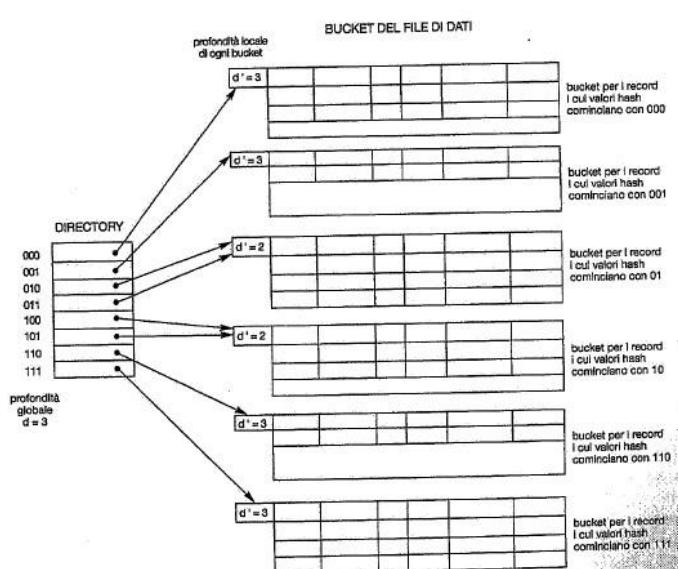


Figura 2.11 - Struttura dello schema per hash estendibile.

locale d' uguale alla profondità globale d della directory, allora la dimensione della directory deve essere raddoppiata; così si può usare un bit aggiuntivo per distinguere fra loro i due nuovi bucket. Ad esempio, se, con riferimento alla Figura 2.11, il bucket per i record i cui valori hash cominciano con 111 va in overflow, allora i due nuovi bucket hanno bisogno di una directory con profondità globale $d = 4$, perché i due nuovi bucket sono ora etichettati 1110 e 1111, e perciò le loro profondità locali sono entrambe pari a 4. La dimensione della directory è perciò raddoppiata, e anche ognuna delle altre locazioni originali nella directory è suddivisa in due locazioni, ciascuna delle quali ha lo stesso valore di puntatore della locazione originale.

Il principale vantaggio dell'hash estendibile, quello che lo rende attraente, è che le prestazioni del file non degradano man mano che il file cresce di dimensione, al contrario di quanto avviene con l'hash esterno statico, in cui le collisioni aumentano e il corrispondente concatenamento causa accessi aggiuntivi. Inoltre, nell'hash estendibile non viene allocato nessuno spazio per una crescita futura, ma bucket aggiuntivi possono essere allocati dinamicamente quando necessario. Lo spazio in più per la tabella di directory è trascurabile. La dimensione massima della directory è 2^k , dove k è il numero di bit nel valore hash. Un altro vantaggio è che lo sdoppiamento causa in molti casi una minore riorganizzazione, dal momento che solo i record in un bucket vengono ridistribuiti sui due nuovi bucket. L'unica occasione in cui una riorganizzazione è più dispendiosa si presenta quando la directory deve essere raddoppiata (o dimezzata). Uno svantaggio è che la directory deve essere ispezionata prima di accedere ai bucket veri e propri, il che ha come risultati due accessi ai blocchi anziché uno solo, com'è invece nell'hash statico. Questa penalizzazione nelle prestazioni è considerata di minore importanza e perciò lo schema è giudicato piuttosto indicato per file dinamici.

Hash lineare. L'idea alla base dell'hash lineare è quella di consentire a un file hash di espandere e ridurre dinamicamente il suo numero di bucket senza aver bisogno di una directory. Si supponga che il file abbia inizialmente M bucket numerati 0, 1, ..., $M - 1$ e usi la funzione mod hash $h(K) = K \bmod M$; questa funzione hash è detta funzione hash iniziale h_i . L'overflow a seguito di collisioni è ancora necessario e può essere gestito mantenendo singole catene di overflow per ogni bucket. Però, quando una collisione porta a un record di overflow in *ogni* bucket del file, il *primo* bucket nel file – bucket 0 – viene sdoppiato in due bucket: il bucket originario 0 e un nuovo bucket M alla fine del file. I record originariamente nel bucket 0 sono distribuiti tra i due bucket sulla base di una diversa funzione hash $h_{i+1}(K) = K \bmod 2M$. Una proprietà fondamentale delle due funzioni hash h_i e h_{i+1} è che ogni record che era inviato al bucket 0 sulla base di h_i sarà inviato o al bucket 0 o al bucket M sulla base di h_{i+1} ; ciò è necessario per l'hash lineare funzioni.

Quando ulteriori collisioni portano a record di overflow, vengono sdoppiati altri bucket nell'ordine *lineare* 1, 2, 3, Se si verificano abbastanza overflow, saranno stati sdoppiati tutti i bucket originari del file, 0, 1, 2, ..., $M - 1$, cosicché ora il file avrà $2M$ bucket anziché M , e tutti i bucket useranno la stessa funzione hash h_{i+1} . Perciò i record in overflow sono alla fine ridistribuiti in bucket regolari, usando la funzione h_{i+1} attraverso uno *sdoppiamento differito* dei loro bucket. Non c'è alcuna directory; è necessario solo un valore n , inizialmente posto a 0 e incrementato di 1 ogni volta che si verifica uno sdoppiamento, per determinare quali bucket sono stati sdoppiati. Per recuperare un record con valore di chiave hash K , dapprima si applica la funzione h_i a K ; se $h_i(K) < n$, allora si applica la funzione h_{i+1} a K perché il bucket è già stato sdoppiato. Inizialmente è $n = 0$, a indicare che la funzione h_i si applica a tutti i bucket; n cresce linearmente quando i bucket vengono sdoppiati.

Quando $n = M$ dopo essere stato incrementato, ciò significa che tutti i bucket originari sono stati sdoppiati e la funzione hash h_{i+1} si applica a tutti i record nel file. A questo punto, n è riportato a 0, e ogni nuova collisione che causa overflow porta all'uso di una nuova funzione hash $h_{i+2}(K) = K \bmod 4M$. In generale si usa una sequenza di funzioni hash $h_{i+j}(K) = K \bmod (2^j M)$, dove $j = 0, 1, 2, \dots$; è necessaria una nuova funzione hash h_{i+j+1} ogni volta che tutti i bucket 0, 1, ..., $(2^j M) - 1$ sono stati sdoppiati e n è riportato a 0.

Tecnologia RAID

Con la crescita esponenziale delle prestazioni e delle capacità dei dispositivi e delle memorie a semiconduttore, si stanno diffondendo microprocessori più veloci con memorie principali sempre più grandi. Per far fronte a questa crescita, è naturale aspettarsi che anche la tecnologia della memoria secondaria tenti di adeguarsi, per prestazioni e affidabilità, alla tecnologia dei processori.

Un miglioramento importante nella tecnologia della memoria secondaria è rappresentato dallo sviluppo della tecnologia **RAID**, acronimo di Redundant Array of Inexpensive (o Independent) Disks (vettori ridondanti di dischi economici o indipendenti). L'idea di RAID ha ricevuto un'adesione molto positiva dall'industria e è stata sviluppata in un insieme variegato di architetture RAID alternative (7 livelli RAID da 0 a 6).

Lo scopo principale della tecnologia RAID è quello di uguagliare i tassi molto diversi di miglioramento delle prestazioni dei dischi rispetto a quelli della memoria principale e dei microprocessori. Mentre le capacità della RAM sono quadruplicate ogni due o tre anni, i *tempi di accesso* al disco stanno migliorando meno del 10% all'anno, e i *tassi di trasferimento* stanno aumentando approssimativamente del 2% annuo. In verità le *capacità* dei dischi stanno aumentando più del 50% all'anno, ma i miglioramenti nella velocità e nel tempo di accesso sono di entità molto inferiore. In Tabella 3 sono illustrate le tendenze nella tecnologia dei dischi in termini dei valori dei parametri del 2003 e dei tassi di miglioramento.

	Valori dei parametri nel 1993	Tasso storico di miglioramento annuo (%)	Valori nel 2003
Densità areale	50-150 Mbit/pollice quadrato	27	36 Gbit/pollice quadrato
Densità lineare	40.000-60.000 bit/pollice	13	570.000 bit/pollice
Densità inter-traccia	1.500-3.000 tracce/pollice	10	64.000 tracce/pollice
Capacità (fattore di forma 3,5")	100-2.000 MB	27	146 GB
Tasso di trasferimento	3-4 MB/s	22	43-78 MB/sec
Tempo di posizionamento	7-20 ms	8	3,5-6 ms

Tabella 3 - Tendenze nella tecnologia dei dischi.

Esiste poi una seconda differenza qualitativa tra la capacità di microprocessori speciali che si rivolgono a nuove applicazioni che riguardano l'elaborazione di dati video, audio, immagini e spaziali e la corrispondente mancanza di accesso rapido a grandi insiemi di dati condivisi.

La soluzione naturale consiste in un grande vettore di piccoli dischi indipendenti che si comporta come un singolo disco logico di maggiori prestazioni. Viene usato un concetto detto **data striping** (suddivisione dei dati), che

utilizza il *parallelismo* per incrementare le prestazioni del disco. Il data striping distribuisce i dati in modo trasparente fra più dischi, così che essi si comportino come un unico disco, grande e veloce. In Figura 2.12 è rappresentato un file distribuito o suddiviso (*striped*) fra quattro dischi. La suddivisione migliora le prestazioni di I/O complessive consentendo che I/O multipli vengano serviti in parallelo, fornendo così alti tassi di trasferimento complessivi. La suddivisione dei dati realizza anche un bilanciamento del carico fra i dischi. Inoltre, memorizzando informazioni ridondanti sui dischi con l'uso della parità o di qualche altro codice a correzione d'errore, può essere migliorata l'affidabilità.

Miglioramento dell'affidabilità

Per un vettore di n dischi la frequenza di un guasto è n volte quella che si ha per un solo disco. Perciò, se l'MTTF (Mean Time To Failure: tempo medio perché si verifichi un guasto) di un'unità disco è considerato di 200.000 ore, ossia circa 22,8 anni (tempi tipici vanno fino a circa 1 milione di ore), quello di un banco di 100 unità disco diventa solo di 2000 ore, ossia 83,3 giorni. Mantenere una sola copia di dati in un tale vettore di dischi causerà una significativa perdita di affidabilità. Un'ovvia soluzione è quella di servirsi di una certa ridondanza dei dati in modo tale che i guasti ai dischi possano essere tollerati. Gli svantaggi sono molti:

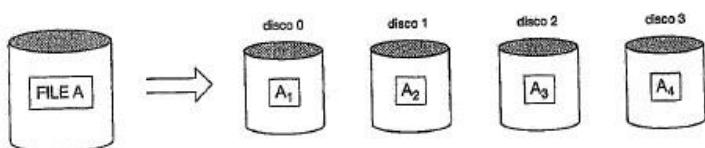


Figura 2.12 - Data striping. Il file A è suddiviso (striped) su quattro dischi.

operazioni di I/O aggiuntive per la scrittura, calcoli supplementari per mantenere la ridondanza e per effettuare il ripristino dagli errori, e capacità di disco aggiuntiva per memorizzare informazioni ridondanti.

Una tecnica per introdurre ridondanza è detta **mirroring** o **shadowing**. I dati sono scritti con ridondanza in due dischi fisici identici, che sono trattati come un solo disco logico. Quando i dati vengono letti, essi possono essere recuperati dal disco con minori ritardi di accodamento, di posizionamento e di rotazione. Se un disco si guasta, l'altro disco viene usato finché il primo non è stato riparato. Si supponga che il tempo medio per la riparazione sia di 24 ore; allora il tempo medio per la perdita di dati in un sistema a disco riflesso (*mirrored disk system*) che usa 100 dischi con MTTF di 200.000 ore ciascuno è di $(200.000)^3/(2 * 24) = 8,33 * 10^8$ ore, cioè di 95.028 anni. La riflessione di disco raddoppia anche il tasso con il quale sono gestite le richieste di lettura, dal momento che una lettura può essere indirizzata all'uno o all'altro disco. Il tasso di trasferimento di ciascuna lettura, però, rimane uguale a quello che si ha per un singolo disco.

Un'altra soluzione al problema dell'affidabilità è quella di memorizzare informazioni supplementari che non sono normalmente necessarie ma che possono essere usate per ricostruire le informazioni perse in caso di guasto del disco. L'introduzione di ridondanze deve considerare due problemi: la scelta di una tecnica per calcolare l'informazione ridondante e la scelta di un metodo per distribuire l'informazione ridondante sul vettore di dischi. Si affronta il primo problema tramite l'uso di *codici a correzione d'errore* che utilizzano bit di parità, o codici specializzati come i codici Hamming. Con lo schema di parità si può considerare che un disco ridondante contenga la somma di tutti i dati presenti negli altri dischi. Quando un disco si guasta le informazioni mancanti possono essere ricostruite tramite un processo simile a una sottrazione.

Per il secondo problema, i due approcci più importanti consistono nel memorizzare le informazioni ridondanti su un piccolo numero di dischi o nel distribuirle uniformemente su tutti i dischi. Quest'ultimo ha come risultato un miglior bilanciamento del carico. I diversi livelli di RAID scelgono una combinazione di queste opzioni per implementare la ridondanza, e perciò per migliorare l'affidabilità.

Miglioramento delle prestazioni

I vettori di dischi si servono della tecnica del data striping per ottenere tassi di trasferimento migliori. Si noti che i dati possono essere letti o scritti solo un blocco alla volta, cosicché un tipico trasferimento comprende 512 byte. La suddivisione su disco può essere eseguita a un livello di granularità più fine, scomponendo un byte di dati in bit e distribuendo i bit su dischi diversi. Perciò il **data striping a livello di bit (bit-level data striping)** consiste nello spezzare un byte di dati e nello scrivere il bit j nel j -esimo disco. Con byte di 8 bit, otto dischi fisici possono essere considerati come un solo disco logico con un aumento di otto volte del tasso di trasferimento dati. Ogni disco partecipa a ciascuna richiesta di I/O e l'ammontare totale di dati letti per ogni richiesta è di otto volte i dati letti dal singolo disco. Lo striping a livello di bit può essere generalizzato a un numero di dischi che sia o un multiplo o un fattore di otto. Perciò in un vettore di quattro dischi il bit n va nel disco $(n \bmod 4)$.

La granularità dell'interleaving dei dati può essere maggiore di un singolo bit; ad esempio, i blocchi di un file possono essere suddivisi su dischi diversi, dando origine allo **striping a livello di blocco (block-level striping)**. In Figura 2.12 è mostrato il data striping a livello di blocco supponendo che il file di dati contenga quattro blocchi. Con lo striping a livello di blocco, richieste indipendenti multiple che accedono a blocchi singoli (piccole richieste) possono essere servite in parallelo da dischi separati, diminuendo così il tempo di accodamento delle richieste di I/O. Le richieste che accedono a più blocchi (grandi richieste) possono essere parallele, riducendo così il loro tempo di risposta. In generale, più elevato è il numero di dischi in un vettore, più ne beneficiano le prestazioni potenziali. Però, supponendo che i guasti siano indipendenti, il vettore di dischi di 100 dischi ha nell'insieme un'affidabilità che è 1/100 di quella di un disco singolo. Perciò la ridondanza tramite codici a correzione d'errore e il mirroring di disco è necessaria per fornire affidabilità insieme con alte prestazioni.

Organizzazioni e livelli RAID

Sono state definite diverse organizzazioni RAID, basate su differenti combinazioni dei due fattori granularità dell'interleaving dei dati (*striping*) e modello usato per calcolare informazioni ridondanti. Nella proposta

iniziale sono stati suggeriti i livelli di RAID da 1 a 5, mentre due livelli aggiuntivi – 0 e 6 – sono stati aggiunti in seguito.

Il livello RAID 0 non presenta dati ridondanti, e perciò ha le migliori prestazioni in scrittura, dal momento che gli aggiornamenti non devono essere duplicati. Però le sue prestazioni in lettura non sono così buone come per il livello RAID 1, che usa dischi riflessi. In quest'ultimo è possibile un incremento di prestazioni, pianificando una richiesta di lettura al disco che presenta il minore ritardo previsto di posizionamento e di rotazione. Il livello RAID 2 usa una ridondanza analoga a quella usata in memoria di lavoro tramite l'uso di codici Hamming, che contengono bit di parità per diversi sottoinsieme sovrapposti di componenti. Perciò, in una versione particolare di questo livello, tre dischi ridondanti sono sufficienti per quattro dischi originali, mentre con il mirroring – come nel livello 1 – ne sarebbero stati richiesti quattro. Il livello 2 prevede sia la rivelazione sia la correzione d'errore, anche se la rivelazione non è in generale richiesta dato che i dischi rotti sono facilmente identificabili.

Il livello RAID 3 usa un solo disco di parità, facendo affidamento sul disk controller per riuscire a capire quale disco si è guastato. I livelli 4 e 5 usano il data striping a livello di blocco, con il livello 5 che distribuisce dati e informazioni di parità su tutti i dischi. Infine, il livello RAID 6 utilizza il cosiddetto schema di ridondanza $P + Q$ che usa i codici di Reed e Solomon per proteggere fino a due guasti di disco usando solo due dischi ridondanti. I sette livelli RAID (da 0 a 6) sono illustrati in Figura 2.13.

La ricostruzione in casi di guasto di disco è più facile per il livello RAID 1, mentre gli altri livelli richiedono la lettura di più dischi. Il livello 1 è usato per applicazioni critiche, come ad esempio per memorizzare log (registrazioni) di transazioni. I livelli 3 e 5 sono preferiti per la memorizzazioni di grandi quantità di dati, con il livello 3 che consente tassi di trasferimento maggiori. I progettisti di un'organizzazione RAID per una data combinazione applicativa devono confrontare molte decisioni progettuali, come ad esempio il livello di RAID, il numero di dischi, la scelta di schemi di parità e il raggruppamento dei dischi per lo striping a livello di blocco. Dettagliati studi di prestazioni sono stati svolti su piccole letture e scritture (che si riferiscono a richieste di I/O per una sola unità di striping) e grandi letture e scritture (che si riferiscono a richieste di I/O per un'unità di striping da ciascun disco di un gruppo a correzione d'errore).

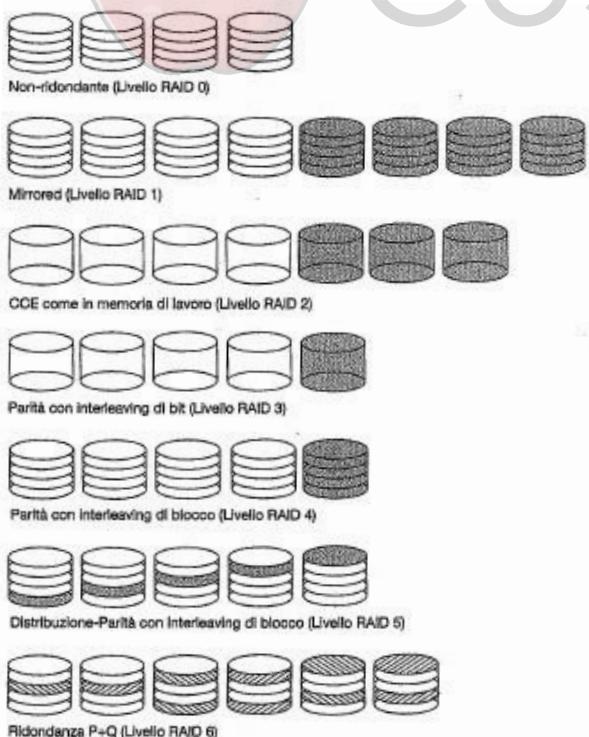


Figura 2.13 – Vari livelli di RAID.

Strutture di indici per i file

In questa parte si darà per scontato che esista già un file con una certa organizzazione primaria con dati non ordinati, ordinati od organizzati secondo una funzione hash come quelle che sono state descritte in precedenza. Si descriveranno altre **strutture ausiliarie di accesso** chiamate **indici**, usate per velocizzare la ricerca dei record in risposta a determinate condizioni di ricerca. Gli indici tipicamente forniscono **percorsi di accesso secondari**, che offrono metodi alterativi per accedere ai record senza influenzare la posizione fisica dei record sul disco. Permettono un accesso efficace sulla base di **campi d'indicizzazione** che vengono utilizzati per costruire l'indice. *Qualsiasi campo* del file può essere usato per creare un indice e sullo stesso file possono essere costruiti *più indici* su campi differenti. È possibile immaginare una varietà di indici, ciascuno dei quali utilizza una particolare struttura dati per velocizzare la ricerca. Per trovare uno o più record nel fine in base a un determinato criterio di selezione su un campo di indicizzazione, inizialmente si deve accedere all'indice per conoscere la posizione di uno o più blocchi nel file in cui si trovano i record richiesti. I tipi di indici principali si basano su file ordinati (indici a un solo livello) o su strutture di dati ad albero (indici a più livelli, alberi B⁺). Gli indici possono essere creati anche tramite funzioni hash oppure altre strutture dati. Successivamente verranno descritti i diversi tipi di indici ordinati a un solo livello: primario, secondario e di cluster. Partendo da un indice a un solo livello memorizzato in un file ordinato, è possibile sviluppare ulteriori indici, dando origine al concetto di indici a più livelli.

Tipi di indici ordinati a un solo livello

L'idea su cui si basa la struttura di accesso degli indici ordinati è simile a quella che sta alla base dell'indice analitico di un libro di testo, il quale riporta in ordine alfabetico concetti fondamentali con indicazione del numero di pagina in cui compaiono. La ricerca nell'indice permette di consultare un elenco di *indirizzi*, in questo caso numeri di pagina, e di utilizzare questi ultimi per individuare un termine nel testo *cercandolo* alle pagine specificate. L'alternativa, se non ci fosse l'indice, sarebbe esaminare lentamente tutto il volume, parole per parola, per trovare il termine a cui si è interessati; quest'operazione corrisponde a eseguire una ricerca lineare su un file. Naturalmente la maggior parte dei libri contiene anche altre informazioni, ad esempio i titoli di capitoli e di paragrafi, che possono aiutare a trovare il termine in questione senza dover consultare tutto il libro. L'indice, però, fornisce l'unica indicazione esatta del punto in cui esso compare.

Per un file con una data struttura di record che consiste di parecchi campi (o attributi), la struttura di accesso a indice di solito è definita su un unico campo, chiamato **campo di indicizzazione** (oppure **attributo di indicizzazione**). L'indice di solito memorizza ogni valore del campo di indicizzazione corredandolo di un elenco di puntatori a tutti i blocchi del disco che contengono record con quel valore del campo. I valori dell'indice sono *ordinati* in modo che si possa eseguire una ricerca binaria. Il file dell'indice è ovviamente molto più piccolo rispetto al file dei dati, quindi una ricerca binaria è ragionevolmente efficace. L'indicizzazione a più livelli evita la necessità di una ricerca binaria, ma richiede la creazione di ulteriori indici all'indice stesso.

Esistono molti tipi di indici ordinati. L'**indice primario** è specificato sul *campo chiave di ordinamento* di un file ordinato di record. Come si è detto in precedenza, il campo chiave di ordinamento è utilizzato per *ordinare fisicamente* i record del file su disco e tutti i record hanno un *valore univoco* per quel campo. Se il campo di ordinamento non è un campo chiave, cioè se molti record nel file possono avere lo stesso valore del campo di ordinamento, può essere usato un altro tipo di indice chiamato **indice di cluster**. Si noti che un file può avere al massimo un campo di ordinamento fisico, quindi può avere al massimo un indice primario oppure un indice di cluster, *ma non entrambi*. Su *qualsiasi campo non di ordinamento* di un file può essere specificato un terzo tipo di indice detto **indice secondario**. Un file può avere molti indici secondari oltre al suo metodo di accesso primario.

Indici primari

Un **indice primario** è un file ordinato i cui record sono di lunghezza fissa e sono costituiti da due campi. Il primo campo è dello stesso tipo di dati del campo chiave di ordinamento, chiamato **chiave primaria**, del file di dati e il secondo campo è un puntatore a un blocco del disco (un indirizzo di blocco). Esiste una **voce** (o **record dell'indice**) nel file dell'indice per ogni *blocco* nel file di dati. Ogni voce è composta da due campi che contengono il valore del campo della chiave primaria del *primo* record del blocco e un puntatore al corrispondente blocco su disco. Nel seguito, si farà riferimento ai due valori della voce *i* come $\langle K(i), P(i) \rangle$.

Per creare l'indice primario sul file ordinato mostrato in Figura 2.7, si usa il campo NOME come chiave primaria, perché questo è il campo di ordinamento del file (dando per scontato che ciascun valore di NOME è unico). Ciascuna voce dell'indice contiene un valore NOME e un puntatore (Figura 3.1). Le prime tre voci dell'indice sono:

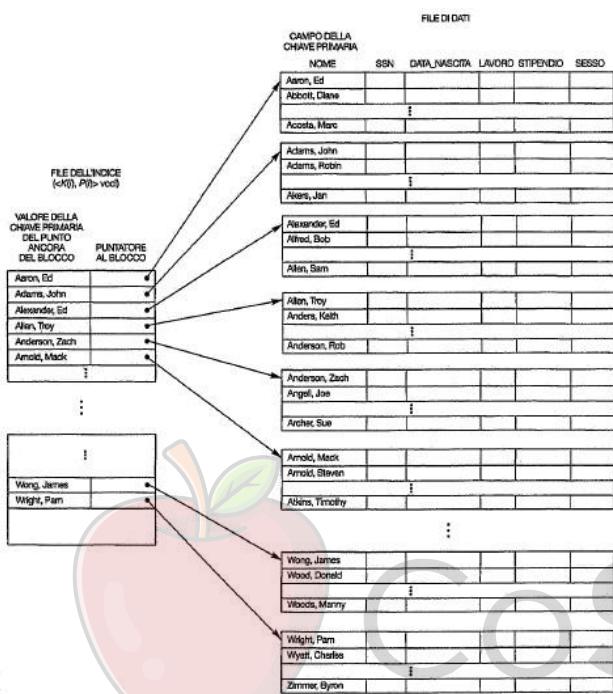


Figura 3.1 – Indice primario sulla chiave di ordinamento dei file mostrato in Figura 2.7.

$\langle K(1) = (\text{Aaron, Ed}), P(1) = \text{indirizzo del blocco 1} \rangle$
 $\langle K(2) = (\text{Adams, John}), P(2) = \text{indirizzo del blocco 2} \rangle$
 $\langle K(3) = (\text{Alexander, Ed}), P(3) = \text{indirizzo del blocco 3} \rangle$

Il numero complessivo di voci dell'indice è uguale al *numero di blocchi su disco* nel file di dati ordinato. Il primo record in ciascun blocco del file di dati è chiamato **record ancora** del blocco o semplicemente **punto ancora**.

Scienze
Associazione

Gli indici di questo tipo possono essere densi o sparsi: un **indice denso** contiene una voce per ogni *valore della chiave di ricerca* (e quindi ogni record) nel file di dati; un indice **sparso** (o **non denso**) contiene voci solo per alcuni valori di ricerca. L'indice primario è quindi un indice non denso (sparso), poiché include una voce per ogni blocco del file di dati piuttosto che per ogni valore di ricerca o per ogni record.

Il file indice di un indice primario occupa meno blocchi rispetto al file dei dati per due motivi. Prima di tutto, vi sono *meno voci* che record nel file di dati. In secondo luogo, ogni voce normalmente è di *dimensioni inferiori* rispetto a un record di dati perché ha solo due campi; di conseguenza un blocco su disco può contenere più voci dell'indice che record di dati. Una ricerca binaria nel file dell'indice richiede quindi meno accessi al blocco rispetto a una ricerca binaria nel file di dati.

Un record il cui valore della chiave primaria è K si trova nel blocco il cui indirizzo è $P(i)$, tale che $K(i) \leq K < K(i+1)$. Il blocco i -esimo del file di dati contiene tutti questi record a causa dell'ordinamento fisico dei suoi record rispetto al campo della chiave primaria. Per recuperare un record, dato il valore K della chiave primaria, si esegue una ricerca binaria nel file dell'indice per trovare la voce i corrispondente dell'indice e poi si prende il blocco del file di dati il cui indirizzo è $P(i)$.

L'Esempio 1 illustra il risparmio negli accessi ai blocchi che si può ottenere quando si usa un indice primario per cercare un record.

ESEMPIO 1. Si supponga di avere un file ordinato con $r = 30.000$ record memorizzati su disco con dimensione del blocco $B = 1024$ byte. I record del file hanno una dimensione fissa e sono indivisibili, con lunghezza del record $R = 100$ byte. Il fattore di blocco del file risulta essere $bfr = \lceil (B/R) \rceil = \lceil (1024/100) \rceil = 10$ record per blocco. Il numero di blocchi necessari per il file è $b = \lceil (r/bfr) \rceil = \lceil (30.000/10) \rceil = 3000$ blocchi. Una ricerca binaria sul file di dati richiede approssimativamente $\lceil \log_2 b \rceil = \lceil \log_2 3000 \rceil = 12$ accessi ai blocchi.

Si supponga ora che il campo della chiave di ordinamento del file sia lungo $V = 9$ byte, un puntatore a blocco sia lungo $P = 6$ byte e si ipotizzi di aver costruito un indice primario per il file. La dimensione di ciascuna voce dell'indice è $R_i = (9 + 6) = 15$ byte, quindi il fattore di blocco per l'indice è $bfr_i = \lceil (B/R_i) \rceil = \lceil (1024/15) \rceil = 68$ voci per blocco. Il numero totale delle voci dell'indice r_i è uguale al numero di blocchi nel file di dati, che è 3000. Il numero di blocchi dell'indice quindi è $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (3000/68) \rceil = 45$ blocchi. Per eseguire una ricerca binaria nel file dell'indice sono necessari $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 45) \rceil = 6$ accessi ai blocchi. Per cercare un record usando l'indice è necessario un ulteriore accesso ai blocchi del file di dati, per un totale di $6 + 1 = 7$ accessi, che corrisponde a un miglioramento rispetto alla ricerca binaria nel file di dati, che richiede 12 accessi.

Un problema serio che si verifica con l'indice primario, così come con qualsiasi file ordinato, è l'inserimento e l'eliminazione dei record. Nel caso dell'indice primario la questione è più grave perché, se si tenta di inserire un record nella sua posizione corretta nel file di dati, si devono non solo spostare gli altri record per creare spazio per quello nuovo, ma anche cambiare alcune voci dell'indice, perché lo spostamento dei record modificherà i punti di aggancio di alcuni blocchi. Usando un file di overflow non ordinato, come si è visto in precedenza, si può limitare questo problema. Un'altra possibilità è utilizzare una lista di record di overflow per ciascun blocco del file di dati. Questa tecnica è simile al metodo di gestione dei record di overflow descritto nel caso dell'hash nel paragrafo precedente. I record all'interno di ogni blocco e la corrispondente lista di overflow possono essere ordinati per migliorare il tempo di recupero. L'eliminazione dei record è gestita utilizzando degli indicatori di eliminazione.

Indici di cluster

Se i record di un file sono ordinati fisicamente rispetto a un campo che non è chiave, cioè che *non* ha un valore distinto per ciascun record, quel campo è chiamato **campo di raggruppamento**. È possibile creare un apposito tipo di indice, detto **indice di cluster**, per velocizzare il recupero dei record che hanno lo stesso valore del campo di raggruppamento. Questo indice differisce da un indice primario, che richiede che il campo di ordinamento del file di dati abbia un *valore distinto* per ogni record.

Anche l'indice di cluster è un file ordinato con due campi: il primo è dello stesso tipo del campo di raggruppamento del file di dati, mentre il secondo è un puntatore a un blocco. Vi è una voce nell'indice di cluster per ogni *valore distinto* del campo di raggruppamento, la quale contiene il valore del campo e un puntatore al *primo blocco* nel file di dati che contiene un record con quel valore del campo di raggruppamento. Se ne veda un esempio in Figura 3.2. Si noti che l'inserimento e l'eliminazione dei record causano ancora dei problemi perché i record dei dati sono fisicamente ordinati. Per ridurre il problema dell'inserimento è possibile riservare un intero blocco (oppure un raggruppamento di blocchi contigui) per *ciascun valore* del campo di raggruppamento; tutti i record con quel valore del campo sono posti nel blocco (o raggruppamento di blocchi). Questo rende l'inserimento e l'eliminazione relativamente semplici (Figura 3.3).

Un indice di cluster è un altro esempio di indice *non denso*, perché contiene una voce per ogni valore distinto del campo di indicizzazione piuttosto che per ogni record nel file. Un indice è qualcosa di simile alle strutture delle directory usate per l'hash estendibile. In entrambi i casi, infatti, si esegue una ricerca per trovare il puntatore al blocco di dati che contiene il record desiderato. La differenza principale è che la ricerca tramite indice utilizza i valori del campo di ricerca, mentre la ricerca delle directory hash usa un valore hash che è calcolato applicando la funzione hash al campo di ricerca.

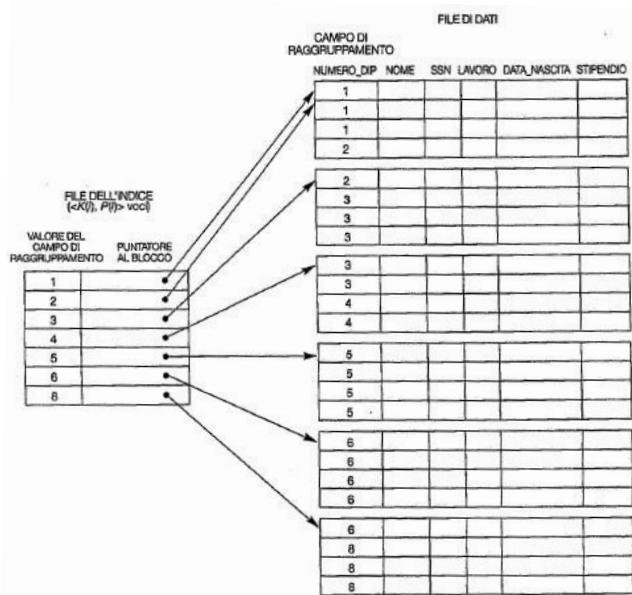


Figura 3.2 – Un indice di cluster sul campo `NUMERO_DIP` (campo di ordinamento non chiave) di un file `IMPIEGATO`.

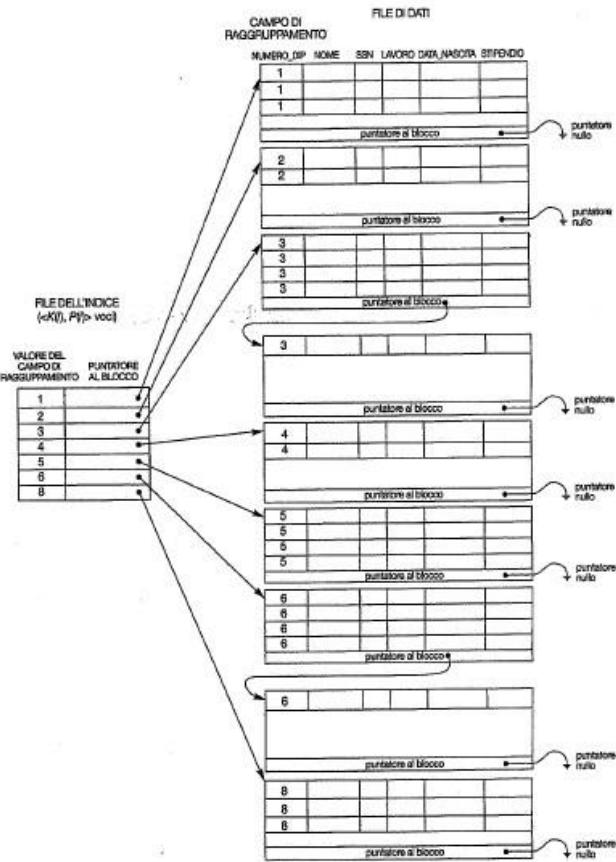


Figura 3.3 – L'indice di cluster usa un blocco separato per ciascun gruppo di record che condividono lo stesso valore del campo di raggruppamento.

Indici secondari

Un **indice secondario** è anch'esso un file ordinato con due campi. Il primo campo è dello stesso tipo di dati di un campo del file di dati, che non viene utilizzato per effettuare l'ordinamento del file di dati, chiamato **campo di indicizzazione**. Il secondo campo è un puntatore a *blocco* oppure un puntatore a *record*. Vi possono essere più indici secondari (e quindi campi di indicizzazione) per lo stesso file.

Si prenda in considerazione prima di tutto una struttura di accesso del tipo indice secondario su un campo chiave che un *valore distinto* per ciascun record. Un campo di questo tipo talvolta è chiamato **chiave secondaria**. In questo caso vi è una voce dell'indice per *ciascun record* del file di dati, la quale contiene il valore della chiave secondaria del record e un puntatore al blocco in cui il record è memorizzato oppure al record stesso. Questo indice, quindi, è **denso**.

Si indica nuovamente i valori dei due campi di una voce i con $<K(i), P(i)>$. Le voci sono **ordinate** rispetto al valore di $K(i)$, quindi si può eseguire una ricerca binaria. Visto che i record del file di dati *non* sono fisicamente ordinati rispetto ai valori del campo della chiave secondaria, non è possibile utilizzare i punti ancora dei blocchi. Questo perché viene creata una voce dell'indice per ciascun record nel file di dati invece che per ogni blocco come nel caso di un indice primario. In Figura 3.4 è illustrato un indice secondario in cui i puntatori $P(i)$ nelle voci dell'indice sono *puntatori a blocchi*, non puntatori a record. Una volta che il blocco appropriato è trasferito in memoria centrale, può essere eseguita la ricerca del record desiderato all'interno del blocco. Un indice secondario di solito ha bisogno di più spazio di memorizzazione e di un tempo di ricerca più lungo rispetto a un indice primario a causa del suo maggiore numero di voci. Il *miglioramento* nel tempo di ricerca di un record arbitrario, tuttavia, è decisamente maggiore per un indice secondario che per un indice primario, visto che se l'indice secondario non esistesse si dovrebbe svolgere una *ricerca lineare* nel file di dati. Nel caso

dell'indice primario, invece, si potrebbe eseguire una ricerca binaria sul file principale, anche se l'indice non esistesse. L'Esempio 2 illustra il miglioramento nel numero di accessi al blocco.

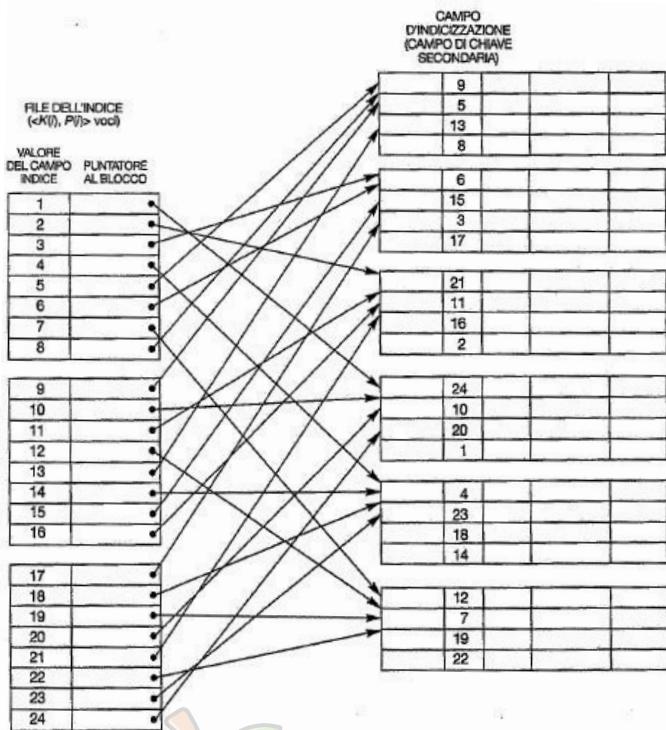


Figura 3.4 – Un indice secondario denso (con puntatori ai blocchi) su un campo chiave sul quale non è stato effettuato un ordinamento del file.

ESEMPIO 2. Si prenda in considerazione il file dell'Esempio 1 con $r = 30.000$ record di lunghezza fissa di dimensioni $R = 100$ byte memorizzati su un disco con dimensioni di blocco $B = 1024$ byte. Il file contiene $b = 3000$ blocchi, come calcolato nell'Esempio 1. Per eseguire una ricerca lineare sul file in media sarebbero necessari $b/2 = 3000/2 = 1500$ accessi a blocco. Si supponga di creare un indice secondario su un campo chiave del file, sul quale non è stato effettuato un ordinamento e che è lungo $V = 9$ byte. Come nell'Esempio 1, il puntatore ai blocchi è lungo $P = 6$ byte, quindi ciascuna voce dell'indice è $R_i = (9 + 6) = 15$ byte e il fattore di blocco per l'indice è $bfr_i = \lceil (B/R_i) \rceil = \lceil (1024/15) \rceil = 68$ voci per blocco. In un indice secondario denso come questo, il numero totale di voci dell'indice r_i è uguale al *numero di record* nel file di dati che è 30.000. Il numero di blocchi necessari per l'indice è quindi $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (30.000/68) \rceil = 442$ blocchi.

Una ricerca binaria su questo indice secondario richiede $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 442) \rceil = 9$ accessi di blocchi. Per cercare un record usando l'indice è indispensabile un ulteriore accesso a blocco del file di dati per un totale di $9 + 1 = 10$ accessi, un enorme miglioramento rispetto ai 1500 accessi mediamente necessari per una ricerca lineare, ma leggermente peggiore dei sette accessi necessari usando l'indice primario.

Si può creare anche un indice secondario su un *campo non chiave* di un file. In questo caso numerosi record nel file di dati possono avere lo stesso valore del campo di indicizzazione. Vi sono varie tecniche per realizzare un indice di questo tipo:

- Opzione 1: inserire più voci di indice con lo stesso valore $K(i)$, uno per ciascun record: essa dà come risultato un indice denso;
- Opzione 2: usare record di lunghezza variabile per le voci dell'indice co un campo ricorrente per il puntatore. Nella voce dell'indice per $K(i)$ viene tenuta una lista di puntatori $\langle P(i,1), \dots, P(i,k) \rangle$; la lista contiene un puntatore a ciascun blocco che ospita un record il cui valore del campo di indicizzazione è uguale a $K(i)$. Per questa tecnica e per la precedente l'algoritmo di ricerca binaria sull'indice deve essere modificato in modo adeguato;

- Opzione 3: mantenere le voci dell'indice a una lunghezza fissa e avere una singola voce per ciascun *valore del campo di indicizzazione*, ma creare un ulteriore livello di gestione e accesso ai puntatori multipli. In questo schema non denso, il puntatore $P(i)$ nella voce dell'indice $\langle K(i), P(i) \rangle$ fa riferimento a un blocco di puntatori ai record; ogni puntatore ai record in quel blocco si riferisce a uno dei record di dati con valore $K(i)$ per il campo di indicizzazione. Se troppi record condividono lo stesso valore $K(i)$, così che i loro puntatori ai record non possono essere contenuti in un singolo blocco del disco, viene utilizzato un agglomerato o una lista concatenata di blocchi. Si tratta della tecnica usata più comunemente (Figura 3.5). Il recupero attraverso l'indice richiede uno o più accessi a blocco aggiuntivi a causa del livello extra, ma gli algoritmi per eseguire ricerche nell'indice e, ancor più importante, per inserire nuovi record nel file di dati sono semplici. Inoltre, le ricerche con complicate condizioni di selezione possono essere gestite facendo riferimento ai soli puntatori ai record, senza dover recuperare molti record di dati non necessari.

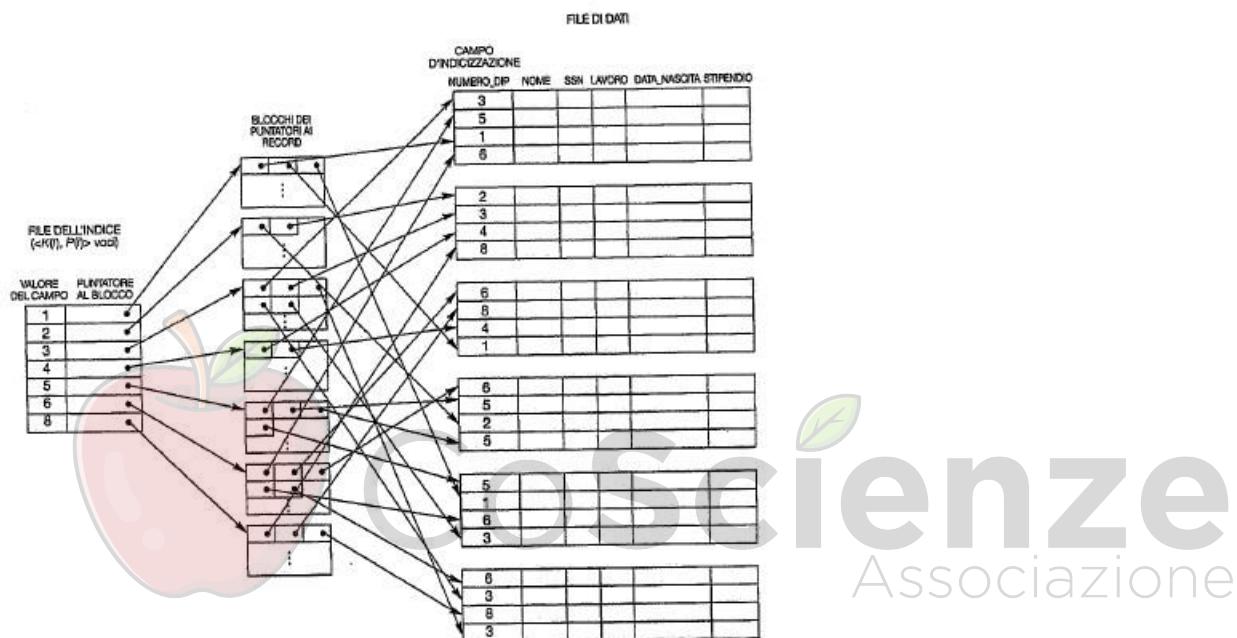


Figura 3.5 - Un indice secondario (con puntatori a record) su un campo non chiave realizzato usando una struttura dati con un ulteriore livello di gestione e accesso in modo che le voci dell'indice siano di lunghezza fissa e abbiano valori di campo univoci.

Si noti che un indice secondario fornisce un **ordinamento logico** dei record attraverso il campo di indicizzazione. Se si accede ai record secondo l'ordine delle voci dell'indice secondario, li si ottiene nell'ordine del corrispondente campo di indicizzazione.

A conclusione di questa prima parte, si riassume quanto detto sui tipi di indici in due tabelle: in Tabella 4 sono mostrate le caratteristiche del campo di indicizzazione di ogni tipo di indice ordinato a un solo livello: primario, di cluster e secondario; in Tabella 5 vengono riepilogate le proprietà di ciascun tipo di indice confrontando il numero di voci dell'indice e specificando quali indici sono densi e quali utilizzano ancora ai blocchi del file di dati.

	Campo ordering	Campo non ordering
Campo chiave	Indice Primario	Indice Secondario (chiave)
Campo non chiave	Indice Clustering	Indice Secondario (non chiave)

Tabella 4 – Tipi di indici.

	Numero di entry di 1° livello	Denso o non denso	Ancoraggio dei blocchi del file dati
Primario	Numero di blocchi del file dati	Non denso	Sì
Clustering	Numero di valori distinti del campo indexing	Non denso	Si/No*
Secondario (chiave)	Numero di record del file dati	Denso	No
Secondario (non chiave)	Num. record del file dati (caso 1) oppure num. valori distinti del campo indexing (caso 2 e 3)	Denso o non denso	No

Tabella 5 – Proprietà dei tipi di indici.

Sì, se ogni valore distinto del campo di ordinamento è collocato all'inizio di un nuovo blocco; altrimenti no.

Indice multilivello

Gli schemi d'indicizzazione descritti finora richiedono che il file dell'indice sia ordinato. Si usa la ricerca binaria per individuare i puntatori a un blocco sul disco oppure a un record (o ai record) nel file che ha il valore specificato del campo di indicizzazione. La ricerca binaria richiede approssimativamente ($\log_2 b_i$) accessi ai blocchi per un indice con b_i blocchi, perché ogni passo dell'algoritmo riduce la parte dell'indice in cui si deve continuare a cercare di un fattore pari a 2. Questo spiega perché si considera la funzione logaritmo in base 2. L'idea che sta alla base di un **indice multilivello** è ridurre a ogni passo la parte dell'indice in cui si continua a cercare di un fattore bfr_i , il fattore di blocco dell'indice, che è maggiore di 2. Lo spazio di ricerca, quindi, viene ridotto molto più velocemente. Il valore bfr_i è chiamato il **fan-out** dell'indice multilivello e si farà riferimento ad esso col simbolo **fo**. La ricerca in un indice multilivello richiede approssimativamente ($\log_{fo} b_i$) accessi ai blocchi, che è un numero inferiore rispetto a quello della ricerca binaria se il fan-out è maggiore di 2. Un indice multilivello considera il file dell'indice, al quale si fa riferimento ora come indice di primo livello (o livello base) di un indice multilivello, come un *file ordinato* con un *valore distinto* per ogni $K(i)$. A questo punto è possibile creare un indice primario per l'indice di primo livello; questo indice dell'indice di primo livello è detto **secondo livello** dell'indice multilivello. Poiché il secondo livello è un indice primario, si possono utilizzare i punti ancora dei blocchi in modo che il secondo livello contenga una voce per *ciascun blocco* del primo livello. Il fattore di blocco bfr_i , per il secondo livello, e per tutti i livelli successivi, è lo stesso dell'indice di primo livello, perché tutte le voci dell'indice hanno la medesima dimensione; ciascuna ha un valore del campo e un indirizzo del blocco. Se il primo livello ha r_1 voci e il fattore di blocco, che è anche il fan-out, è $bfr_i = fo$, il primo livello necessita di $[(r_1/fo)]$ blocchi, che è quindi il numero di voci r_2 necessarie al secondo livello dell'indice.

Si può ripetere questo procedimento per il secondo livello. Il **terzo livello**, che è un indice primario per l'indice di secondo livello, ha una voce per ogni blocco del secondo livello; il numero delle voci del terzo livello è quindi $r_3 = [(r_2/fo)]$. Si noti che è richiesto un secondo livello solo se il primo necessita di più di un blocco di disco e, in modo simile, è richiesto un terzo livello solo se il secondo occupa più di un blocco. Si può ripetere il procedimento precedente finché tutte le voci di un livello t dell'indice possono essere contenute in un singolo blocco. Questo blocco di livello t -esimo viene chiamato livello **superiore** dell'indice. Ogni livello riduce il numero di voci del livello precedente di un fattore di fo , il fan-out dell'indice, cosicché è possibile usare la formula $1 \leq (r_1/(fo)^t)$ per calcolare t . Quindi un indice multilivello con r_1 voci del primo livello avrà approssimativamente t livello, dove $t = [(\log_{fo}(r_1))]$.

Lo schema multilivello qui descritto può essere usato su qualsiasi tipo di indice, cioè primario, di cluster, o secondario, purché l'indice di primo livello abbia *valori distinti per K(i)* e *voci di lunghezza fissa*. In Figura 3.6 è rappresentato un indice multilivello creato su un indice primario. L'Esempio 3 illustra il miglioramento in termini di accessi a blocchi quando viene utilizzato un indice multilivello per cercare un record.

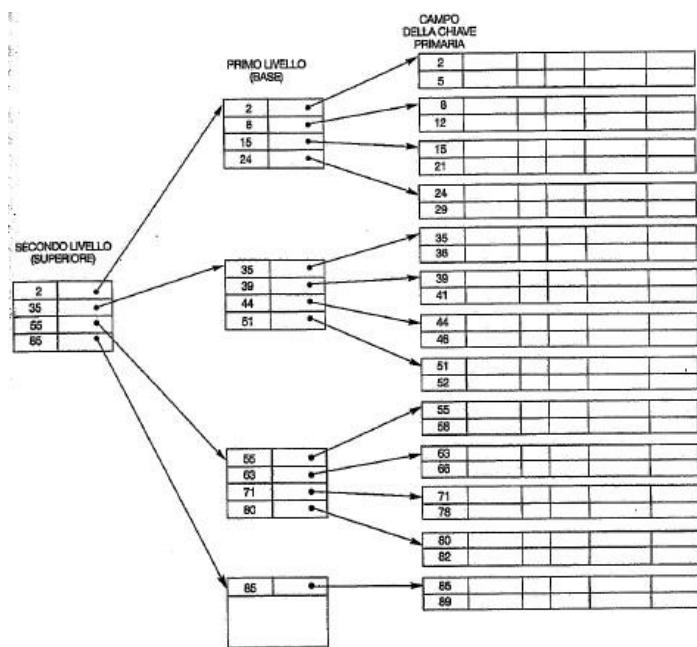


Figura 3.6 - Un indice primario a due livelli che assomiglia all'organizzazione ISAM.

ESEMPIO 3. Si supponga che l'indice secondario denso dell'Esempio 2 sia trasformato in un indice multilivello. Si è già calcolato il fattore di blocco dell'indice $bfr_i = 68$ voci dell'indice per blocco, che è anche il fan-out fo per l'indice multilivello; è stato calcolato anche il numero di blocchi del primo livello $b_1 = 442$. Il numero di blocchi del secondo livello sarà $b_2 = \lceil (b_1/fo) \rceil = \lceil (442/68) \rceil = 7$ blocchi e il numero di blocchi del terzo livello sarà $b_3 = \lceil (b_2/fo) \rceil = \lceil (7/68) \rceil = 1$ blocco. Il terzo livello, quindi, è il livello superiore dell'indice e $t = 3$. Per accedere a un record eseguendo una ricerca nell'indice multilivello, si deve accedere a un blocco di ciascun livello più al blocco del file di dati, quindi servono $t + 1 = 3 + 1 = 4$ accessi a blocchi. Si confronti questo risultato con quello dell'Esempio 2, in cui erano necessari 10 accessi dei blocchi poiché venivano usati un indice a un solo livello e la ricerca binaria.

Come si è visto, un indice multilivello riduce il numero degli accessi a blocchi durante la ricerca di un record, per un dato valore del campo di indicizzazione. Occorre ancora affrontare i problemi di gestione delle eliminazioni e degli inserimenti nell'indice, perché tutti i livelli dell'indice sono *file fisicamente ordinati*. Per mantenere i vantaggi dell'utilizzo dell'indicizzazione multilivello senza incorrere nei problemi di eliminazione e di inserimento, i progettisti hanno adottato un tipo di indice multilivello che lascia dello spazio libero in ciascuno dei suoi blocchi per l'immissione di nuove voci. Questo è detto **indice dinamico multilivello** ed è spesso implementato usando le strutture di dati chiamate alberi B e alberi B⁺ descritti successivamente.

Indici dinamici multilivello implementati da alberi B e alberi B⁺

Gli alberi B e gli alberi B⁺ sono casi speciali degli alberi della ben nota struttura di dati. Si presenta qui di seguito molto brevemente la terminologia usata nella loro descrizione. Un **albero** è formato da **nodi**. Ogni nodo, tranne uno speciale chiamato **radice**, ha un **padre** e zero o più nodi **figlio**. La **radice** non ha il nodo padre. Un nodo che non ha alcun nodo figlio è chiamato **foglia**; un nodo che non è foglia è un **nodo interno**. Il **livello** di un nodo è sempre superiore di una unità rispetto al livello del suo nodo padre, mentre il livello del nodo radice è zero. Un **sottoalbero** di un nodo è costituito da quel nodo e da tutti i suoi nodi **discendenti**, i suoi nodi figli, i nodi figli dei suoi nodi figli e così via. Una precisa definizione ricorsiva di un sottoalbero dice che un sottoalbero è costituito da un nodo *n* e dai sottoalberi di tutti i nodi figli *n*. In Figura 3.7 è illustrata una struttura di dati ad albero: il nodo radice è A e i suoi nodi figli sono B, C e D. I nodi E, J, C, G, H e K sono nodi foglia. Di solito si visualizza un albero con il nodo radice posto nella parte superiore, proprio come

mostrato in figura. Un modo per implementare un albero è inserire in ciascun nodo tanti puntatori quanti sono i nodi figli di quel nodo. In alcuni casi in ciascun nodo è anche memorizzato un puntatore al padre. Oltre ai puntatori, un nodo di solito contiene qualche tipo di informazioni. Quando un indice multilivello viene implementato come albero, le informazioni includono i valori del campo di indicizzazione usati per guidare la ricerca di un particolare record.

Successivamente sono presentati gli alberi di ricerca e gli alberi B, che possono essere utilizzati come indici dinamici multilivello per guidare la ricerca dei record in un file di dati. I nodi dell'albero B sono occupati fra il 50 e il 100 per cento e i puntatori ai blocchi di dati sono memorizzati sia nei nodi interni sia nei nodi foglia dell'albero. Saranno poi presentati gli alberi B⁺, una variante degli alberi B in cui i puntatori ai blocchi dei dati di un file sono memorizzati solo nei nodi foglia; questo può richiedere meno livelli e permette di ottenere indici di capacità più elevata.

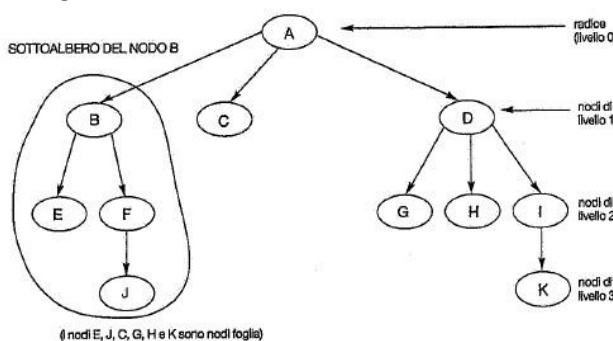


Figura 3.7 - Una struttura di dati ad albero che mostra un albero non bilanciato.

ricerca e gli alberi B, che possono essere utilizzati come indici dinamici multilivello per guidare la ricerca dei record in un file di dati. I nodi dell'albero B sono occupati fra il 50 e il 100 per cento e i puntatori ai blocchi di dati sono memorizzati sia nei nodi interni sia nei nodi foglia dell'albero. Saranno poi presentati gli alberi B⁺, una variante degli alberi B in cui i puntatori ai blocchi dei dati di un file sono memorizzati solo nei nodi foglia; questo può richiedere meno livelli e permette di ottenere indici di capacità più elevata.

Alberi di ricerca e alberi B

Un albero di ricerca è un tipo di albero speciale che viene usato per guidare la ricerca di un record, dato il valore di uno dei suoi campi. Gli indici multilivello trattati in precedenza possono essere pensati come una variante di un albero di ricerca: ogni nodo dell'indice multilivello può avere fino a *fo* puntatori e *fo* valori chiave, dove *fo* è il fan-out dell'indice. I valori dei campi dell'indice di ciascun nodo guidano l'utente al nodo successivo finché raggiunge il blocco del file di dati che contiene i record richiesti. Seguendo un puntatore si limita a ogni passo la ricerca a un sottoalbero dell'albero di ricerca e si ignorano tutti i nodi che non sono di quel sottoalbero.

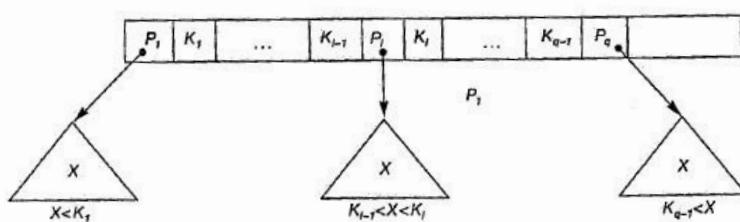


Figura 3.8 - Un nodo di un albero di ricerca con i tre sottoalberi che da lui si dipartono.

Alberi di ricerca. Un albero di ricerca è leggermente differente da un indice multilivello. Un **albero di ricerca** di ordine p è un albero tale che ogni suo nodo contiene *al massimo* $p - 1$ valori di ricerca e i p puntatori sono nell'ordine $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$, in cui $q \leq p$; ogni P_i è un puntatore a un nodo figlio (oppure è un puntatore vuoto) e ogni K_i è un valore di ricerca preso da un insieme ordinato di valori.

Si suppone che tutti i valori di ricerca siano univoci. In Figura 3.8 viene mostrato un nodo di un albero di ricerca. Due vincoli devono sempre essere rispettati in un albero di ricerca:

1. All'interno di ciascun nodo, $K_1 < K_2 < \dots < K_{q-1}$;
2. Per tutti i valori X dei sottoalberi ai quali si fa riferimento da P_i , si ha $K_{i-1} < X < K_i$ per $1 < i < q$;
 $X < K_i$ per $i = 1$ e
 $K_{i-1} < X$ per $i = q$.

Ogni volta che si cerca un valore X , si segue il puntatore appropriato P_i secondo le formule esposte nel punto 2. In Figura 3.9 è rappresentato un albero di ricerca di ordine $p = 3$, i cui valori di ricerca sono interi. Si noti che alcuni dei puntatori P_i posti in un nodo possono essere puntatori nulli.

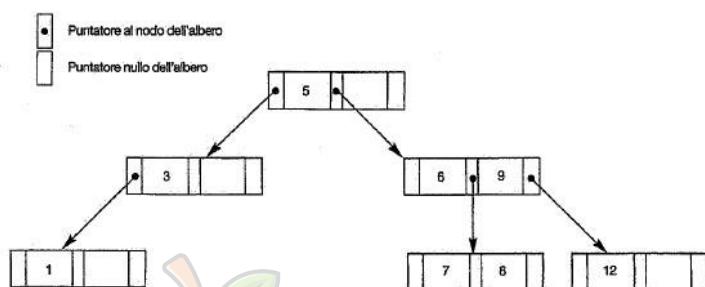


Figura 3.9 - Un albero di ricerca di ordine $p = 3$.

Si può usare un albero di ricerca come meccanismo per ricercare i record immagazzinati in un file su disco. I valori nell'albero possono essere i valori di uno dei campi del file chiamato **campo di ricerca** (che equivale al campo di indicizzazione se un indice multilivello viene utilizzato per guidare la ricerca).

Ogni valore chiave dell'albero è associato a un puntatore al record nel file di dati che ha quel

valore. In alternativa il puntatore può puntare al blocco del disco che contiene quel record. Lo stesso albero di ricerca può essere memorizzato su disco assegnando ogni nodo dell'albero a un blocco del disco. Quando viene inserito un nuovo record si deve aggiornare l'albero di ricerca inserendo nell'albero una voce contenente il valore del campo di ricerca del nuovo record e un puntatore al nuovo record.

Nuovi algoritmi sono necessari per inserire e cancellare i valori di ricerca nell'albero di ricerca, mentre si continuano a rispettare i due vincoli precedenti. In generale questi algoritmi non garantiscono che un albero di ricerca sia **bilanciato**, cioè che tutti i suoi nodi foglia siano allo stesso livello. L'albero in Figura 3.7 non è bilanciato perché ha nodi foglia ai livelli 1, 2 e 3. Mantenere un albero di ricerca bilanciato è importante, perché ciò garantisce che nessun nodo si troverà a livelli molto alti e quindi richiederà molti accessi ai blocchi durante la ricerca. Un altro problema con gli alberi di ricerca è che l'eliminazione dei record possa lasciare alcuni nodi nell'albero quasi vuoti, sprecando quindi spazio per la memorizzazione e aumentando il numero dei livelli. La tecnica degli alberi B affronta entrambi questi problemi specificando ulteriori vincoli sull'albero di ricerca.

Gli alberi B. L'albero B soddisfa ulteriori vincoli che assicurano che l'albero sia sempre bilanciato e che lo spazio sprecato a seguito di cancellazione, se ce n'è, non diventi mai eccessivo. Gli algoritmi per l'inserimento e la cancellazione, però, diventano più complessi allo scopo di soddisfare questi vincoli. Ciononostante, in genere gli inserimenti e le cancellazioni sono processi semplici; diventano complicati solo in particolari circostanze, cioè ogni volta che si tenta un inserimento in un nodo che è già completo oppure una cancellazione da un nodo che così diventa più della metà vuoto. In modo più formale, un **albero B** di **ordine p** , quando è usato come una struttura di accesso su un *campo chiave* per cercare i record in un file di dati, può essere definito nel seguente modo:

- Ogni nodo interno dell'albero B (Figura 3.10°) ha questa forma:

$$\langle P_1, \langle K_1, Pr_1 \rangle, P_2, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_q \rangle$$

dove $q \leq p$. Ogni P_i è un **puntatore a un albero**, cioè un puntatore a un altro nodo nell'albero B. Ogni Pr_i è un **puntatore ai dati**, cioè un puntatore al record il cui valore del campo chiave è uguale a K_i (oppure al blocco del file di dati che contiene quel record);

- All'interno di ogni nodo, $K_1 < K_2 < \dots < K_{q-1}$;
- Per tutti i valori X del campo chiave di ricerca nel sottoalbero a cui si è fatto riferimento da P_i (il sottoalbero i -esimo, Figura 3.10a), si ha:

$$K_{i-1} < X < K_i \text{ per } 1 < i < q; X < K_1 \text{ per } i = 1 \text{ e } K_{i-1} < X \text{ per } i = q;$$

- Ogni nodo contiene al massimo p puntatori dell'albero;
- Ogni nodo, tranne i nodi radice e i nodi foglia, ha almeno $\lceil (p/2) \rceil$ puntatori dell'albero; il nodo radice ha almeno due puntatori dell'albero a meno che sia l'unico nodo dell'albero;
- Un nodo con q puntatori dell'albero, dove $q \leq p$, contiene $q - 1$ valori del campo chiave di ricerca (e quindi ha $q - 1$ puntatori ai dati);
- Tutti i nodi foglia sono allo stesso livello e hanno la medesima struttura dei nodi interni, a parte il fatto che tutti i loro *puntatori a un albero* P_i sono nulli.

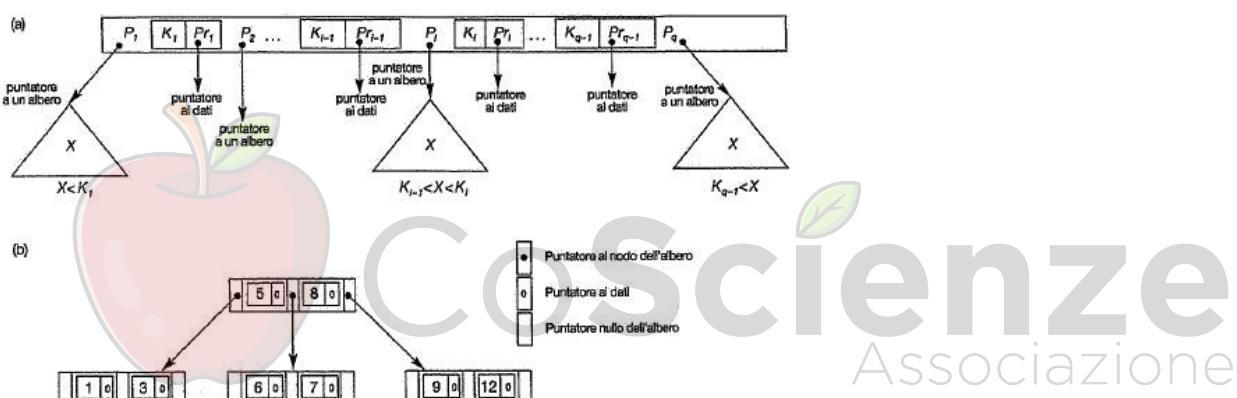


Figura 3.10 - La struttura dell'albero B. (a) Un nodo di un albero B con $(q - 1)$ valori di ricerca. (b) Un albero B di ordine $p = 3$. I valori sono stati inseriti nell'ordine 8, 5, 1, 7, 3, 12, 9, 6.

In Figura 3.10(b) è illustrato un albero B di ordine $p = 3$. Si noti che tutti i valori di ricerca K di un albero B sono univoci perché si suppone che l'albero sia usato come una struttura di accesso su un campo chiave. Se si usa un albero B su un campo non chiave, si deve cambiare la definizione dei puntatori al file Pr_i , perché i puntatori Pr_i devono far riferimento a un blocco o raggruppamento di blocchi, che contiene i puntatori ai record del file. Questo ulteriore livello di indici è simile all'opzione 3 presentata per gli indici secondari.

Un albero B inizia con un solo nodo radice (che è anche un nodo foglia) a livello 0. Quando il nodo radice diventa completo e con $p - 1$ valori della chiave di ricerca e si tenta di inserire un'altra voce nell'albero, il nodo radice si divide in due nodi di livello 1. Solo il valore centrale è tenuto nel nodo radice, mentre il resto dei valori sono divisi equamente tra gli altri due nodi. Quando uno dei nodi non radice è completo e una nuova voce dovrebbe esservi inserita, il nodo è diviso in due nodi dello stesso livello e la voce centrale è spostata verso il nodo padre insieme ai due puntatori ai nuovi nodi ottenuti dalla divisione. Se il nodo padre è completo viene diviso anch'esso. La divisione può propagarsi per tutto il tragitto verso il nodo radice, creando un nuovo livello se anche la radice deve essere divisa.

Se la cancellazione di un valore fa sì che un nodo sia completo per meno della metà, viene fuso con i suoi nodi vicini; anche questa fusione può propagarsi lungo tutto il tragitto verso la radice. La cancellazione, quindi, può ridurre il numero dei livelli dell'albero. È stato mostrato attraverso l'analisi e la simulazione che, dopo numerosi inserimenti e cancellazione casuali in un albero B, i nodi sono approssimativamente completi

al 69% quando il numero dei valori nell'albero si stabilizza. Questo vale anche per gli alberi B⁺. Se ciò avviene, la divisione e l'unione dei nodi si verificano solo raramente, quindi l'inserimento e la cancellazione diventano abbastanza efficaci. Se il numero dei valori aumenta ancora, l'albero si espanderà senza problemi, anche se si può verificare la divisione dei nodi e così alcuni inserimenti richiederanno più tempo. L'Esempio 4 mostra come si effettua il calcolo dell'ordine p di un albero B memorizzato su disco.

ESEMPIO 4. Si supponga che il campo di ricerca sia lungo $V = 9$ byte, la dimensione del blocco del disco sia $B = 512$ byte, il puntatore a un record (di dati) sia $P_r = 7$ byte e il puntatore a un blocco sia $P = 6$ byte. Ogni nodo dell'albero B può avere *al massimo* p puntatori dell'albero, $p - 1$ puntatori ai dati e $p - 1$ valori del campo della chiave di ricerca (Figura 3.10a). Questi devono essere contenuti in un singolo blocco del disco, perché ciascun nodo dell'albero B deve corrispondere a un blocco del disco. Quindi si deve avere:

$$\begin{aligned}(p * P) + ((p - 1) * (P_r + V)) &\leq B \\ (p * 6) + ((p - 1) * (7 + 9)) &\leq 512 \\ (22 * p) &\leq 528\end{aligned}$$

Il valore p deve soddisfare la disegualanza precedente, il che dà $p = 23$ ($p = 24$ non viene scelto per motivi forniti in seguito).

In generale un nodo dell'albero B può contenere ulteriori informazioni necessarie per gli algoritmi che manipolano l'albero, ad esempio il numero di voci q nel nodo e un puntatore al nodo padre. Prima di eseguire il calcolo precedente per p , quindi, si dovrebbe ridurre la dimensione del blocco della quantità di spazio necessario per tutte queste informazioni. Nel seguito si illustra come calcolare il numero di blocchi e di livelli per un albero B.

ESEMPIO 5. Si supponga che il campo di ricerca dell'Esempio 4 sia un campo chiave ma non di ordinamento e che sia necessario costruire un albero B su questo campo. Si ipotizzi che ciascun nodo dell'albero B sia completo al 69%. Ogni nodo, in media, avrà $p * 0,69 = 23 * 0,69$ oppure approssimativamente 16 puntatori e, quindi, 15 valori della chiave di ricerca. Il **fan-out medio** è $fo = 16$. Si può iniziare dalla radice e vedere quanti valori e puntatori possono esistere in media a ogni livello successivo:

Radice:	1 nodo	15 voci	16 puntatori
Livello 1:	16 nodi	240 voci	256 puntatori
Livello 2:	256 nodi	3840 voci	4096 puntatori
Livello 3:	4096 nodi	61.440 voci	

A ciascun livello si è calcolato il numero di voci moltiplicando il numero totale dei puntatori del livello precedente per 15, il numero medi di voci in ciascun nodo. Quindi, per le date dimensioni del blocco, dimensione del puntatore e dimensione del campo della chiave di ricerca, un albero B a due livelli contiene $3840 + 240 + 15 = 4095$ voci in medi; un albero B a tre livelli contiene 65.535 voci in media.

Gli alberi B sono usati talvolta come organizzazione del file di dati. In questo caso tutti i record sono memorizzati all'interno dei nodi dell'albero B invece che solo le voci <chiave di ricerca, puntatore del record>. Questo funziona bene per i file con un *numero relativamente piccolo di record* e una *dimensione ridotta del record*, altrimenti il fan-out e il numero di livelli diventano troppo grandi per permettere un accesso efficace. Riassumendo, gli alberi B forniscono una struttura di accesso multilivello che produce un albero bilanciato in cui ogni nodo è riempito almeno per metà. Ciascun nodo di un albero B di ordine p può avere al massimo $p - 1$ valori di ricerca.

Alberi B⁺

La maggior parte delle odierne implementazioni di indici multilivello dinamici utilizza una variante della struttura di dati ad albero B, chiamata **albero B⁺**. In un albero B, tutti i valori del campo di ricerca appaiono una volta a un dato livello nell'albero, insieme a un puntatore ai dati. In un albero B⁺ i puntatori ai dati sono memorizzati *solo nei nodi foglia dell'albero*; quindi la struttura dei nodi foglia differisce dalla struttura dei nodi interni. I nodi foglia contengono una voce per *ogni* valore del campo di ricerca, insieme a un puntatore al record dei dati (oppure al blocco che contiene questo record) se il campo di ricerca è un campo chiave. Per un campo di ricerca non chiave, il puntatore fa riferimento a un blocco che contiene i puntatori ai record del file di dati, creando un livello ulteriore di indici. I nodi foglia dell'albero B⁺ di solito sono collegati tra loro per fornire un accesso ordinato al campo di ricerca e ai record. Questi nodi foglia sono simili al primo livello (base) di un indice. I nodi interni dell'albero B⁺ corrispondono agli altri livelli di un indice multilivello. Alcuni valori del campo di ricerca dai nodi foglia sono *ripetuti* nei nodi interni dell'albero B⁺ per guidare la ricerca. La struttura dei *nodi interni* di un albero B⁺ di ordine p (Figura 3.11°) è la seguente:

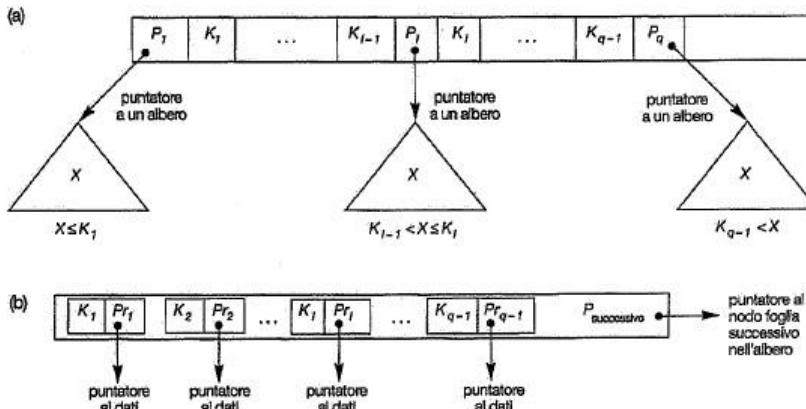


Figura 3.11 – I nodi di un albero B⁺. (a) Un nodo interno di un albero B⁺ con $q - 1$ valori di ricerca. (b) Un nodo foglia di un albero B⁺ con $q - 1$ valori di ricerca e $q - 1$ puntatori ai dati.

1. Ogni nodo interno ha la seguente forma:

$$\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$$
 in cui $q \leq p$ e ogni P_i è un **puntatore a un albero**;
2. All'interno di ogni nodo interno, $K_1 < K_2 < \dots < K_{q-1}$;
3. Per tutti i valori X del campo di ricerca nel sottoalbero a cui si fa riferimento da P_i , si ha
 $K_{i-1} < X \leq K_i$ per $1 < i < q$;
 $X \leq K_i$ per $i = 1$;
 $K_{i-1} < X$ per $i = q$ (Figura 3.11a);
4. Ogni nodo interno ha al massimo p puntatori ad un albero;
5. Ogni nodo interno, tranne la radice, ha almeno $\lceil (p/2) \rceil$ puntatori dell'albero; il nodo radice ha almeno due puntatori dell'albero se è un nodo interno;
6. Un nodo interno con q puntatori, $q \leq p$, ha $q - 1$ valori del campo di ricerca.

La struttura dei *nodi esterni* di un albero B⁺ dell'ordine p (Figura 3.11b) è la seguente:

- Ogni nodo foglia ha la forma:

$$\langle\langle K_1, Pr_1 \rangle, \langle K_2, Pr_2 \rangle, \dots, \langle K_q, Pr_q \rangle, P_{successivo} \rangle$$

in cui $q \leq p$, ogni Pr_i è un puntatore ai dati e $P_{successivo}$ si riferisce al *nodo foglia* successivo dell'albero B⁺;

- All'interno di ogni nodo foglia, $K_1 < K_2 < K_{q-1}, q \leq p$;
- Ciascun Pr_i è un **puntatore ai dati** che fa riferimento al record il cui valore del campo di ricerca è K_i o a un blocco del file che contiene il record (o a un blocco di puntatori a record che fa riferimento ai record il cui valore del campo di ricerca è K_i se il campo di ricerca non è una chiave);
- Ogni nodo foglia ha almeno $\lceil (p/2) \rceil$ valori;
- Tutti i nodi foglia sono allo stesso livello.

I puntatori nei nodi interni sono *puntatori a un albero* di blocchi che sono nodi dell'albero, mentre i puntatori nei nodi foglia sono *puntatori ai dati* che fanno riferimenti ai record o ai blocchi del file di dati, a eccezione del puntatore $P_{successivo}$ che è un puntatore a un albero del successivo nodo esterno. Iniziando dal nodo foglia più a sinistra è possibile attraversare i nodi foglia come se fossero una lista concatenata, usando i puntatori $P_{successivo}$. Questo fornisce un accesso ordinato ai record di dati basato sul campo di indicizzazione. Può essere incluso anche un puntatore $P_{precedente}$. Per un albero B⁺ su un campo non chiave, è necessario un livello ulteriore di indici come quello mostrato in Figura 3.5, così che i puntatori Pr sono puntatori ai blocchi che contengono un insieme di puntatori ai record effettivi del file di dati, come specificato nell'opzione 3 del paragrafo precedente.

Poiché le voci nei *nodi interni* di un albero B⁺ includono valori di ricerca e puntatori a un albero senza alcun puntatore ai dati, un nodo interno di un albero B⁺ può contenere più voci del nodo corrispondente di un albero B. A parità di dimensione del blocco (nodo), l'ordine p sarà più grande per un albero B⁺ rispetto a un albero B, come illustrato nell'Esempio 6. Questo fa sì che l'albero B⁺ abbia meno livelli, migliorando il tempo di ricerca. Poiché le strutture dei nodi interni e foglia di un albero B⁺ sono diverse, l'ordine p può essere differente nei due casi. Si userà p per denotare l'ordine dei *nodi interni* e p_{foglia} per denotare l'ordine dei *nodi foglia* e definito come il numero massimo dei puntatori ai dati presenti in un nodo foglia.

ESEMPIO 6. Per calcolare l'ordine p di un albero B⁺ si supponga che la lunghezza del campo della chiave di ricerca sia $V = 9$ byte, la dimensione del blocco sia $B = 512$ byte, il puntatore a un record sia $P_r = 7$ byte e il puntatore a un blocco sia $P = 6$ byte, come nell'Esempio 4. Un nodo interno dell'albero B⁺ può avere fino a p puntatori a un albero e $p - 1$ valori del campo di ricerca; questi devono essere contenuti in un singolo blocco. Quindi si avrà:

$$\begin{aligned}(p * P) + ((p - 1) * V) &\leq B \\(p * 6) + ((p - 1) * 9) &\leq 512 \\(15 * p) &\leq 521\end{aligned}$$

Si può scegliere che p sia il valore più grande che soddisfa la diseguaglianza precedente, il che dà $p = 34$. Ciò è maggiore rispetto al valore 23 calcolato per l'albero B e dà come risultato un fan-out più ampio e un maggior numero di voci in ogni nodo interno di un albero B⁺ rispetto al corrispondente albero B. I nodi foglia dell'albero B⁺ avranno lo stesso numero di valori e puntatori, a eccezione del fatto che i puntatori sono puntatori ai dati, e un puntatore al nodo successivo. Quindi l'ordine p_{foglia} per i nodi foglia può essere calcolato nel modo seguente:

$$\begin{aligned}(p_{foglia} * (P_r + V)) + P &\leq B \\(p_{foglia} * (7 + 9)) + 6 &\leq 512 \\(16 * p_{foglia}) &\leq 506\end{aligned}$$

Ciascun nodo foglia può quindi contenere fino a $p_{\text{foglia}} = 31$ combinazioni di puntatori ai dati o valori chiave, dando per scontato che i puntatori ai dati siano i puntatori a record.

Anche nel caso dell'albero B^+ , può essere necessario che ogni nodo contenga ulteriori informazioni per implementare gli algoritmi d'inserimento e di cancellazione. Queste informazioni possono comprendere il tipo di nodo (interno o foglia), il numero di voci q correnti del nodo e i puntatori ai nodi padre e fratelli. Prima di eseguire i calcoli precedenti per p e p_{foglia} , quindi, si dovrebbe ridurre la dimensione del blocco della quantità di spazio necessario per contenere tutte queste informazioni. L'esempio successivo mostra come si calcola il numero di voci di un albero B^+ .

ESEMPIO 7. Si supponga di costruire un albero B^+ sul campo dell'Esempio 6. Per calcolare il numero approssimativo di voci nell'albero B^+ si supponga che ogni nodo sia completo al 69%. In media ciascun nodo interno avrà $34 * 0,69$ o approssimativamente 23 puntatori e quindi 22 valori. Ogni nodo foglia in media conterrà $0,69 * p_{\text{foglia}} = 0,69 * 31$ o approssimativamente 21 puntatori ai record di dati. L'albero B^+ avrà il seguente numero medio di voci a ogni livello:

Radice:	1 nodo	22 voci	23 puntatori
Livello1:	23 nodi	506 voci	529 puntatori
Livello2:	529 nodi	11.638 voci	12.167 puntatori
Livello esterno:	12.167 nodi		255.507 puntatori ai record

Per la dimensione del blocco, la dimensione del puntatore e la dimensione del campo di ricerca indicati precedentemente, l'albero B^+ a tre livelli contiene in media fino a 255.507 puntatori ai record. Si confronti questo valore con le 65.535 voci necessarie per il corrispondente albero B dell'Esempio 5.

Inserimento e cancellazione con alberi B^+

In Figura 3.12 viene illustrato l'inserimento dei record in un albero B^+ di ordine $p = 3$ e $p_{\text{foglia}} = 2$. Prima di tutto si osservi che la radice è l'unico nodo nell'albero, quindi è anche un nodo foglia. Non appena si crea più di un livello, l'albero viene diviso in nodi interni e foglia. Si noti che *ogni valore chiave deve esistere a livello delle foglie*, perché tutti i puntatori ai dati si trovano a livello delle foglie. Tuttavia, solo alcuni valori sono presenti nei nodi interni per guidare la ricerca. Si consideri anche che ogni valore che appare in un nodo interno compare anche come *il valore più a destra* tra i nodi foglia del sottoalbero a cui fa riferimento il puntatore all'albero posto a sinistra del valore.

Quando un *nodo foglia* è completo e viene inserita una nuova voce, il nodo **trabocca** (overflow) e il suo contenuto viene diviso in due. Le prime $j = \lceil ((p_{\text{foglia}} + 1)/2) \rceil$ voci del nodo originale sono mantenute in questo stesso nodo, mentre le restanti voci sono spostate in un nuovo nodo foglia. Il valore di ricerca j -esimo è replicato nel nodo interno padre e un ulteriore puntatore al nuovo nodo è creato nel padre. Questi valori devono essere inseriti nel nodo padre nella sequenza corretta. Se il nodo interno padre è completo, il nuovo valore gli causerà un trabocco, quindi anch'esso dovrà essere diviso in due. Le voci nel nodo interno fino a P_j , il puntatore a un albero j -esimo dopo l'inserimento del nuovo valore e del puntatore, con $j = \lceil (p + 1)/2 \rceil$, sono mantenuti, mentre il valore di ricerca j -esimo è spostato nel padre, ma non ripetuto. Un nuovo nodo interno conterrà le voci rimanenti da P_{j+1} in poi. Questa divisione può propagarsi lungo tutto il tragitto fino a creare un nuovo nodo radice e quindi un nuovo livello dell'albero B^+ .

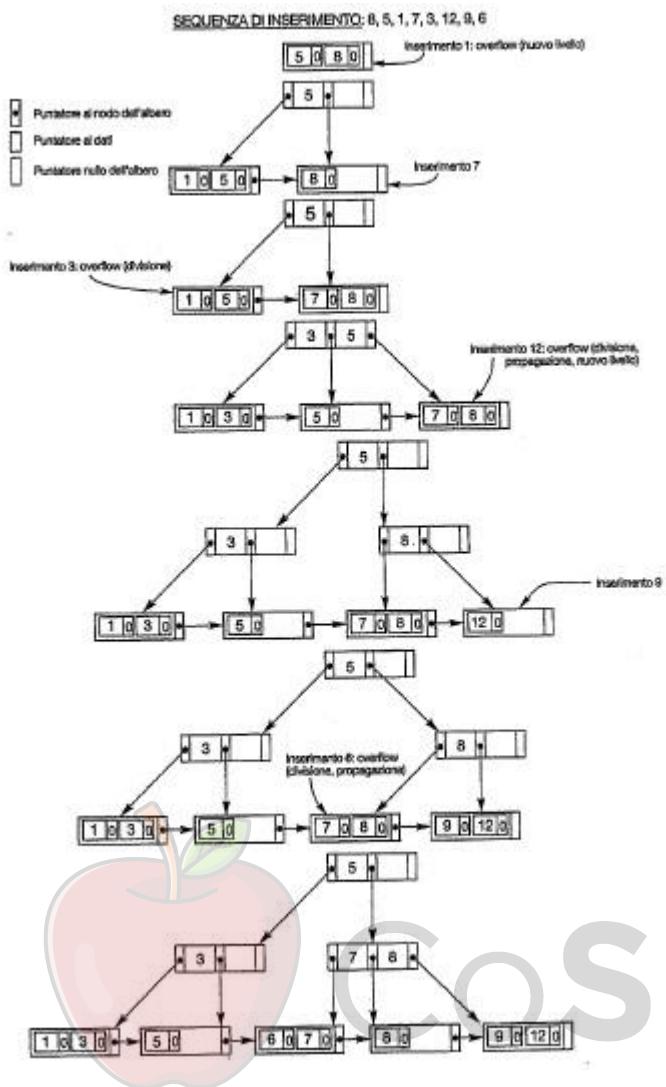


Figura 3.12 - Un esempio di inserimento in un albero B^+ con $p = 3$ e $p_{\text{foglia}} = 2$.

L'inserimento di 6 provoca un overflow nel nodo $(7, 8)$, che viene scisso, propagando il valore 7 nel nodo padre.

In Figura 3.13, è illustrata la cancellazione di una voce da un albero B^+ . quando una voce è cancellata, è sempre rimossa dal livello delle foglie. Se capita che si trovi in un nodo interno, deve essere eliminata anche da lì. In questo caso il valore alla sua sinistra nel nodo foglia deve sostituirlo nel nodo interno, perché quel valore ora è la voce più a destra nel sottoalbero. La cancellazione può causare uno **svuotamento** riducendo il numero di voci nel nodo foglia sotto il minimo richiesto. In questo caso si cerca di trovare un nodo foglia **fratello**, cioè un nodo esterno esattamente a sinistra o a destra del nodo con lo svuotamento, e di **ridistribuire** le voci tra il nodo e suo fratello in modo che entrambi siano almeno completi a metà; altrimenti il nodo è fuso con i suoi fratelli e il numero di nodi foglia si riduce. Un metodo comune è provare a ridistribuire le voci nel fratello a sinistra; se non è possibile, viene fatto un tentativo di ridistribuzione con il fratello a destra. Se anche questo non è possibile, i tre nodi vengono fusi ottenendo due nodi esterni. In questo caso lo svuotamento può propagarsi ai nodi **interni** perché sono necessari un puntatore a un albero e un valore di ricerca in meno. Questo effetto può propagarsi e ridurre i livelli dell'albero.

Si vogliono inserire dei record in un albero B^+ di ordine $p = 3$ e $p_{\text{foglia}} = 2$, nella sequenza: 8, 5, 1, 7, 3, 12, 9, 6.

L'inserimento dei valori 8 e 5 non provoca overflow: sono entrambi inseriti nella radice.

L'inserimento di 1 provoca overflow. Il nodo è scisso ed il valore centrale è ripetuto in un nuovo nodo radice. Alcuni valori sono replicati nei nodi interni per guidare la ricerca.

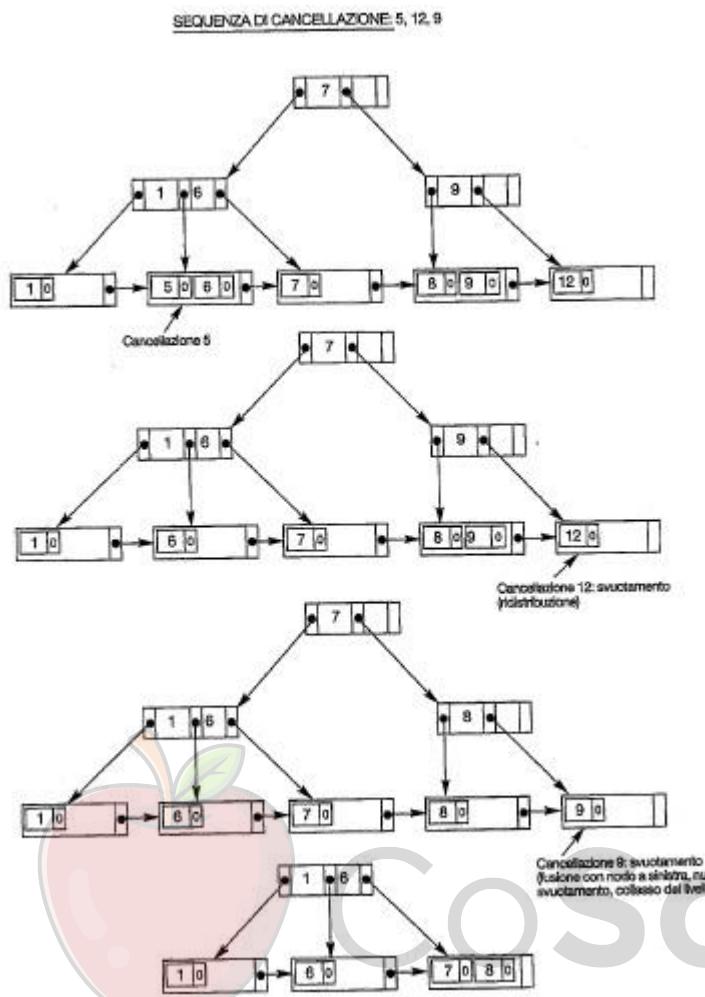
L'inserimento di 7 non provoca overflow. Si noti che tutti i valori sono a livello foglia, perché i puntatori ai dati sono tutti a quel livello.

L'inserimento di 3 provoca overflow nel nodo $(1, 5)$, che viene scisso, propagando il valore 3 nel nodo padre.

Si noti che un valore che compare nel nodo interno, compare anche come valore più a destra del sottoalbero referenziato dal puntatore alla sinistra di tale valore.

L'inserimento di 12 provoca un overflow nel nodo $(7, 8)$, che viene scisso, propagando il valore 8 nel nodo padre. La scissione si propaga fino alla radice, creando un nuovo livello nell'albero B^+ .

L'inserimento del 9 non provoca overflow.



Dato il seguente albero B^+ , di ordine $p = 3$ e $p_{\text{foglia}} = 2$, si vuol cancellare i record 5, 12, 9 e 6.

La cancellazione di 5 non pone alcun problema.

La cancellazione di 12 viene risolta con una ridistribuzione, e l'aggiornamento del valore del nodo interno da 9 a 8.

La cancellazione di 9 causa un underflow.

I puntatori del nodo padre vengono redistribuiti con quelli del fratello sinistro.

La cancellazione di 6 causa un underflow.

La redistribuzione dei nodi utilizzando i fratelli a destra e a sinistra causa un altro underflow nel nodo padre, che causa un underflow nel nodo radice, portando ad una riduzione del numero di livelli.

Gestione delle Transazioni

La **transazione** fornisce un meccanismo per descrivere le unità logiche di elaborazione delle basi di dati. I sistemi di gestione delle transazioni sono sistemi con grandi basi di dati e centinaia di utenti che eseguono transazioni contemporaneamente, come sistemi di prenotazione, sistemi bancari ecc...

È possibile classificare i database system in base al numero di utenti che possono utilizzare il sistema in **modo concorrente**:

- Un DBMS è **single-user** (*monoutente*) se al più un utente per volta può usare il sistema.
- Un DBMS è **multi-user** se più utenti possono usare il sistema concorrentemente.

Più utenti possono accedere al database simultaneamente grazie al concetto di **multiprogrammazione**, che consente ad un computer di elaborare più programmi o transazioni simultaneamente.

Quando si ha una sola CPU, necessariamente si elabora un processo alla volta; infatti l'illusione di avere più programmi che vengono eseguiti contemporaneamente è fornita dai *sistemi operativi multiprogrammati*. Tali sistemi eseguono alcune istruzioni da un programma, per poi sospenderlo ed eseguire altre istruzioni da altri programmi e così via. Quando un programma viene riattivato, esso riparte dal punto in cui era stato sospeso.

Su *sistemi monoprocesso*, l'esecuzione concorrente dei programmi è quindi intervallata (**interleaved**), l'interleaving ha luogo, in genere, quando un programma effettua operazioni di I/O che automaticamente lasciano la CPU inattiva. Su *sistemi multiprocesso*, invece, l'esecuzione dei programmi avviene realmente in **parallello**.

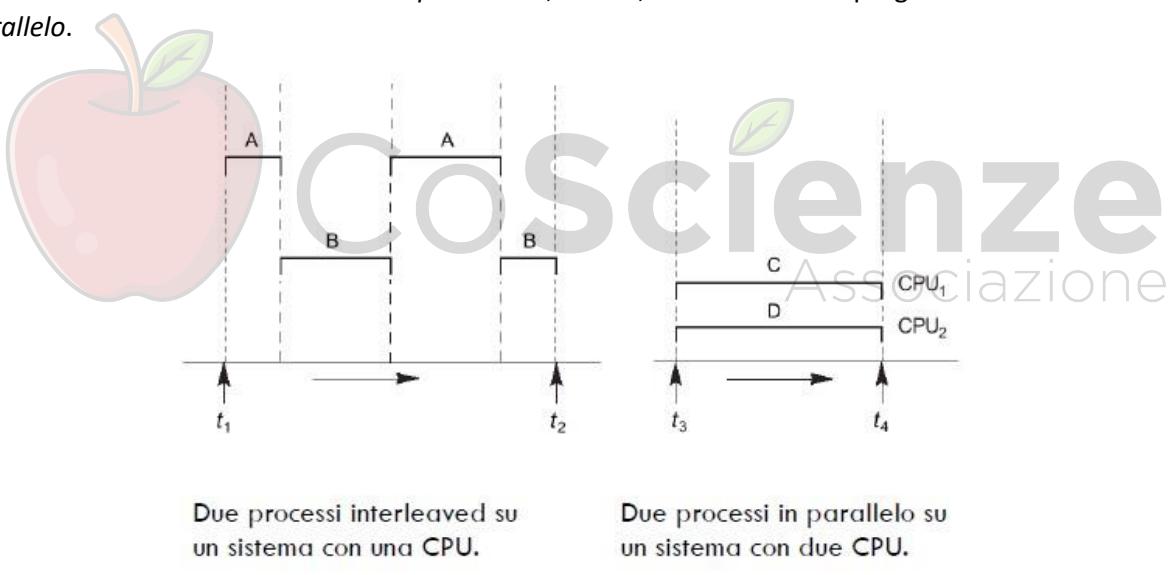


Figura 4.1 – Esempi di multiprogrammazione. A sinistra due processi in esecuzione ad intervalli. A destra, invece, due processi che vengono eseguiti in parallelo.

Le transazioni

Informalmente, una transazione è un insieme di operazioni che accedono al database, viste logicamente come un'istruzione singola ed indivisibile.

Per mostrare la gestione delle transazioni, verrà utilizzata una visione semplificata di un database system, dove un database è visto come una collezione di data item, ognuno con un nome. La dimensione del data item è detta **granularità**: può essere un singolo campo di un record, così come un intero blocco di un disco.

Con tale semplificazione, le **possibili operazioni** di accesso al database, che una transazione può effettuare sono:

- **Read_item(X)**: legge l'item di nome X in una variabile di programma. Per semplicità si assume che anche la variabile si chiami X.
- **Write_item(X)**: scrive il valore della variabile di programma X nell'elemento X del database.

La Read_item(X)

Sapendo che l'unità di trasferimento di dati è il blocco, i passi per eseguire un comando Read_item(X) sono:

1. Trovare l'indirizzo del blocco che contiene X.
2. Copiare tale blocco in un buffer in memoria centrale (*se tale blocco non è già in un buffer*).
3. Copiare l'elemento X dal buffer alla variabile di programma X.

La Write_item(X)

I passi per eseguire un comando Write_item(X) sono:

1. Trovare l'indirizzo del blocco che contiene X.
2. Copiare tale blocco in un buffer in memoria centrale (*se tale blocco non è già in un buffer*).
3. Copiare l'elemento X dalla variabile di programma di nome X nella sua locazione nel buffer.
4. Memorizzare il blocco aggiornato dal buffer al disco.

Controllo della concorrenza

Il controllo della concorrenza e dei meccanismi di recovery (recupero) riguardano principalmente i comandi di accesso al database in una transazione. Transazioni inviate da più utenti, che possono accedere e aggiornare gli elementi del database, vengono eseguite in modo concorrente. Se l'esecuzione concorrente non è controllata, si possono avere problemi di **database inconsistente** (due dati che rappresentano la stessa informazione hanno valori diversi, così facendo i dati non sono più affidabili). Di seguito viene mostrato un esempio di problemi con le transazioni:

Si supponga di avere un database per la prenotazione di posti in aereo, in cui è memorizzato un record per ogni volo.

Si supponga di avere due transazioni, T_1 e T_2 , dove la transazione T_1 cancella N prenotazioni da un volo il cui numero di posti occupati è memorizzato nell'item del database di nome X, e riserva lo stesso numero su un altro volo il cui numero di posti occupati è memorizzato nell'item di nome Y.

T_2 prenota M posti sul primo volo referenziato nella transazione T_1 .

Le due transazioni per la gestione dei posti:

(a) T_1

```
read_item (X);
X:=X-N;
write_item (X);
read_item (Y);
Y:=Y+N;
write_item (Y);
```

(b) T_2

```
read_item (X);
X:=X+M;
write_item (X);
```

Si osservino i possibili **problemi** che possono sorgere eseguendo T_1 e T_2 concorrentemente.

Figura 4.2 – Due transazioni in un sistema di prenotazione voli. (a) La transazione cancella N prenotazioni da un volo e prenota su un altro volo. (b) Prenota M posti sul primo volo referenziato da (a).

Aggiornamento perso

Il problema dell'aggiornamento perso: infatti supponendo che T_1 e T_2 siano avviate contemporaneamente e che le loro operazioni siano interleaved (interfogliate) dal sistema operativo nel modo seguente:

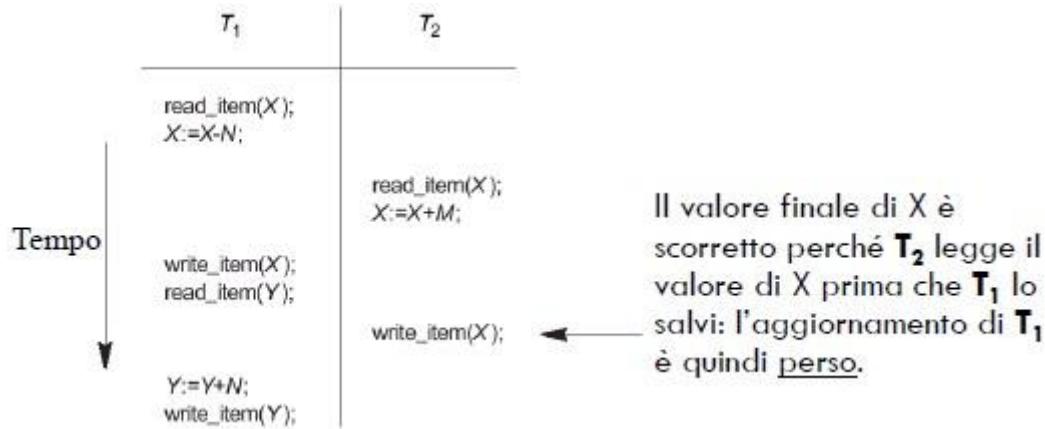


Figura 4.3 - Problema dell'aggiornamento perso di due transazioni interleaved.

Aggiornamento temporaneo (o lettura sporca)

Una transazione aggiorna un elemento ma poi essa fallisce per un qualche motivo, non riuscendo a salvare tale aggiornamento. L'elemento aggiornato è però letto da un'altra transazione prima che esso sia riportato al suo valore originario.

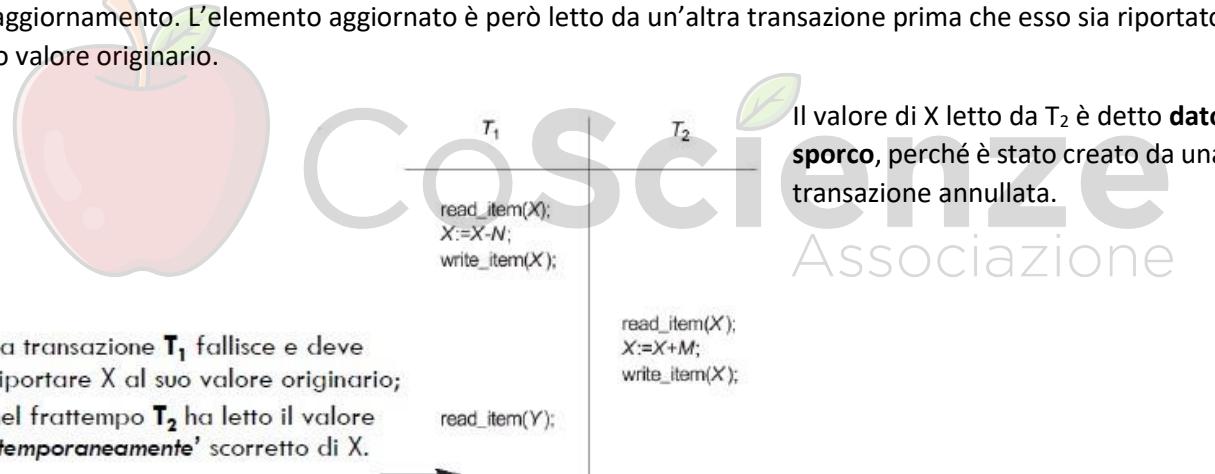


Figura 4.4 - Problema della lettura sporca di due transazioni interleaved.

Totalizzazione scorretta

Se una transazione sta calcolando una funzione di aggregazione su un certo insieme di record, mentre altre transazioni stanno aggiornando alcuni di tali record, la funzione può calcolare alcuni valori prima dell'aggiornamento ed altri dopo.

Una transazione T_3 sta calcolando il numero totale di prenotazioni su tutti i voli mentre T_1 è in esecuzione:

T_1	T_3
	<code>sum:=0; read_item(A); sum:=sum+A;</code>
	<code>⋮</code>
<code>read_item(X); X:=X-N; write_item(X);</code>	<code>⋮</code>
<code>read_item(Y); Y:=Y+N; write_item(Y);</code>	<p>Il risultato di T_3 è sbagliato, poiché T_3 legge il valore di X dopo che sono stati sottratti N posti, e prima che gli stessi siano sommati ad Y.</p>

Figura 4.5 - Problema della totalizzazione scorretta di due transazioni interleaved.

Letture non ripetibili

Tale problema avviene se una transazione T_1 legge due volte lo stesso item, ma tra le due letture una transazione T_2 ne ha modificato il valore. Ad esempio: durante una prenotazione di posti aerei, un cliente chiede informazioni su più voli. Quando il cliente decide, la transazione deve rileggere il numero di posti disponibili sul volo scelto per completare la prenotazione, ma potrebbe non trovare più la stessa disponibilità.

Tecniche di recovery

È necessario l'operazione di **recovery** perché quando viene inoltrata una transazione, il sistema deve far sì che: tutte le operazioni siano completate con successo ed il loro effetto sia registrato permanentemente nel database oppure la transazione annullata non abbia effetti né sul database né su qualunque altra transazione. Vi possono essere diverse failure, ed esse in genere vengono suddivise in fallimenti di transazione, di sistemi e di media. Le possibili ragioni di una *failure* sono:

1. Un **crash di sistema** durante l'esecuzione della transazione.
2. **Errore di transazione o di sistema** (overflow, divisione per zero, valori errati di parametri, ecc...).
3. **Errori locali o condizioni eccezionali** rilevati dalla transazione (ad esempio, i dati per la transazione possono non essere trovati o essere non validi, tipo un ABORT programmato a fronte di una richiesta di un prelievo da un fondo scoperto).
4. **Controllo della concorrenza**: il metodo di controllo della concorrenza può decidere di abortire la transazione perché viola la serializzabilità o perché varie transazioni sono in deadlock.
5. **Fallimento di disco**: alcuni blocchi di disco possono perdere i dati per un malfunzionamento in lettura o scrittura, o a causa di un crash della testina del disco.
6. **Problemi fisici e catastrofi** (fuoco, sabotaggio, furto, caduta di tensione, errato montaggio di nastro da parte dell'operatore, ecc...).

I problemi 1 - 4 sono i più frequenti, ed è più facile effettuarne il recovery.

Concetti sulle transazioni

Una transazione è **un'unità atomica di lavoro** che, o è completata nella sua interezza o è integralmente annullata. Per motivi di recovery, il sistema deve tenere traccia dell'inizio, della fine o dell'abort di ogni transazione. Il manager di recovery tiene quindi traccia delle seguenti operazioni:

- **BEGIN_TRANSACTION**: marca l'inizio dell'esecuzione della transazione.
- **READ** o **WRITE**: specifica operazioni di lettura o scrittura sul database, eseguite come parte di una transazione.
- **END_TRANSACTION**: specifica che le operazioni di READ e WRITE sono finite e marca il limite di fine esecuzione della transazione.
- **COMMIT_TRANSACTION**: segnala la fine con successo della transazione, in modo che qualsiasi cambiamento può essere reso permanente, senza possibilità di annullarlo.
- **ROLL-BACK** (o **ABORT**): segnala che la transazione è terminata senza successo e tutti i cambiamenti o effetti nel database devono essere annullati.

Operazioni addizionali:

- **UNDO**: simile al roll-back, eccetto che si applica ad un'operazione singola piuttosto che a una intera transazione.
- **REDO**: specifica che certe operazioni devono essere ripetute.

Stati di una transazione

Si può usare un diagramma degli stati per evidenziare come evolve lo stato di una transazione:

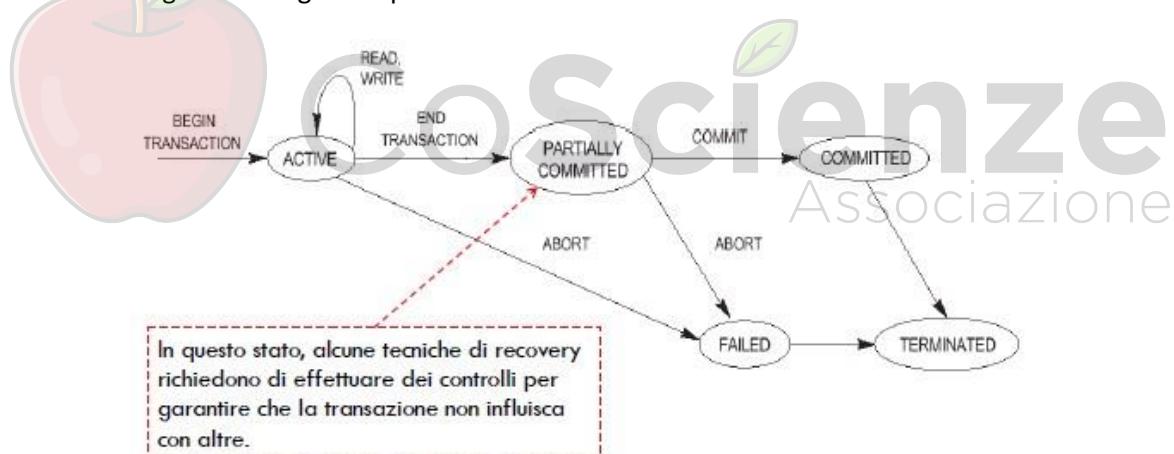


Figura 4.6 – Diagramma degli stati di una transazione.

Il System Log

Per effettuare il recovery di transazioni abortite, il sistema mantiene un **log** per tenere traccia delle operazioni che modificano il database. Il system log contiene quindi informazioni su tutte le transazioni che modificano i valori degli item del database. Si analizzano ora le entries del System Log: il log è strutturato come una lista di record. In ognuno di essi è memorizzato un **ID univoco** della transazione **T**, generato in automatico dal sistema. Alcuni tipi di entry possibili nel logo sono:

- **[start_transaction, T]**: la transazione T ha iniziato la sua esecuzione.
- **[write_item, T, X, old_value, new_value]**: la transazione T ha cambiato il valore dell'item X da old_value a new_value.
- **[read_item, T, X]**: la transazione T ha letto l'item X.
- **[commit, T]**: la transazione T è terminata con successo e le modifiche possono essere memorizzate in modo permanente.
- **[abort, T]**: la transazione T è fallita.

Scrittura forzata del log

Il file di log deve essere tenuto su disco; aggiornare il log implica copiare il blocco dal disco al buffer in memoria, aggiornare il buffer e riscrivere il buffer su disco. In caso di una failure, solo le entry su disco vengono usate nel processo di recovery. Poiché un blocco viene tenuto in memoria finché non è pieno, prima che una transazione raggiunga il punto di commit, ogni parte del log in memoria deve essere scritta (**scrittura forzata o force writing**).

Proprietà delle transazioni

Le transazioni dovrebbero possedere alcune proprietà (dette **ACID properties**, dalle loro iniziali):

- **Atomicità**: l'atomicità rappresenta il fatto che una transazione è un'unità *indivisibile* di esecuzione; o vengono resi visibili tutti gli effetti di una transazione, oppure la transazione non deve avere alcun effetto sulla base di dati, con un approccio "tutto o niente". In pratica non è possibile lasciare la base di dati in uno stato intermedio attraversato durante l'elaborazione della transazione. (*responsabilità del recovery Subsystem*).
- **Consistency preserving**: l'esecuzione di una transazione non viola i vincoli di integrità definiti sulla base di dati. Quando il sistema rileva che una transazione sta violando uno dei vincoli, esso interviene per annullare la transazione o per correggere la violazione del vincolo. Quindi una transazione deve far passare il database da uno stato consistente ad un altro (*responsabilità dei programmati*).
- **Isolamento**: l'isolamento richiede che l'esecuzione di una transazione sia indipendente dalla contemporanea esecuzione di altre transazioni, rendendo invisibili i vari aggiornamenti finché non è committed. In particolare, si richiede che il risultato dell'esecuzione concorrente di un insieme di transazioni sia analogo al risultato che le stesse transazioni otterrebbero qualora ciascuna di esse fosse eseguita da sola (*responsabilità del sistema per il controllo della concorrenza*).
- **Durability**: la persistenza richiede che l'effetto di una transazione che ha eseguito il commit correttamente non venga più perso. In pratica, una base di dati deve garantire che nessun dato venga perso per nessun motivo (*responsabilità del sistema di gestione dell'affidabilità*).

Gli Schedule

Informalmente, uno **schedule** è l'ordine in cui sono eseguite le operazioni di più transazioni processate in modo interleaved. Formalmente, uno schedule S di n transazioni T_1, T_2, \dots, T_n è un ordinamento delle operazioni delle transazioni, soggetto al vincolo che per ogni transazione T_i che partecipa in S , le operazioni in T_i in S devono apparire nello stesso ordine di apparizione in T_i . si supponga che l'ordinamento delle operazioni in S sia totale.

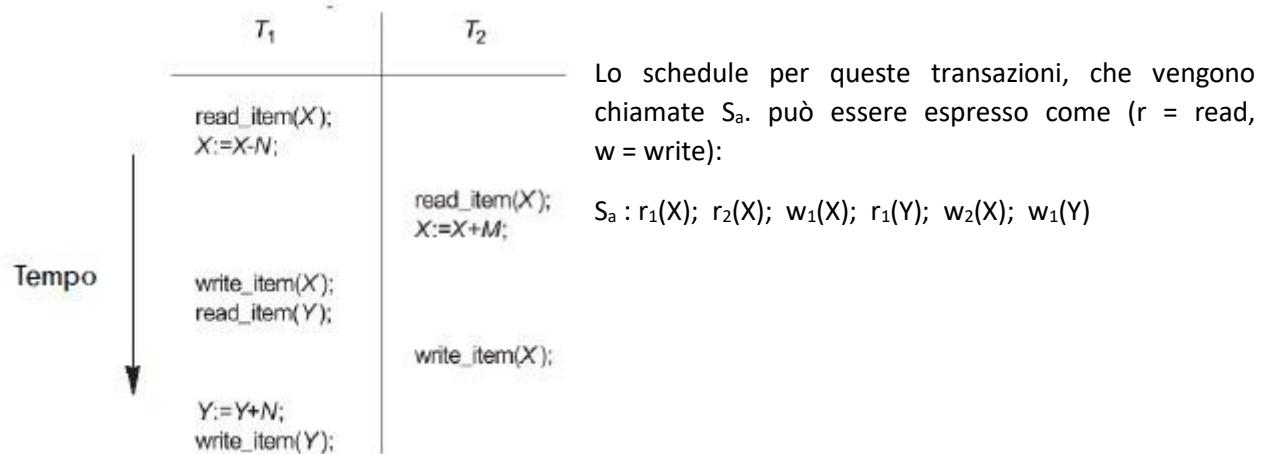


Figura 4.7 – Schedulazione di transazioni.

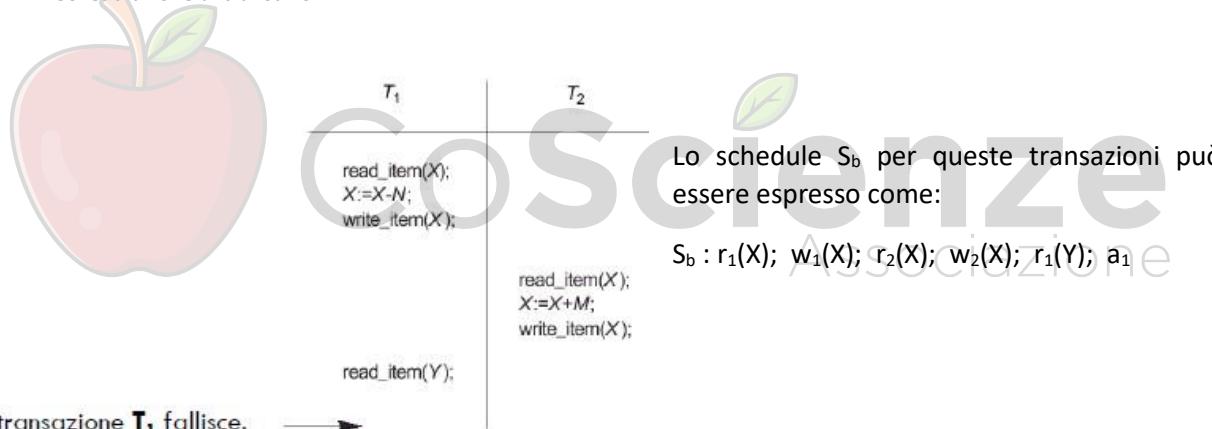
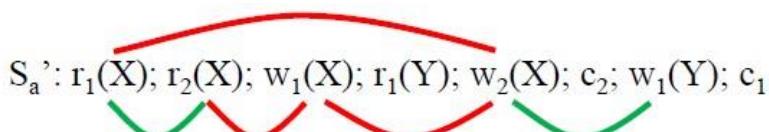


Figura 4.8 – Schedulazione di transazioni, ma con la transazione T_1 che fallisce.

Due operazioni in uno schedule sono in **conflitto** se:

1. Appartengono a differenti transazioni,
2. Accedono allo stesso elemento X ,
3. Almeno una delle due operazioni è una `write_item(X)`.



In **verde** le coppie che non generano conflitti e in **rosso** le coppie che generano conflitti.

Schedule completo

Uno schedule S di n transazioni T_1, T_2, \dots, T_n è uno **schedule completo** se valgono le seguenti condizioni:

1. Le operazioni in S sono esattamente quelle in T_1, T_2, \dots, T_n , inclusa un'operazione di commit o di abort come ultima operazione di ogni transazione in S.
2. Per ogni coppia di operazioni della stessa transazione T_i , il loro ordine di occorrenza in S è lo stesso che è in T_i .
3. Per ogni coppia di operazioni in conflitto, una deve occorrere prima dell'altra nello schedule.

Uno schedule completo non contiene transazioni attive, perché sono tutte committed o aborted.

Dato uno schedule S, si definisce **proiezione committed** $C(S)$, uno schedule che contiene solo le operazioni in S che appartengono a transazioni committed.

È importante caratterizzare i tipi di schedule in base alla possibilità di effettuare il recovery. Infatti, si vuol garantire che: *per una transazione committed non è mai necessario il roll-back*. Ed uno schedule con tale proprietà è detto **recoverable**.

Alternativamente, uno schedule S è detto recoverable se nessuna transazione T in S fa un commit, finché tutte le transazioni T', che hanno scritto un elemento letto da T, hanno fatto un commit. Si mostra di seguito un esempio di schedule recoverable.

Sia S_a' : $r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1$

Esso risulta essere recoverable, ma soffre del problema dell'aggiornamento perso.

Sia S_c' : $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1$

Esso non è recoverable perché T_2 legge X da T_1 e poi fa il commit prima di T_1 . Se T_1 abortisce dopo c_2 , allora il valore di X che T_2 legge non è più valido e T_2 deve essere abortito dopo aver fatto commit.

Se si modifica S_c da:

Sia S_c' : $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1$

a

$r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2$

S_c diviene uno schedule recoverable.

Roll-Back in cascata

Negli schedule recoverable nessuna transazione committed ha necessità di roll-back. Si possono però avere **roll-back in cascata** se una transazione non committed legge un dato scritto da una transazione fallita.

Uno schedule è detto **cascadeless (evitare il roll-back in cascata)** se ogni transazione nello schedule legge elementi scritti solo da transazioni committed. Viene mostrato un esempio di Cascadeless Schedule:

Sia S_e : $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2$

Esso non è cascadeless: T_2 legge X scritto da T_1 prima che T_1 raggiunga il commit o l'abort.

S_e' : $r_1(X); w_1(X); r_1(Y); w_1(Y); c_1; r_2(X); w_2(X); c_2$

Schedule stretti

Uno schedule è detto **stretto** se le transazioni non possono né leggere né scrivere un elemento X finché l'ultima transazione che ha scritto X non è completata (con *commit* o *abort*). Tali schedule stretti semplificano il processo di recovery poiché occorre solo ripristinare la **before image** (*old_value*) di un dato X. Ad esempio:

Sia $S_f: w_1(X, 5); w_2(X, 8); a_1$

Si supponga che inizialmente $X = 9$. S_f è cascadeless ma non stretto. Se T_1 fallisce, la procedura di recovery ripristina il valore di X a 9, anche se è stato modificato da T_2 .

Serializzabilità di schedule

Oltre a caratterizzare gli schedule in base alla possibilità di recovery, si vorrebbe poterli classificare anche in base al loro comportamento in ambiente concorrente. Uno schedule è **seriale** se per ogni transazione T nello schedule, tutte le operazioni di T sono eseguite senza interleaving. Altrimenti non risulta essere seriale.

Ad esempio: i due possibili casi di schedule seriali con due transazioni T_1 e T_2 :

- a) T_1 esegue prima di T_2
- b) T_1 esegue dopo T_2

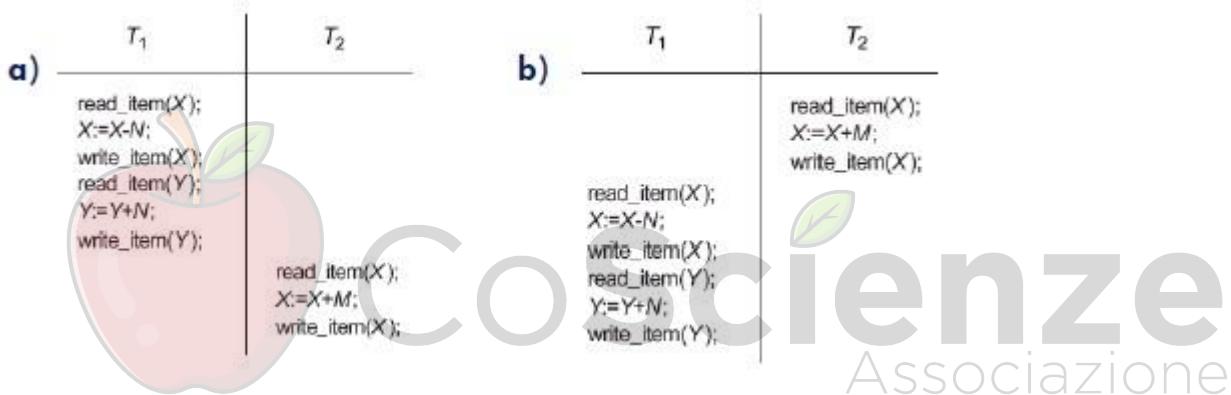


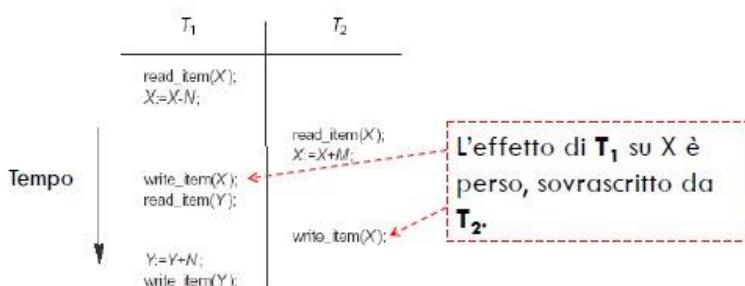
Figura 4.9 – Schedulazione seriale di transazioni.

Gli **schedule seriali** sollevano alcuni problemi, infatti essi limitano la concorrenza o le operazioni di interleaving:

- Se una transazione aspetta una operazione di I/O, non si può allocare la CPU ad un'altra transazione.
- Se una transazione T dura a lungo, le altre transazioni devono aspettare che finisca.

Gli schedule seriali, in pratica, sono **inaccettabili**.

Anche nel caso degli **schedule non seriali** vi possono essere problemi di aggiornamento perso, aggiornamento temporaneo, somma scorretta, ecc...



Esistono degli schedule non seriali che danno risultati corretti.

Figura 4.10 – Problemi con gli schedule non seriali.

Schedule serializzabili

Uno schedule S di n transazioni è **serializzabile** se è “equivalente” a qualche schedule seriale delle stesse n transazioni. Dati n schedule, abbiamo $n!$ possibili seriali. Quando due schedule possono essere detti “equivalenti”? Due schedule sono detti **result equivalent** se producono lo stesso stato finale del database. Non è una definizione accettabile: la *produzione dello stesso stato può essere accidentale*.

S_1	S_2	
<code>read_item(X); X:=X+10; write_item(X);</code>	<code>read_item(X); X:=X*1.1; write_item(X);</code>	I due schedule sono result equivalent solo se X = 100.

Figura 4.11 – Schedule equivalenti.

Una definizione più appropriata è quella di **conflict equivalent**: due schedule sono *conflict equivalent* se l’ordine di ogni coppia di operazioni in conflitto è lo stesso in entrambi gli schedule.

Ad esempio, in S_1 sono presenti $r_1(X)$; $w_2(X)$ e nello schedule S_2 sono in ordine inverso, $w_2(X)$; $r_1(X)$.

Il valore $r_1(X)$ può essere differente nei due schedule.

Uno schedule è **conflict serializable** se è *conflict equivalent* a qualche schedule seriale S' . È possibile determinare, per mezzo di un semplice algoritmo, la *conflict serializzabilità* di uno schedule. Molti metodi per il controllo della concorrenza non testano la serializzabilità, piuttosto utilizzano protocolli che garantiscono che uno schedule sarà serializzabile. Si mostra un esempio di conflict serializzabilità:

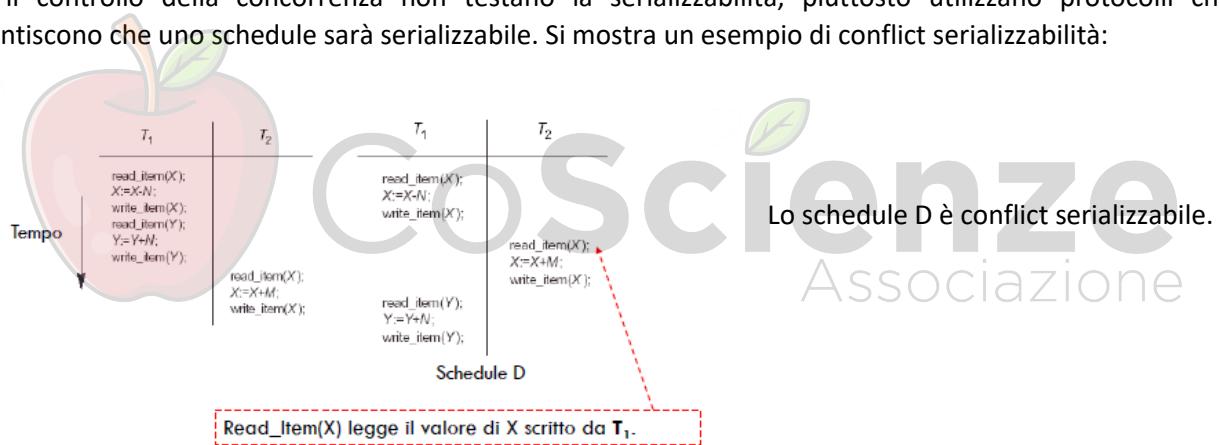


Figura 4.12 – Conflict serializzabilità.

Algoritmo per la serializzabilità

L’algoritmo cerca solo le operazioni di `read_item` e `write_item`, per costruire un **grafo di precedenza** (o **grafo di serializzazione**). Un grafo di precedenza è un grafo diretto:

- $G = (N, E)$, con un insieme di nodi: $N = \{T_1, T_2, \dots, T_n\}$ ed un insieme di archi $E = \{e_1, e_2, \dots, e_m\}$.

Ogni arco è della forma $(T_j \rightarrow T_k)$, con $1 \leq j, k \leq n$, ed è creato se un’operazione in T_j appare nello schedule prima di qualche operazione in conflitto in T_k .

Algoritmo 1:

1. Per ogni transazione T_i nello schedule S, creare un nodo T_i nel grafo;
2. Per ogni caso in S dove T_j esegue una $\text{read_item}(X)$ dopo che T_i esegue una $\text{write_item}(X)$, creare un arco $(T_i \rightarrow T_j)$ nel grafo;
3. Per ogni caso in S dove T_j esegue una $\text{write_item}(X)$ dopo che T_i esegue una $\text{read_item}(X)$, creare un arco $(T_i \rightarrow T_j)$ nel grafo;
4. Per ogni caso in S dove T_j esegue una $\text{write_item}(X)$ dopo che T_i esegue una $\text{write_item}(X)$, creare un arco $(T_i \rightarrow T_j)$ nel grafo;
5. Lo schedule è serializzabile se e solo se il grafo non contiene cicli.

Esempio 1

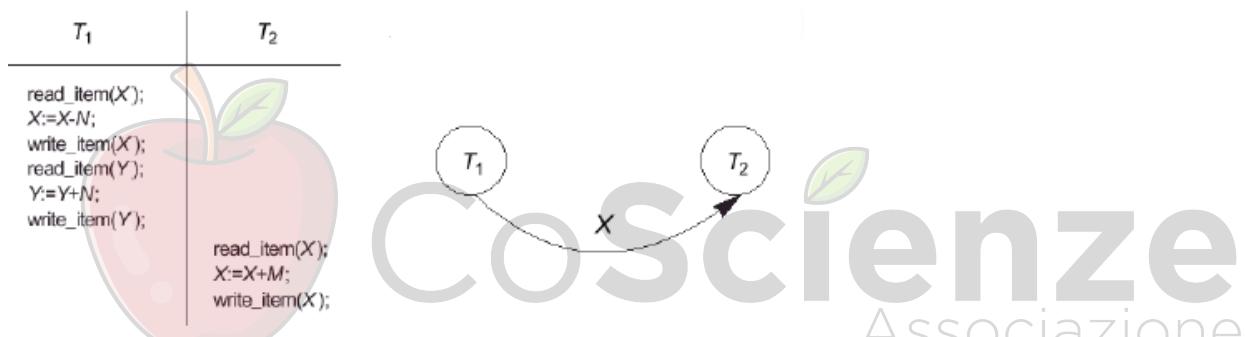


Figura 4.13 – Schedule seriale con due transazioni, ed il relativo grafo di precedenza.

Esempio 2

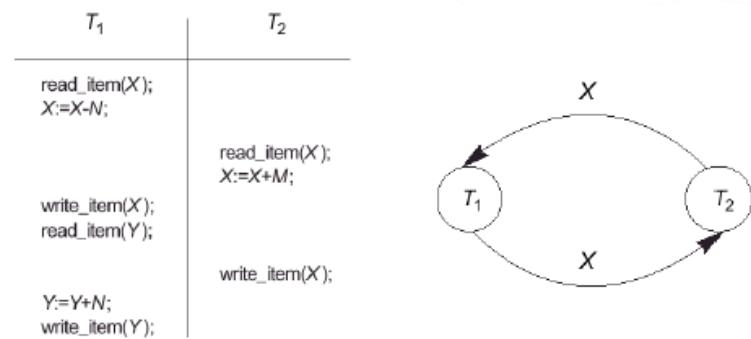


Figura 4.14 – Schedule seriale con due transazioni, ed il relativo grafo di precedenza che ne evidenzia la non serializzabilità.

Se non ci sono cicli nel grafo di precedenza relativo ad uno schedule S si può **creare uno schedule seriale equivalente S'** ordinando le transazioni che partecipano allo schedule come segue:

- Se esiste un arco fra T_i e T_j ; T_i deve apparire prima di T_j nello schedule seriale equivalente.

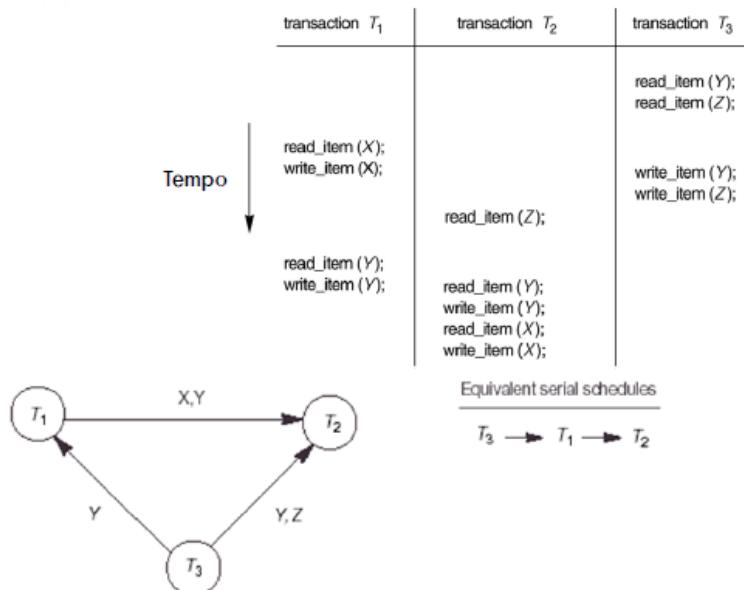


Figura 4.15 – Esempio di costruzione di uno schedule seriale.

Supporto alle transazioni in SQL

Il concetto di transazione in SQL è simile a quanto visto finora: *una transazione è una singola unità logica di lavoro con la proprietà dell'atomicità*. Di default, in SQL ogni singola istruzione è una transazione. Non esiste uno Statement di *Begin_Transaction*, poiché l'inizio di una transazione viene determinato implicitamente. Deve esserne però esplicitata la fine, con le istruzioni **COMMIT** o **ROLLBACK**.

Ogni transazione in SQL ha tre caratteristiche, specificate per mezzo dell'istruzione **SET TRANSACTION** che inizia una transazione:

- **Modalità di accesso.** Specifica se l'accesso ai dati è in sola lettura o in lettura/scrittura.
- **Dimensione dell'area diagnostica.** Specifica lo spazio da usare per informazioni all'utente sull'esecuzione delle transazioni.
- **Isolation Level.** Specifica la politica di gestione delle transazioni concorrenti.

È molto importante utilizzare due commit (uno prima e uno dopo). SET TRANSACTION deve essere la prima istruzione SQL di una transazione.

- Il COMMIT appena prima assicura che ciò sia vero.
- Il COMMIT alla fine rilascia le risorse possedute dalla transazione.

La *modalità di accesso* può essere specificata come **READ ONLY** o **READ WRITE** (di default).

- La modalità **READ WRITE** permette l'esecuzione di comandi di aggiornamento, inserimento, cancellazione e creazione.
- La modalità **READ ONLY** serve unicamente per il recupero di dati.

Alcune transazioni effettuano istruzioni di SELECT su diverse tabelle e dovranno vedere dati coerenti, dati che si riferiscono allo stesso istante di tempo. **SET TRANSACTION READ ONLY** specifica questo meccanismo più protetto di gestione dei dati.

Nessun comando può modificare i dati di un'area su cui vengono effettuate operazioni di SELECT attraverso questo tipo di transazioni.

L'opzione *dimensione dell'area di diagnosi* **DIAGNOSTIC SIZE n** specifica un valore intero *n*, che indica il numero di condizioni che possono essere mantenute contemporaneamente nell'area di diagnosi. Tali condizioni forniscono informazioni di feedback (*errori o eccezioni*) riguardo i comandi SQL eseguiti più di recente.

L'opzione *livello di isolamento* è specificata usando l'istruzione **ISOLATION LEVEL <isolamento>**. I possibili valori sono:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE (è il livello predefinito)

Oltre alle già viste violazioni di “*lettura sporche*” e “*lettura non ripetibile*”, usando SQL può sorgere il problema delle “*lettura fantasma*”. Ad esempio, si supponga che una transazione **T₁** legga una serie di righe basate su una condizione di WHERE. Se una transazione **T₂** aggiunge dei valori che soddisfano la condizione di WHERE, una riesecuzione di **T₁** vedrà delle righe nuove (*fantasma*), non presenti in precedenza.

Violazioni basate sui livelli di isolamento definiti in SQL



Livello di isolamento	Tipo di violazione		
	Lettura sporca	Lettura non ripetibile	Fantasma
READ UNCOMMITTED	SI	SI	SI
READ COMMITTED	NO	SI	SI
REPEATABLE READ	NO	NO	SI
SERIALIZABLE	NO	NO	NO

Transazione SQL: Esempio

```

EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
  READ WRITE
  DIAGNOSTICS SIZE 5
  ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (FNAME, LNAME,
                                SSN, DNO, SALARY)
  VALUES ('Robert', 'Smith', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLOYEE
  SET SALARY = SALARY * 1.1 WHERE DNO = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ...

```

La transazione produce prima l'inserimento di una nuova riga nella tabella EMPLOYEE e successivamente esegue l'aggiornamento dello stipendio di tutti gli impiegati che lavorano nel reparto 2.

In caso di errore in una qualsiasi istruzione SQL, l'intera transazione viene annullata (rollback).

Ogni valore di stipendio aggiornato viene ripristinato al suo valore precedente e la nuova riga viene rimossa.

Tecniche per il Controllo della Concorrenza

L'esecuzione di transazioni concorrenti senza alcun controllo può comportare svariati problemi al database, e perciò è necessario evitare che esse interferiscano fra di loro garantendo l'**isolamento**. Per favorire ciò si utilizzano delle tecniche di gestione delle transazioni, per garantire che il database sia sempre in uno stato consistente; tali tecniche garantiscono la serializzabilità degli schedule, usando particolari **protocolli**.

- **Tecniche di locking.** I data item sono bloccati per prevenire che transazioni multiple accedano allo stesso item concorrentemente.
- **Timestamp.** Un timestamp è un identificatore unico per ogni transazione, generato dal sistema. Un protocollo può usare l'ordinamento dei timestamp per assicurare la serializzabilità.

Un fattore importante è la **granularità**, infatti la granularità di un data item è la porzione del database rappresentata dal data item stesso. Esso può essere della dimensione che varia da un attributo ad un singolo blocco di un disco o anche un intero file, o un intero database.

Tecniche di Locking

Le tecniche di locking si basano sul concetto di "blocco" (*lock*) di un item. Un **lock** è una variabile associata ad un data item nel database, e descrive lo stato di quell'elemento rispetto alle possibili operazioni applicabili ad esso. I lock, quindi, sono un mezzo per sincronizzare l'accedo da parte di transazioni concorrenti agli elementi del database. Possono essere utilizzati diversi tipi di lock per il controllo della concorrenza, in particolare verranno esaminati:

- **Lock binari.** Risultano essere semplici ma molto restrittivi e **non vengono** molto usati nella pratica.
- **Lock shared/esclusivi.** Vengono usati molto nei DBMS commerciali e forniscono maggiori capacità di controllo e concorrenza.

Lock binari

Un lock binario, come dice il nome stesso, può assumere due valori (o stati): **locked** (o valore 1) e **unlocked** (o valore 0). A ciascun elemento X del database viene associato un distinto lock:

- Se $lock(X) = 1$, le operazioni del database non possono accedere all'elemento X.
- Se $lock(X) = 0$, si può accedere all'elemento X quando richiesto.

Le transazioni che usano lock binari devono contenere operazioni di *lock_item* e *unlock_item*.

Una transazione chiede di accedere a un elemento X con l'istruzione **lock_item(X)**: se $lock(X) = 1$, la transazione è forzata ad attendere altrimenti se così non fosse viene posto il $lock(X)$ a 1, ottenendo l'accesso all'elemento. Al termine dell'utilizzo di X, la transazione invia un'istruzione di **unlock_item(X)**, che pone $lock(X)$ a 0, permettendo l'accesso all'item ad altre transazioni.

Un lock binario rafforza la **mutua esclusione** di un data item. Le operazioni di *lock* e *unlock_item* devono essere implementate come **unità indivisibili** (*sezioni critiche*), nel senso che non è consentito alcun interleaving dall'avvio fino al termine dell'operazione di lock/unlock o all'intervento della transazione di una coda di attesa. Il DBMS dispone di un sottosistema di **lock manager** per seguire e controllare gli accessi ai lock.

Lock_Item

```

Lock_Item (X):
  B: if Lock(X) = 0
  then Lock(X) ← 1;
  else
    begin
      wait (until Lock(X)=0 e il lock manager
            seleziona la transazione);
      goto B;
    end
  
```

Il comando di wait è considerato fuori dalla operazione di lock_item:
altre transazioni che vogliono accedere a X si trovano nella stessa coda.

La transazione è messa in una coda di attesa per l'item X finché X viene sbloccato e la transazione ne ottiene l'accesso

Unlock_Item

```

Unlock_Item (X):
  Lock(X) ← 0;
  if qualche transazione è in attesa
  then sveglia una delle transazioni in attesa;
  
```

Per implementare un lock binario è necessario solo una variabile binaria LOCK associata ad ogni data item X del database. Ogni lock può essere visto come un record con tre campi:

< nome data item, LOCK, transazione >

Con associata una coda delle transazioni che stanno provando ad accedere all'elemento. Gli elementi che non sono nella lock table sono considerati non bloccati (*unlocked*). L'organizzazione della tabella è basata su *hash file*. Usando uno schema di lock binario, ogni transazione deve obbedire alle seguenti regole:

1. Una transazione T deve impartire l'operazione di *lock_item(X)* prima di eseguire una *Read_item(X)* o *Write_item(X)*.
2. Una transazione T deve impartire l'operazione di *unlock_item(X)* dopo aver completato tutte le operazioni di *Read_item(X)* e *Write_item(X)*.
3. Una transazione T non impartirà un *lock_item(X)* se già vale il lock sull'elemento X.
4. Una transazione T non impartirà un *unlock_item(X)* a meno che non valga già un lock sull'elemento X.

Al più una transazione può mantenere il lock su un elemento X; vale a dire che due transazioni non possono accedere allo stesso elemento concorrentemente.

Lock shared/esclusivi

Il lock binario è troppo restrittivo, poiché l'accesso ad un data item è consentito ad una sola transazione per volta, è possibile consentire l'accesso in sola lettura a più transazioni contemporaneamente. Se una transazione deve scrivere un data item X, deve avere un **accesso esclusivo** su X. Per questo motivo si utilizza un *multiple mode lock*, cioè un lock che può avere più stati.

In questo caso le operazioni di lock diventano tre:

- Read_lock(X)
- Write_lock(X)
- Unlock(X)

Ed un lock ha tre possibili stati:

- Read_locked (share locked)
- Write_locked (exclusive locked)
- Unlocked

Ciascuna delle tre operazioni, Read_lock(X), Write_lock(X), Unlock_item(X), deve essere considerata indivisibile: nessun interleaving deve essere consentito dall'inizio dell'operazione fino al completamento o all'inserimento della transazione in una coda di attesa per quell'elemento.

Una possibile implementazione dei lock shared/esclusivi è che ogni lock è rappresentato un record con quattro campi:

< Nome data item, Lock, Numero di read, Transazione/i bloccante/i >

Dove Lock assume un valore che permette di distinguere tra Read_locked, Write_locked e Unlocked. Per risparmiare spazio, il sistema mantiene nella *lock table* i record per gli elementi locked.

Read_Lock (X)

Read_Lock(X):

```
B: if LOCK(X) = "unlocked"
    then begin
        LOCK(X) ← "read_locked";
        numero_di_read (X) ← 1;
    end;
else
    if LOCK(X) = "read_locked"
    then numero_di_read (X) = numero_di_read (X) + 1;
    else
        begin
            wait (until LOCK(X) = "unlocked" and il gestore di
            lock sceglie la transazione);
            goto B;
        end;
end;
```

Write_Lock(X)

Write_Lock(X):

```
B: if LOCK(X) = "unlocked"
    then LOCK(X) ← "write_locked";
else
    begin
        wait (until LOCK(X) = "unlocked" e il
        gestore di lock sceglie la transazione);
        goto B;
    end;
```

Unlock(X)

Unlock(X):

```
if LOCK(X) = "write_locked"
then
begin
    LOCK(X) ← "unlocked";
    sveglia una delle transazioni in attesa se ne esistono;
end;
else if LOCK(X) = "read_locked"
then
begin
    numero_di_read (X) = numero_di_read (X) - 1;
    if numero_di_read (X) = 0
    then
        begin
            LOCK(X) = "unlocked";
            sveglia una delle transazioni in attesa se ne esistono;
        end
    end;
end;
```

Regole per lock shared/esclusivi

Usando uno schema di shared/exclusive, ogni transazione deve obbedire alle seguenti regole:

1. Una transazione T deve impartire l'operazione di Read_Lock(X) o Write_Lock(X) prima di eseguire una Read_item(X).
2. Una transazione T deve impartire l'operazione di Write_Lock(X) prima di eseguire una Write_item(X).
3. Una transazione T deve impartire l'operazione di Unlock(X) dopo aver completato tutte le operazioni di Read_item(X) o Write_item(X).
4. Una transazione T non impartirà un Read_Lock(X) se già è in possesso di un lock condiviso in lettura o lock esclusivo in scrittura sull'elemento X.
5. Una transazione T non impartirà un Write_Lock(X) se già vale il lock in lettura o scrittura sull'elemento X.
6. Una transazione T non impartirà un Unlock(X) a meno che non valga già un lock sull'elemento X.

I vincoli 4 e 5 possono essere tralasciati per permettere **conversioni di lock**.

Conversioni di Lock

Una transazione può invocare un Read_Lock(X) e poi successivamente **incrementare** il lock, invocando un Write_Lock(X). Tale conversione è possibile solo se T è l'unica transazione che ha un Read_Lock su X, altrimenti deve aspettare. È possibile anche **decrementare** un lock, se una transazione T invoca una Write_Lock(X) e successivamente una Read_Lock(X).

Per permettere tali conversioni, è necessario che sia mantenuto un identificatore della transazione nella struttura del record per ciascun lock. È ovviamente necessario modificare le operazioni di Read_Lock, Write_Lock e Unlock per supportare l'informazione aggiuntiva.

Lock e serializzabilità

Lock binari e multiple-mode non garantiscono la serializzabilità degli schedule. Infatti:

T ₁	T ₂	
read_lock(Y); read_item(Y); unlock(Y);	read_lock(X); read_item(X); unlock(X);	Dati i valori iniziali X=20 e Y=30;
write_lock(X); read_item(X); X:=X+Y; write_item(X); unlock(X);	write_lock(Y); read_item(Y); Y:=X+Y; write_item(Y); unlock(Y);	Se T ₁ è seguito da T ₂ : X=50, Y=80 Se T ₂ è seguito da T ₁ : X=70, Y=50

Esempio di schedule seriali.

T ₁	T ₂	
read_lock(Y); read_item(Y); unlock(Y);	read_lock(X); read_item(X); unlock(X); write_lock(Y); read_item(Y); Y:=X+Y; write_item(Y); unlock(Y);	Dati i valori iniziali X=20 e Y=30; Risultato dello schedule S: X=50, Y=50
write_lock(X); read_item(X); X:=X+Y; write_item(X); unlock(X);		Y è stato sbloccato troppo presto!

Esempio di schedule non serializzabile.

Occorre un **protocollo** per stabilire il posizionamento delle operazioni di lock/unlock in ogni transazione.

Protocollo Two-Phase Locking

Una transazione T segue il protocollo **Two-Phase Locking** (2PL) se tutte le operazioni di locking (Read_lock, Write_Lock) precedono la prima operazione di Unlock nella transazione. Una transazione del genere può essere divisa in due fasi:

1. Expanding phase (espansione).
2. Shrinking phase (contrazione).

Nella **expanding phase**, possono essere *acquisiti nuovi lock* su elementi ma nessuno può esserne rilasciato. Invece, nella **shrinking phase** i *lock esistenti possono essere rilasciati* ma non possono essere acquisti nuovi lock.

Se la conversione di lock è permessa, l'upgrading deve essere fatto durante la fase di espansione ed il downgrading durante la contrazione. Di seguito un esempio di transazioni 2PL:

Le transazioni T_1 e T_2 viste in precedenza non seguono il protocollo 2PL. Quindi vengono riscritte come T'_1 e T'_2 :

T_1	T'_1	T_2	T'_2
<pre>read_lock(Y); read_item(Y); unlock(Y); write_lock(X); read_item(X); X:=X+Y; write_item(X); unlock(X);</pre>	<pre>read_lock(Y); read_item(Y); write_lock(X); unlock(Y); read_item(X); X:=X+Y; write_item(X); unlock(X);</pre>	<pre>read_lock(X); read_item(X); unlock(X); write_lock(Y); read_item(Y); Y:=X+Y; write_item(Y); unlock(Y);</pre>	<pre>read_lock(X); read_item(X); unlock(X); write_lock(Y); unlock(X); read_item(Y); Y:=X+Y; write_item(Y); unlock(Y);</pre>

2PL e serializzabilità

È dimostrabile che se **ogni** transazione in uno schedule segue il protocollo 2PL, allora lo schedule è serializzabile. 2PL può però **limitare la concorrenza** in uno schedule: la garanzia della serializzabilità viene pagata al costo di non consentire alcune situazioni di concorrenza possibili, poiché alcuni elementi possono essere bloccati più del necessario, finché la transazione necessita di effettuare letture e scritture.

2PL Conservativo

Il protocollo 2PL appena visto è detto **2PL di base**. Una variazione del 2PL è nota come **2PL conservativo** (o **statico**). Esso richiede che una transazione, prima di iniziare, blocchi tutti gli elementi a cui accede, pre-dichiarando i propri **read_set** e **write_set**:

- Il **read_set** è l'insieme di tutti i data item che saranno *letti* dalla transazione.
- Il **write_set** è l'insieme di tutti i data item che saranno *scritti* dalla transazione.

Se qualche data item dei due insiemi non può essere bloccato, la transazione resta in attesa finché tutti gli elementi necessari non divengono disponibili. Il 2PL conservativo è un protocollo **deadlock-free**.

Non viene usato nella pratica perché è necessario pre-dichiarare il read-set ed il write-set (cosa difficile in molte situazioni).

2PL Stretto

La variazione più diffusa del protocollo 2PL è il **2PL stretto**, che garantisce *schedule stretti*, in cui le transazioni non possono né scrivere né leggere un elemento X finché l'ultima transazione che ha scritto X non termina (con commit o abort). Nel 2PL stretto, quindi, una transazione non rilascia nessun lock esclusivo finché non termina, ed esso non risulta essere deadlock-free.

Generazione automatica di richieste di read e write lock

In molti casi il **sottosistema per il controllo della concorrenza** genera automaticamente le richieste di lock:

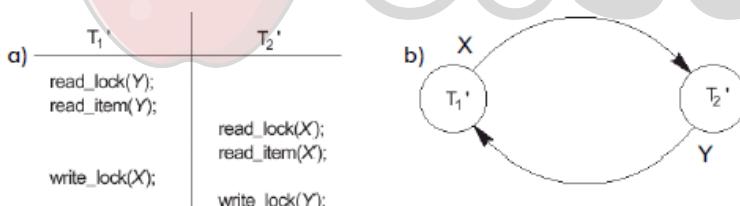
- Quando la transazione effettua una `Read_item(X)`, il sistema genera una operazione `Read_Lock(X)`.
- Quando la transazione effettua una `Write_item(X)`, il sistema genera una operazione `Write_Lock(X)`.

Se T invoca un `Read_item(X)`, il sistema invoca un `Read_Lock(X)` per T. Se lo stato di `LOCK(X)` = "write_locked" da una T', il sistema pone T nella coda di attesa per X, altrimenti esegue il `Read_Lock(X)` e quindi l'operazione di `Read_item(X)` per T.

Se T invoca un `Write_item(X)`, il sistema invoca un `Write_Lock(X)` per T. Se lo stato di `LOCK(X)` = "write_locked" OR "read_locked" da una T', il sistema pone T nella coda di attesa per X. Se lo stato di `LOCK(X)` = "read_locked" dall'unica transazione T, il sistema promuove il lock a "write_locked" ed esegue l'operazione di `Write_item(X)` per T. Se lo stato di `LOCK(X)` = "unlocked", il sistema esegue la `Write_Lock(X)` e quindi l'operazione di `Write_item(X)` per T.

Il protocollo di lock a due fasi garantisce la serializzabilità, ma non consente tutti i possibili schedule serializzabili; cioè alcuni schedule serializzabili vengono vietati da protocollo. Essi possono causare **deadlock e starvation**.

Si ha un **deadlock** quando due o più transazioni aspettano qualche item bloccato da altre transazioni T' in un insieme. Ogni transazione T' è in una coda di attesa e aspetta che un elemento sia rilasciato da un'altra transazione in attesa. Per prevenire il deadlock, occorre usare un protocollo apposito (**deadlock prevention protocol**).



Esempio di deadlock:

- a) Uno schedule di T_1' e T_2' in deadlock.
- b) Il grafo wait-for (delle attese) corrispondente.

Il protocollo a prevenzione di deadlock usato nel 2PL conservativo richiede che ogni transazione blocchi tutti i data item di cui ha bisogno in anticipo; se qualcosa non può essere ottenuta, nessun elemento è bloccato per cui la transazione aspetta e riprova in seguito. Lo *svantaggio* è la limitazione della concorrenza.

Sono stati proposti molti altri schemi per la prevenzione di deadlock:

- Basati su timestamp.
- Senza timestamp:
 - Algoritmo non-waiting.
 - Algoritmo cautious waiting.
- Basati su time-out, dove T richiede un lock aspettando fino ad un certo **time-out**. Se il lock non viene assegnato in tempo, T subisce un abort, esegue un rollback e ricomincia (*indipendentemente dalla presenza di deadlock*).

Prevenzione di deadlock basata su timestamp

Il **Timestamp TS(T)** di una transazione T è un identificatore unico associato ad ogni transazione. Un timestamp si basa sull'ordine di partenza delle transazioni: Se T_1 inizia prima di T_2 , allora

$$TS(T_1) < TS(T_2)$$

T_i prova a bloccare X che è bloccato da T_j . Schema **wait-die**:

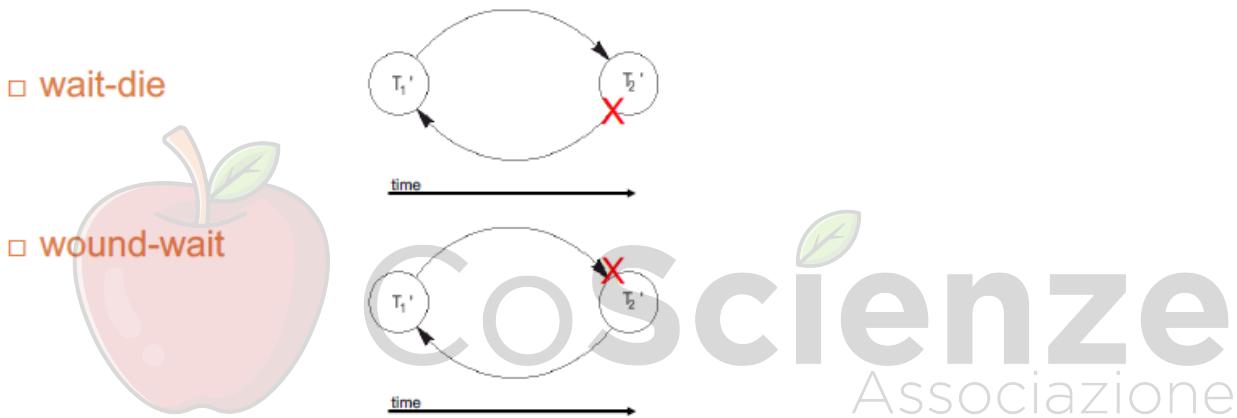
se $TS(T_i) < TS(T_j)$, (T_i più vecchia) allora T_i aspetta, altrimenti (T_i più giovane) T_i viene abortita (**abort**) e ripartirà con lo stesso timestamp.

Con lo schema **wound-wait**:

se $TS(T_i) < TS(T_j)$, (T_i più vecchia) allora T_j fallisce (**preempt**) e viene riavviata successivamente con lo stesso timestamp, altrimenti (T_i più giovane) T_i aspetta.

Entrambi gli schemi fanno fallire la transazione **più giovane**. Sono deadlock-free, ma possono causare l'abort di transazioni senza necessità (*non provocherebbero realmente un deadlock*).

- T'_1 prova a bloccare X che è bloccato da T'_2 .
- T'_2 prova a bloccare Y che è bloccato da T'_1 .



Schemi senza timestamp

No-waiting (NG): Se una transazione T non è in grado di ottenere un lock viene immediatamente interrotta e successivamente fatta ripartire dopo un intervallo di tempo, senza effettuare controlli sulla possibilità che si possa verificare un deadlock. Nessuna T aspetta, quindi non esiste la possibilità di deadlock. Questo metodo, però, non è pratico e può causare la terminazione e il riavvio di transazioni inutilmente.

Cautious Waiting (CW): Se T_i cerca un lock sull'item X ma non è in grado in quanto X è bloccato da T_k : Se T_k non è in attesa di qualche altro item bloccato, allora T_i aspetta, altrimenti T_i viene terminata.

Deadlock detection

Un approccio più pratico risulta essere quello del Deadlock Detection, dove *il sistema controlla l'esistenza di un deadlock*. Per riconoscere uno stato di deadlock, si utilizza il **grafo wait_for** (delle attese):

- Si crea un nodo per transazione in esecuzione.
- Si aggiunge un arco tra il nodo T_i e il nodo T_j , se T_i aspetta di bloccare un elemento usato da T_j .
- Si cancella un arco tra T_i e T_j appena l'elemento richiesto da T_i viene allocato a T_i .

Se c'è un **ciclo nel grafo**, si è in uno stato di deadlock.

Una volta riconosciuto un deadlock, si sceglie quale transazione abortire, usando un **criterio di selezione della vittima**.

L'algoritmo che seleziona la vittima in genere evita di scegliere transazioni che:

- Sono in esecuzione da molto tempo.
- Hanno effettuato molti aggiornamenti.

La **deadlock detection** è una soluzione valida quando non ci sono molte interferenze tra le transazioni. Le transazioni sono brevi ed ognuna di essa blocca pochi elementi, in alternativa per transazioni lunghe si usa il **deadlock prevention**.

Starvation

La **starvation** è un altro problema che può sorgere con l'utilizzo dei lock. Una transazione è nello stato di starvation se non può procedere per un tempo indefinito mentre altre transazioni nel sistema continuano normalmente. La causa è nello schema di waiting non sicuro che dà la precedenza ad alcune transazioni invece di altre.

Un possibile schema di waiting sicuro (*safe*) usa una coda **first-come-first-serve** (FCFS) dove le transazioni bloccano gli elementi rispettando l'ordine con cui hanno richiesto il lock.

Un altro schema è basato su **priorità**, che aumenta proporzionalmente al tempo atteso dalla transazione. Starvation si può avere anche negli algoritmi per il trattamento del deadlock, se l'algoritmo seleziona ripetutamente la stessa transazione come vittima. Per porre una **soluzione** a ciò, l'algoritmo può usare priorità più alte per transazioni che sono state abortite più volte.

Gli schemi **wait-die** e **wound-wait** escludono la starvation.

Granularità degli item

Tutte le tecniche per il controllo della concorrenza assumono che il database sia formato da un insieme di data item. Un data item può essere:

- Un record del database.
- Un campo di un record del database.
- Un blocco del disco.
- Un intero file.
- L'intero database.

La dimensione di un data item è detta **granularità di un data item**. Tale granularità può essere:

- **Fine**: riferita a data item di piccole dimensioni come un campo di un record.
- **Grossa**: riferita a data item di dimensioni maggiori come file o database intero.

La granularità **influenza le prestazioni** nel controllo della concorrenza e del recovery. Maggiore è il livello di granularità, minore è la concorrenza permessa.

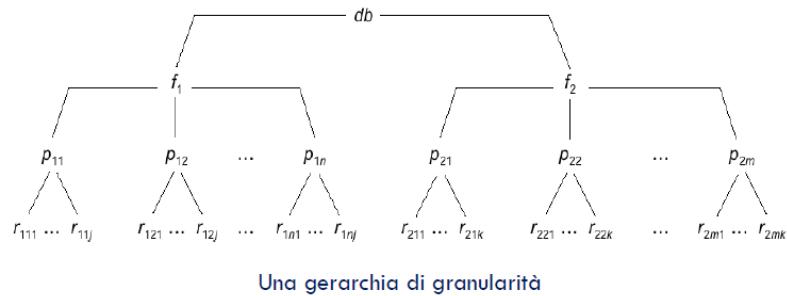
Se la dimensione di un data item è un blocco del disco, una transazione che necessita di leggere un record X in un blocco B effettuerà un lock dell'intero blocco. Altre transazioni, interessate a record diversi da X ma ugualmente in B, resteranno quindi inutilmente in attesa.

Per contro, a un data item di dimensioni inferiori corrisponde un numero maggiore di item nel database: ci sarà quindi una quantità superiore di lock ed il lock manager introdurrà un overhead nel sistema a causa delle molte operazioni che dovrà gestire. Sarà richiesto molto spazio per gestire la tabella dei lock.

La taglia migliore dipende dal tipo di transazioni coinvolte. Se una transazione tipica accede a:

- Un piccolo numero di record, è vantaggioso avere una granularità di un record.
- Molti record nello stesso file, è vantaggioso avere una granularità a livello blocco o a livello file.

La soluzione è nella possibilità di definire **granularità multiple**: un lock può essere chiesto su item a qualsiasi livello di granularità.



Tecniche di Recovery

Se viene sottomessa una transazione T, possono esserci due possibilità:

1. O tutte le operazioni di T sono completate ed il loro effetto è registrato permanentemente nel database.
2. Oppure T non ha nessun effetto né sul database né sulle altre transazioni (*failure*).

Nel secondo caso il sistema non può permettere che alcune operazioni di T siano portate a termine ed altre no.

Effettuare un recovery (o recupero) da una transazione fallita significa **ripristinare** il database al più recente stato *consistente* appena prima del failure (o guasto). Per fare ciò, il sistema deve tener traccia dei cambiamenti causati dall'esecuzione di transazioni sui dati. Tali informazioni sono memorizzate nel **system log**. Il log è strutturato come una lista di record; in ogni record è memorizzato un **ID univoco** della transazione **T**, generato in automatico dal sistema. Tipi di entry possibili nel log:

- [start_transaction, T]
- [write_item, T, X, old_value, new_value]
- [read_item, T, X]
- [commit, T]
- [abort, T]

Esistono varie **strategie di recovery**, le tipiche:

1. Se il danno è notevole, a causa di un *failure catastrofico* (ad esempio il crash di un disco), si ripristina una precedente copia di back-up da una memoria di archivio e si ricostruisce lo stato più vicino a quello corrente riapplicando (*redo*) tutte le transazioni completate (*committed*) salvate nel log fino al momento del failure.
2. Se, a seguito di un *failure non catastrofico*, il database non è danneggiato fisicamente, ma è solo in uno stato inconsistente, si effettua l'*UNDO* delle operazioni che hanno causato l'inconsistenza. Eventualmente si effettua il *REDO* di alcune operazioni. Non è necessario effettuare il restore del database, poiché basta la consultazione del system log.

Tecniche di recovery da failure non catastrofici

Concettualmente si possono distinguere due tecniche principali per il recovery da failure non catastrofici:

- **Tecniche ad aggiornamento differito** (*deferred update*) – Algoritmo NO-UNDO/REDO.
- **Tecniche ad aggiornamento immediato** (*immediate update*) – Algoritmo UNDO/REDO.

Tecniche ad aggiornamento differito

Con questa tecnica, i dati **non** sono fisicamente aggiornati fino all'esecuzione del commit di una transazione. Le modifiche effettuate da una transazione sono memorizzate in un suo spazio di lavoro locale (*workspace* o *buffer*). Durante il commit, gli aggiornamenti sono salvati persistentemente prima nel log e poi nel database. Se la transazione fallisce, non è necessario l'*UNDO*; può però essere necessario il *REDO* di alcune operazioni (commit con scrittura nel log ma non nel database).

Tecniche ad aggiornamento immediato

Il database può essere **aggiornato fisicamente** prima che la transazione effettui il commit. Le modifiche sono registrate prima nel log (con un force-writing) e poi sul database, permettendo comunque il recovery. Se una transazione fallisce dopo aver effettuato dei cambiamenti, ma prima del commit, l'effetto delle sue operazioni nel database deve essere annullato, occorre quindi effettuare il *rollback* della transazione (UNDO); può però essere necessario il REDO di alcune operazioni (commit con scrittura nel log ma non nel database).

Caching dei blocchi

Le tecniche di recovery possono essere influenzate da particolari gestioni del file system da parte del sistema operativo, in particolare dal **buffering** e dal **caching** di blocchi del disco in memoria centrale.

Per migliorare l'efficienza degli accessi a disco, i blocchi del disco contenenti i dati manipolati spesso dal DBMS, sono conservati (*cached*) in un buffer della memoria centrale. I dati sono quindi aggiornati in memoria, prima di essere riscritti su disco.

Sebbene la gestione del caching sia un compito del sistema operativo, spesso è il DBMS ad occuparsene esplicitamente, a causa dello stretto accoppiamento con le tecniche di recovery. Il DBMS gestisce direttamente una serie di buffer, che formano la **cache del DBMS**.

Per tenere traccia dei data item presenti in cache, si usa una **directory**, ovvero una tabella con entry del tipo:

<indirizzo pagina su disco, posizione nel buffer>

Accesso ai dati con cache

Quando il DBMS richiede l'accesso ad un data item, se ne controlla la presenza in cache. Se risulta essere già presente, il DBMS ne ottiene l'accesso, altrimenti se non è presente:

1. Deve essere individuato il blocco su disco contenente l'item.
2. Il blocco deve essere copiato nella cache.
3. Se la cache è piena, sono necessarie strategie tipiche dei sistemi operativi per il rimpiazzamento delle pagine (LRU – least recently used, FIFO – first in first out, ecc...).

Ad ogni blocco nella cache si può associare un **dirty bit**, per evidenziare se qualche elemento del buffer è stato modificato o meno. Al caricamento di un blocco in un buffer, il suo dirty bit è posto a 0. In seguito ad una modifica del contenuto del buffer, il suo dirty bit è posto a 1. Un buffer deve essere salvato su disco solo se il suo dirty bit vale 1.

Strategie per lo svuotamento della cache

Esistono due strategie principali per lo svuotamento di buffer modificati:

1. **In-place updating**: il buffer è riscritto nella stessa posizione originaria sul disco. Si mantiene in cache una singola copia di ogni blocco del disco. È necessario usare il system log per il recovery.
2. **Shadowing**: un buffer può essere riscritto in una locazione differente, permettendo la presenza su disco di più versioni di un data item. Sia il vecchio valore (BFIM – *before image*), sia quello aggiornato (AFIM, *after image*) di un data item possono essere presenti sul disco contemporaneamente.

Utilizzando l'*In-place updating* è necessario il system log per il recovery. Il log contiene due tipi di informazioni: quelle per l'UNDO e quelle per il REDO.

Un'entry del log di tipo UNDO include il vecchio valore (BFIM) dell'item salvato (necessario per effettuare un UNDO).

Un'entry del log di tipo REDO include il nuovo valore (AFIM) dell'item salvato (necessario per effettuare un REDO).

Protocollo WAL (Write-Ahead Logging)

Per permettere il recovery con l'In-place updating, le entry appropriate devono essere salvate nel log su disco prima di applicare i cambiamenti del database. In questo protocollo:

1. L'AFIM non può **sovrascrivere** la BFIM di un elemento sul disco finché non sono stati memorizzati i record di log di tipo UNDO della transazione.
2. L'operazione **commit** non può essere completata finché non sono scritti su disco tutti i record di log di tipo UNDO e di tipo REDO.

CheckPoint nel Log

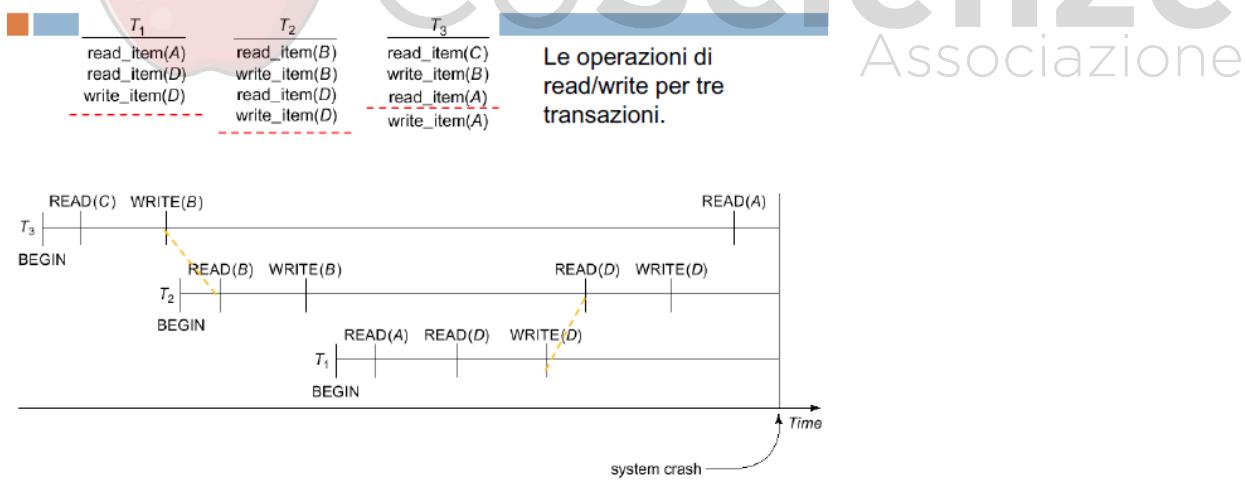
Nel log vengono utilizzate particolari entry, dette **checkpoint**. Nel log viene registrato un **[checkpoint]** periodicamente, quando il DBMS salva su disco tutti i blocchi modificati in cache. In questo modo, tutte le transazioni che hanno effettuato la **commit prima del checkpoint** non richiedono operazioni di REDO in caso di crash, poiché le loro modifiche sono già state rese permanenti.

Il recovery manager crea dei checkpoint ad intervalli regolari (ad esempio ogni **m** minuti o ogni **t** transazioni terminate).

Per creare un checkpoint si effettuano le seguenti operazioni:

1. Sospendere temporaneamente l'esecuzione di tutte le transazioni.
2. Scrivere su disco il contenuto di tutti i buffer modificati (scrittura forzata).
3. Scrivere una entry di checkpoint nel log.
4. Riprendere l'esecuzione delle transazioni sospese.

Se una transazione fallisce per una qualsiasi ragione, può essere necessario effettuarne il **rollback**. Il rollback di una transazione **T** richiede il rollback di tutte le transazioni che hanno letto il valore di qualche dato scritto da **T**, e così via (**rollback in cascata**). È complesso da gestire e richiede molto tempo, infatti i meccanismi di recovery sono progettati per evitare di usarlo. Di seguito un esempio.



Operazioni effettuate prima del crash.

	A	B	C	D
	30	15	40	20
[start-transaction, T ₃]				
[read_item, T ₃ , C]				
* [write_item, T ₃ , B, 15, 12]		12		
[start-transaction, T ₂]				
[read_item, T ₂ , B]				
** [write_item, T ₂ , B, 12, 18]		18		
[start-transaction, T ₁]				
[read_item, T ₁ , A]				
[read_item, T ₁ , D]				
[write_item, T ₁ , D, 20, 25]			25	
[read_item, T ₂ , D]				
** [write_item, T ₂ , D, 25, 26]			26	
[read_item, T ₃ , A]				

← system crash

Il System log al momento del crash.

Tecniche di recovery basate su Deferred Update

Protocollo di Deferred Update

È noto che con questa tecnica i dati non sono aggiornati fisicamente fino all'esecuzione della commit di una transazione. Si definisce un **protocollo di deferred update**:

1. Una transazione non può cambiare il database finché non raggiunge il punto di commit.
2. Una transazione non raggiunge il punto di commit finché tutte le sue operazioni di aggiornamento non sono registrate nel log e tale log è scritto su disco.

L'algoritmo è NO-UNDO/REDO, infatti non è mai necessario l'UNDO, ed il REDO è richiesto solo se il crash si ha dopo il commit, ma prima dell'aggiornamento del database.

Recovery con Deferred Update in ambiente monoutente (Single-user)

In questo caso l'algoritmo di recovery è abbastanza semplice: l'algoritmo RDU_S (Recovery usando la Deferred Update in ambiente Single-user) chiama una procedura REDO per rieseguire delle operazioni di write_item. La procedura RDU_S mantiene due liste di transazioni:

1. Transazioni committed a partire dall'ultimo checkpoint,
2. Transazioni attive (contiene al più una transazione, perché il sistema è single-user).

Algoritmo RDU_S

Applicare l'operazione REDO a tutte le operazioni write_item delle transazioni committed nel log, nell'ordine in cui sono scritte nel log. Rilanciare la transazione attiva.

REDO (WRITE-OP): per effettuare il REDO dell'operazione WRITE-OP bisogna esaminare la sua entry nel log [write_item, T, X, new_value] e porre il valore dell'elemento X a new-value (l'AFIM).

T_1	T_2	
read_item(A)		[start-transaction, T_1]
read_item(D)	read_item(B)	[write_item, $T_1, D, 20$]
write_item(D)	write_item(B)	[commit, T_1]
	read_item(D)	[start-transaction, T_2]
	write_item(D)	[write_item, $T_2, B, 10$]
		[write_item, $T_2, D, 25$] ← system crash

Le operazioni di read/write per due transazioni.

Il System log.

L'algoritmo rieffettuerà l'operazione [write_item, $T_1, D, 20$], poiché T_1 era committed.

NOTA: La REDO è **idempotente**, perché se il sistema fallisce durante il recovery, deve essere possibile rifare il recovery, ed il risultato, con o senza crash, deve essere lo stesso.

Recovery con Deferred Update in ambiente Multiutente

In ambienti multiutente, i processi di recovery e di controllo della concorrenza sono interrelati. Maggiore è il grado di concorrenza, più tempo viene impiegato per effettuare il recovery.

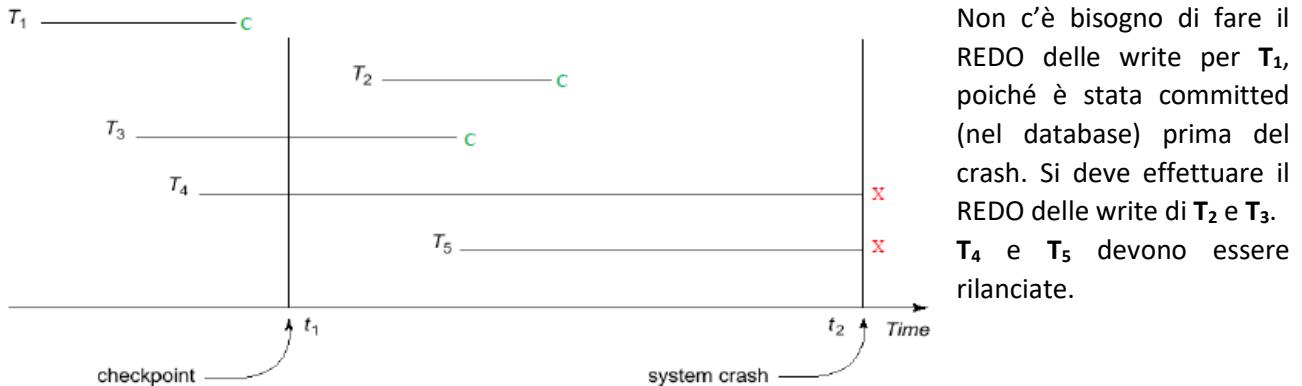
Si consideri un sistema multiutente così gestito: Controllo della concorrenza 2PL stretto (two-phase locking), ovvero il lock è mantenuto fino al punto di commit.

Algoritmo RDU_M

Esso utilizza due liste di transazioni:

1. Transazioni committed **T** a partire dall'ultimo checkpoint.
2. Transazioni attive **T'**.

Fare il REDO di tutte le operazioni di write_item delle transazioni **T** committed nel log, nell'ordine in cui sono state scritte nel log. Le transazioni **T'** attive e non ancora committed devono essere rilanciate.



L'algoritmo può essere migliorato. Se un elemento X è stato aggiornato più volte da transazioni committed a partire dall'ultimo checkpoint, è sufficiente effettuare il REDO solo dell'ultimo aggiornamento.

Svantaggi del Deferred Update

- Limitazione dell'esecuzione concorrente delle transazioni dovuto al 2PL stretto.
- Eccessivo spazio richiesto per memorizzare gli elementi aggiornati prima del commit di una transazione.

Vantaggi del Deferred Update

- Se una transazione è abortita, viene risottomessa senza che sia stato alterato il database su disco. Non è necessario il rollback.
- Una transazione non leggerà mai un dato che sia stato modificato da una transazione non committed (2PL stretto). È escluso il **cascading rollback**.

Tecniche di Recovery basate su Immediate Update

Quando una transazione effettua un comando di aggiornamento:

1. L'operazione viene registrata nel log (write-ahead logging protocol).
2. L'operazione viene applicata nel database.

In questo caso si **necessita di rollback**.

I tipi di Immediate Update si dividono in due categorie:

1. Se tutti gli aggiornamenti di una transazione sono riportati nel database prima del commit, non è mai richiesto il REDO (algoritmo di recovery di tipo UNDO/NO-REDO).
2. Se la transazione raggiunge il commit prima che tutti gli aggiornamenti siano riportati nel database può essere necessario il REDO (algoritmo di recovery di tipo UNDO/REDO).

Algoritmo RIU_S

L'algoritmo RIU_S (Recovery usando l'Immediate Update in ambiente Single-User) utilizza due liste di transazioni:

1. Transazioni committed **T** a partire dall'ultimo checkpoint.
2. Transazione attiva **T'**.

L'algoritmo RIU_S permette di:

- Eseguire l'UNDO delle operazioni write_item della transazione attiva **T'** contenuta nel log.
- Eseguire il REDO delle operazioni write_item delle transazioni committed **T** nell'ordine scritto nel log.

Algoritmo RIU_M

L'algoritmo RIU_M utilizza due liste di transazioni:

1. Transazioni committed **T** a partire dall'ultimo checkpoint.
2. Transazioni attive **T'**.

L'algoritmo RIU_M permette di:

- Eseguire l'UNDO delle operazioni write_item delle transazioni attive **T'** contenute nel log nell'ordine inverso rispetto a quello del log.
- Eseguire il REDO delle operazioni write_item delle transazioni committed **T** nell'ordine scritto nel log.

Recovery nei sistemi Multidatabase

Una Transazione Multidatabase è una *transazione che richiede l'accesso a database multipli*. Questi database possono essere gestiti da DBMS differenti (relazionali, O.O., gerarchici, ecc...). Vi è un meccanismo di recovery a due livelli:

1. Local recovery manager.
2. Global recovery manager (*coordinatore*).

Protocollo di commit a due fasi: fase 1

1. Tutti i database coinvolti dalla transazione segnalano al coordinatore di aver completato la loro parte.
2. Il coordinatore manda ad ogni partecipante il messaggio "*prepare for commit*".
3. Ogni partecipante forza su disco tutte le informazioni per il recovery locale e risponde "OK" al coordinatore. Se per qualche ragione non può esser fatto il commit, vi sarà una risposta "NOT OK".

Protocollo di commit a due fasi: fase 2

Se il coordinatore riceve "OK" da tutti i partecipanti, invia un comando di commit ai database coinvolti. Ogni partecipante segnala il [commit] nel system log, ove necessario, ed aggiorna il database.

Se qualche partecipante ha fornito un "NOT OK", la transazione fallisce e il coordinatore invia un messaggio UNDO ai partecipanti.

Backup e Recovery di Database da failure catastrofici

La principale tecnica è quella del back-up di database. Periodicamente il database e il system log sono ricoppiati su un device di memoria economico. Il system log è back-upped più di frequente rispetto al database intero. In caso di failure catastrofico, tutto il database viene ricaricato su dischi e, seguendo il system log, gli effetti delle transazioni committed vengono ripristinati.

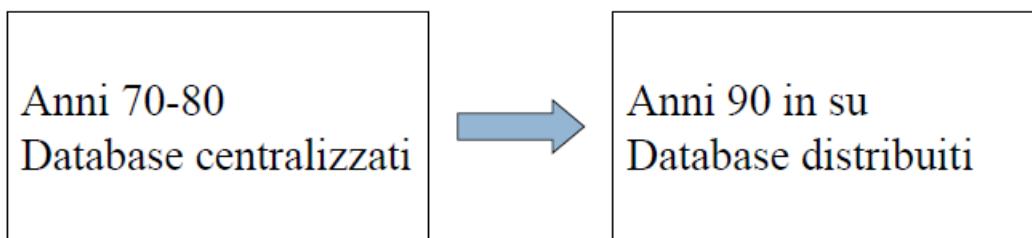
Per non perdere tutte le transazioni effettuate dall'ultimo back-up, i file di log sono ricoppiati molto frequentemente. È possibile fare ciò grazie alle ridotte dimensioni di tali file rispetto alla taglia dell'intero database.



Basi di Dati Distribuite ed Architetture Client-Server

La tecnologia per i DDB (Database Distribuiti) è il risultato dell'unione di due tecnologie:

- La tecnologia delle **basi di dati**.
- La tecnologia delle **reti** e della **comunicazione** che ha avuto un'evoluzione enorme (cellulari, internet, ecc....)



Utilizzare un DDB consente alle organizzazioni di effettuare un processo di *decentralizzazione* che viene raggiunta a livello di sistema, e di *integrare* le informazioni a livello logico. Un **database distribuiti** è un singolo database logico che è sparso fisicamente attraverso computer in località differenti e connessi attraverso una rete di comunicazione dati.

Invece, un **sistema per la gestione di basi di dati distribuite** (DDBMS) è un sistema software che gestisce un database distribuito rendendo la distribuzione *trasparente* all'utente. La rete quindi consentire agli utenti di *condividere dati*, ovvero: un utente della località A deve essere in grado di accedere (ed eventualmente aggiornare) i dati della località B.

I computer ovviamente possono variare dai microcomputer ai super-computer.

Tipi di architetture multiprocessore

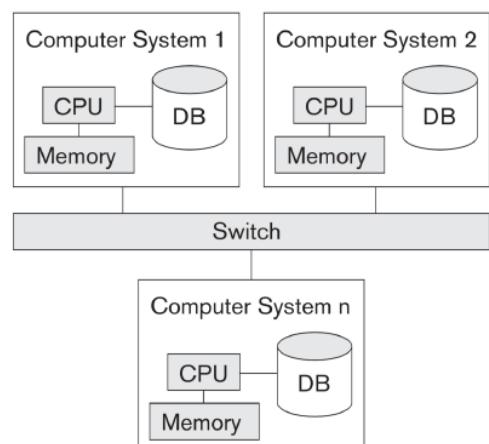
Vi sono due tipologie di architetture multiprocessore:

- Architettura a **memoria condivisa (shared memory)**, dove diversi processori condividono sia la memoria primaria, sia la secondaria.
- Architettura a **disco condiviso (shared disk)**, dove i processori condividono la memoria secondaria, ma ognuno possiede la propria memoria primaria.

Usando queste architetture i processori possono comunicare senza utilizzare la rete (vi è meno overhead). I DBMS che si basano su queste architetture si chiamano **Parallel DBMS**.

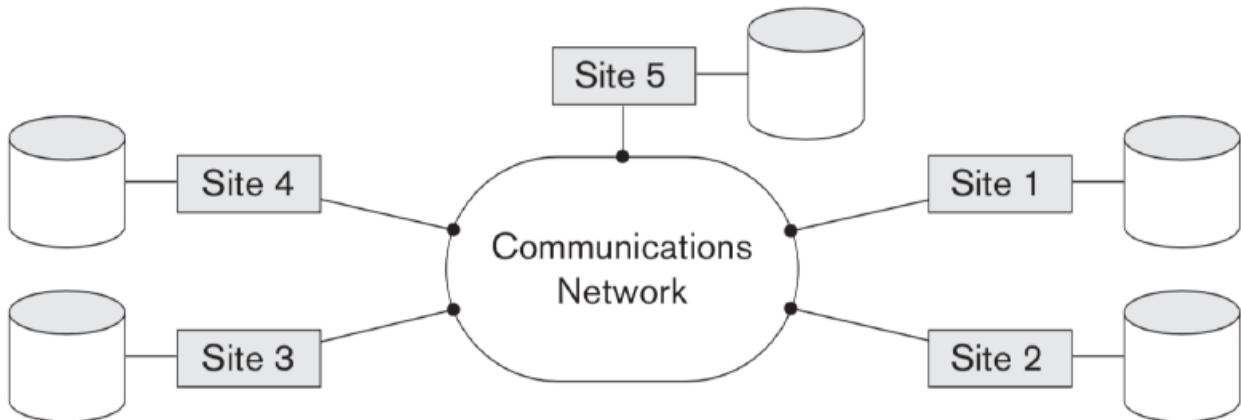
Architettura Shared Nothing (a memorie separate)

Ogni processore possiede la sua memoria primaria e secondaria e comunicano attraverso la rete. Non esiste una memoria comune e i nodi sono simmetrici ed omogenei.



Architettura di un DDB

Un database distribuito è caratterizzato dall'**eterogeneità** dell'hardware e dei sistemi operativi di ogni nodo.



Un database distribuito richiede **DDBMS multipli**, funzionanti ognuno su di un sito remoto (nodo). Gli ambienti di database distribuiti si differenziano in base a:

- Il grado di cooperazione dei DDBMS.
- La presenza di un **sito master** che coordina le richieste che coinvolgono dati memorizzati in siti multipli.

I **vantaggi** dei Database Distribuiti, innanzitutto partono dalla gestione dei dati distribuiti a diversi livelli di **trasparenza**, ovvero sono nascosti i dettagli riguardo l'ubicazione di ogni data item presente nel sistema.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	-----	-------	---------	-----	--------	-----------	-----

WORKS_ON

Essn	Pno	Hours
------	-----	-------

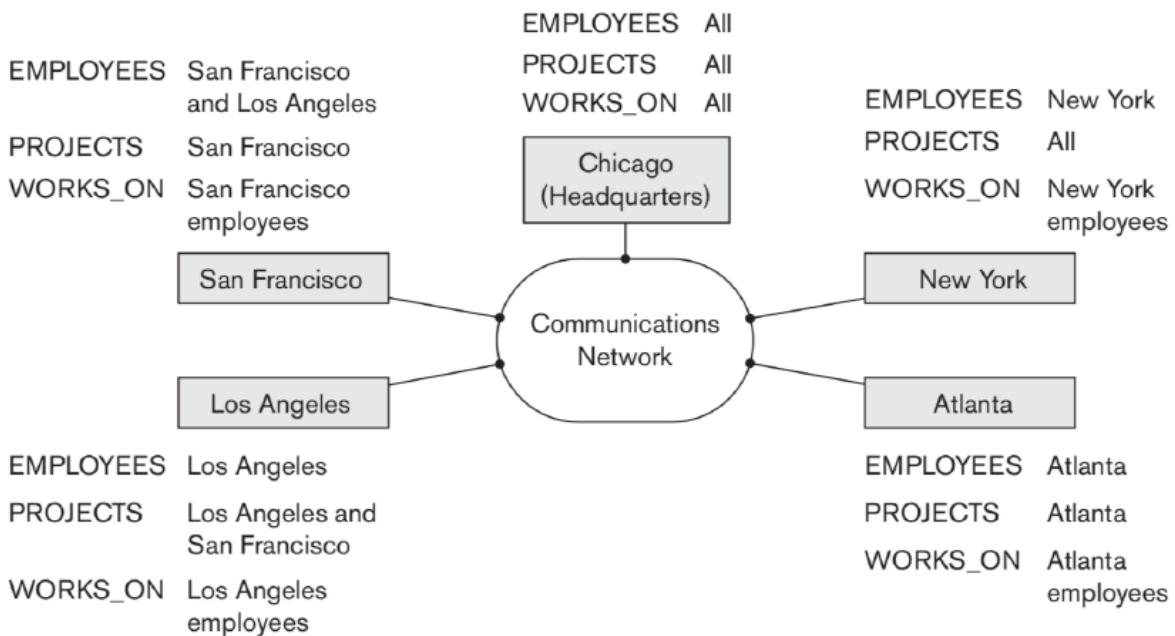
PROJECT

Pname	Pnumber	Plocation	Dnum
-------	---------	-----------	------

Le tabelle potrebbero essere frammentate orizzontalmente o memorizzate con replicazione.

Livelli di trasparenza

Vengono elencati di seguito i vari livelli di trasparenza:



Trasparenza di distribuzione o di rete: dove vi è libertà dell'utente dai dettagli della rete. Essa si suddivide in:

- **Trasparenza di locazione:** un comando utilizzato è indipendente dalla località dei dati e dalla località di emissione del comando.
- **Trasparenza di naming:** la specifica di un nome di un oggetto implica che una volta che un nome è stato fornito, gli oggetti possono essere referenziati usando quel nome, senza dover fornire dettagli addizionali.

Trasparenza di replicazione: dove delle copie dei dati possono essere mantenute in siti multipli per una migliore disponibilità, prestazioni ed affidabilità. L'utente non percepisce l'esistenza di tali copie.

Trasparenza di frammentazione: sono possibili due tipi di frammentazione:

- La frammentazione **orizzontale** che distribuisce una relazione attraverso insieme di tuple.
- La frammentazione **verticale** che distribuisce una relazione in sotto relazioni formate da un sottoinsieme delle colonne della relazione originale.

Una **query globale** dell'utente deve essere trasformata in una serie di **frammenti di query**. La trasparenza di frammentazione consente all'utente di non percepire l'esistenza di tali frammenti.

Vantaggi dei DDB

Vi sono vari vantaggi alla base dell'utilizzo dei DDB. Infatti, essi presentano:

- **Affidabilità:** la probabilità che il sistema sia funzionante ad un certo istante.
- **Disponibilità:** la probabilità che il sistema è continuamente disponibile durante un intervallo di tempo.

Quando un sito fallisce, gli altri possono continuare ad operare, solo i dati del sito fallito sono inaccessibili. La replicazione dei dati aumenta ancora di più l'affidabilità e la disponibilità.

In un sistema distribuito è più facile espandere il sistema in termini di aggiungere nuovi dati, accrescere il numero di siti o aggiungere nuovi processori (**miglioramento delle prestazioni**).

La **localizzazione dei dati** dove, praticamente, i dati sono mantenuti nei siti più vicini a dove sono più utilizzati; ciò comporta una riduzione di accesso a reti geografiche. Ogni sito ha un database di taglia più piccola e le query e le transazioni sono processate più rapidamente. Le transazioni su ogni sito sono in numero minore rispetto ad un database centralizzato.

Funzionalità addizionali caratteristiche dei DDBMS

Rispetto ai tradizionali DBMS, i DDBMS devono fornire le seguenti funzionalità:

- **Mantenere traccia dei dati**, della loro distribuzione, frammentazione e replicazione nel catalogo.
- **Processare query distribuite**: l'abilità di accedere a siti remoti e trasmettere query e dati tra i vari siti attraverso la rete di comunicazione.
- **Gestire le transazioni distribuite**, query e transazioni devono accedere ai dati da più di un sito mantenendo l'integrità del DB.
- Gestire la replicazione dei dati, quindi decidere quando replicare un dato e **mantenere la consistenza fra le copie**.
- Gestire il recovery di un DDB, ovvero il **fallimento di uno dei siti e dei link di comunicazione**.
- Sicurezza: le transazioni distribuite devono essere gestite garantendo la **sicurezza dei dati e l'accesso degli utenti**.
- Gestire il catalogo distribuito: il catalogo deve contenere i dati dell'intero DB ed essere globale. Deve essere deciso **come e dove distribuire il catalogo**.

A *livello hardware*, si devono considerare le seguenti differenze rispetto ai DBMS:

- Esistono diversi computer, detti siti o nodi.
- I siti devono essere connessi attraverso una rete di comunicazione per trasmettere dati e comandi ad altri siti.

I siti possono essere connessi attraverso una rete locale e/o distribuiti geograficamente e connessi ad una rete geografica.

Il Design dei DDB

Il design dei DDB prevede l'applicazione di tecniche relative a:

- **Frammentazione** dei dati.
- **Replicazione** dei dati.
- **Allocazione** dei frammenti.

Le informazioni relative saranno memorizzate nel catalogo.

Frammentazione dei dati

Deve essere deciso quale sito deve memorizzare quale porzione del database. Si assume che non ci siano replicazione dei dati. Una relazione può:

1. Essere memorizzata per intero in un sito.
2. Essere divisa in unità più piccole distribuite.

EMPLOYEE									
FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

Lo schema del database "Company".

DEPARTMENT			
DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATION	
DNUMBER	DLOCATION

PROJECT			
PNAME	PNUMBER	PLOCATION	DNUM

WORKS_ON		
ESSN	PNO	HOURS

DEPENDENT				
ESSN	DEPARTMENT_NAME	SEX	BDATE	RELATIONSHIP

Frammentazione Orizzontale

Un frammento orizzontale di una relazione è un sottoinsieme delle tuple della relazione. le tuple vengono assegnate ad un determinato frammento orizzontale in base al valore di uno o più attributi. Ad esempio, considerato il database "Company", esso viene distribuito su 3 siti e si definiscono 3 frammenti orizzontali in base al valore del numero del dipartimento (DNO=5) (DNO=4) (DNO=1). Ogni frammento contiene le tuple degli impiegati che lavorano per un particolare dipartimento.



La segmentazione orizzontale

La segmentazione orizzontale divide una relazione raggruppando righe per creare sottoinsiemi di tuple, ciascuno con un significato logico. La segmentazione orizzontale **derivata** applica la partizione di una relazione primaria anche a relazioni secondarie, collegate alla prima con una chiave esterna. Ad esempio, dal partizionamento di DIPARTIMENTO deriva il partizionamento di IMPIEGATO e PROGETTO.

Frammentazione Verticale

La frammentazione verticale divide una relazione "verticalmente" per colonne. Un frammento verticale di una relazione mantiene solo determinati attributi di una relazione. Ad esempio, IMPIEGATO può essere suddiviso in due frammenti:

- Uno contenente le informazioni personali {NOME, DATA_DI_NASCITA, INDIRIZZO E SESSO}.
- L'altro contenente {SSN, SALARIO, SUPERSSN, NUMERO_DIPARTIMENTO}.



La frammentazione dell'esempio precedente non consente di ricostruire la tupla IMPIEGATO originale, infatti non ci sono attributi in comune fra i due frammenti. La **soluzione** è aggiungere

SSN agli attributi personali. È necessario includere la chiave primaria o una chiave candidata in ogni frammento verticale.

La frammentazione orizzontale e l'algebra relazionale

Ogni frammento orizzontale su di una relazione R può essere specificato attraverso un'operazione $\sigma_{C_i}(R)$. Un insieme di frammenti orizzontali tale che le condizioni C_1, \dots, C_n includono tutte le tuple della relazione, è detto *frammentazione orizzontale completa*. In molti casi i frammenti sono disgiunti: nessuna tupla in R soddisfa $C_i \text{ AND } C_j$ per $i \neq j$.

La frammentazione verticale e l'algebra relazionale

Un frammento verticale può essere specificato da una operazione dell'algebra relazionale $\pi_{L_i}(R)$. Un insieme di frammenti verticali la cui lista di proiezioni L_1, \dots, L_n include tutti gli attributi di R e condivide solo la chiave primaria è detta frammentazione verticale completa. Vengono presentate ora le caratteristiche della frammentazione verticale completa. Devono essere soddisfatte le seguenti condizioni:

- $L_1 \cup L_2 \cup \dots \cup L_n = \text{ATTRIBUTI } (R)$
- $L_i \cap L_j = \text{PK}(R)$ per ogni $i \neq j$.

Per ricostruire la relazione R dai frammenti verticali è necessario fare l'OUTER JOIN dei frammenti. Ad esempio, i due frammenti $L_1 = \{\text{NOME, INDIRIZZO}\}$ e $L_2 = \{\text{SSN, NOME, SALARIO}\}$, non costituiscono un frammento verticale completo.

Frammentazione Mista

In questa modalità è possibile combinare la frammentazione orizzontale e verticale. Ad esempio, si combinano le due frammentazioni viste per la relazione IMPIEGATO. La relazione originaria può essere ricostruita applicando OUTER JOIN e UNION nell'ordine appropriato.

Schema di frammentazione

Uno **schema di frammentazione** di un database è la definizione di un insieme di frammenti che include tutte le tuple e gli attributi del database. Esso, inoltre, consente la ricostruzione dell'intero database applicando una sequenza di operazioni di OUTER JOIN e UNION.

Schema di allocazione

Lo **schema di allocazione** descrive l'allocazione dei frammenti ai siti del database. Associa ad ogni frammento il sito in cui deve essere memorizzato. Un frammento memorizzato su più di un sito si dice *replicato*.

Replicazione ed allocazione di dati

La replicazione consente di migliorare la disponibilità dei dati. Le modalità di replicazione sono le seguenti:

- Replicare l'intero database in ogni sito, migliora le prestazioni per le query. L'overhead risulta essere eccessivo per l'aggiornamento dei dati e il controllo della concorrenza e recovery sono complessi.
- Nessuna replicazione, frammenti disgiunti (eccetto per la chiave).
- Replicazione parziale, una soluzione intermedia, dove alcuni frammenti possono essere replicati ed altri no.

Ad esempio, i lavoratori con PC wireless memorizzano sul proprio PC parte del database e la sincronizzano periodicamente con il database server.

Schema di replicazione

Lo **schema di replicazione** è una descrizione della replicazione dei frammenti. Ogni segmento deve essere assegnato ad un sito del database, e questo processo porta il nome di *allocazione dei dati*. La scelta dei siti e del grado di replicazione dipende da:

- Prestazioni.
- Obiettivi di disponibilità del sistema.
- Tipo e frequenza delle transazioni sottomesse ad ogni sito.

Criteri per distribuire i dati

Trovare una soluzione ottima alla distribuzione dei dati è un problema difficile. Se è richiesta un'alta disponibilità e le transazioni possono essere sottomesse ad ogni sito e molte transazioni sono solo di *retrieval*, si può adottare una soluzione di replicazione totale.

Se alcune transazioni che accedono a porzioni del database sono sottomesse solo da particolari siti, su quei siti possono essere allocati i frammenti corrispondenti.

Se sono richiesti molti *aggiornamenti* è conveniente ridurre la replicazione dei dati.

Esempi di frammentazione, allocazione e replicazione

Si supponga che un'organizzazione "Company" abbia tre siti, uno per ogni dipartimento.

Il **sito 2** e il **sito 3** sono relativi ai dipartimenti 5 e 4 rispettivamente. Da ognuno di questi siti si accede di frequente ai dati associati agli *impiegati* e ai *progetti* dei dipartimenti corrispondenti. Si assume che i siti 2 e 3 accedano soprattutto alle informazioni NOME, SSN, SALARIO, SUPERSSN di IMPIEGATO.

Il **sito 1** è usato dagli amministratori per accedere ai dati dell'intera compagnia regolarmente.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

La base di dati relazionale "COMPANY"

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

L'intero database può essere memorizzato nel **sito 1**. I frammenti del **sito 2** e **sito 3** sono determinati frammentando orizzontalmente DIPARTIMENTO in base alla chiave NUMERO_DIPARTIMENTO.

Applicando la frammentazione *derivata* alle relazioni IMPIEGATO, PROGETTO e LOCAZIONE_DIPARTIMENTO basate sulla chiave esterna che riferisce il dipartimento. Frammentando verticalmente la relazione IMPIEGATO per includere solo gli attributi {NOME, SSN, SALARIO, SUPERSSN, NUMERO_DIPARTIMENTO}.

Sorgono problemi con la relazione WORKS_ON (M:N), infatti WORKS_ON (ESSN, PNO, Hours) non ha per attributo il numero di dipartimento. Ogni tupla collega un impiegato ed un progetto. È possibile frammentarla in base a:

- Al dipartimento per cui l'impiegato lavora.
- Al dipartimento che controlla il progetto.

In questo caso viene scelto l'OR delle ipotesi.

EMPD_5

Fname	Minit	Lname	Ssn	Salary	Super_ssn	Dno
John	B	Smith	123456789	30000	333445555	5
Franklin	T	Wong	333445555	40000	888665555	5
Ramesh	K	Narayan	666884444	38000	333445555	5
Joyce	A	English	453453453	25000	333445555	5

Dati al sito 2

DEP_5

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22

DEP_5_LOCS

Dnumber	Location
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON_5

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0

PROJS_5

Pname	Pnumber	Plocation	Dnum
Product X	1	Bellaire	5
Product Y	2	Sugarland	5
Product Z	3	Houston	5

EMPD_4

Fname	Minit	Lname	Ssn	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	25000	987654321	4
Jennifer	S	Wallace	987654321	43000	888665555	4
Ahmad	V	Jabbar	987987987	25000	987654321	4

Dati al sito 3

DEP_4

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Administration	4	987654321	1995-01-01

DEP_4_LOCS

Dnumber	Location
4	Stafford

WORKS_ON_4

Essn	Pno	Hours
333445555	10	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0

PROJS_4

Pname	Pnumber	Plocation	Dnum
Computerization	10	Stafford	4
New_benefits	30	Stafford	4

I frammenti di WORKS_ON per gli impiegati che lavorano nel dipartimento 5:

G1

<u>Essn</u>	<u>Pno</u>	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0

C1 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

C = [ESSN IN (SELECT SSN FROM EMPLOYEE WHERE DNO=5)]

G2

<u>Essn</u>	<u>Pno</u>	Hours
333445555	10	10.0

C2 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G3

<u>Essn</u>	<u>Pno</u>	Hours
333445555	20	10.0

C3 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

I frammenti di WORKS_ON per gli impiegati che lavorano nel dipartimento 4:

G4

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

C4 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G5

<u>Essn</u>	<u>Pno</u>	Hours
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0

C5 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G6

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

C6 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

C = [ESSN IN (SELECT SSN FROM EMPLOYEE WHERE DNO=4)]

I frammenti di WORKS_ON per gli impiegati che lavorano nel dipartimento 1:

G7

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

C7 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 5))

G8

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

C8 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 4))

G9

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

C9 = C and (Pno in (SELECT Pnumber FROM PROJECT WHERE Dnum = 1))

C = [ESSN IN (SELECT SSN FROM EMPLOYEE WHERE DNO=1)]

Allocazione dei segmenti

L'unione dei frammenti G1, G2, G3, G4, e G7 sono allocati nel **sito 2**. L'unione dei frammenti G2, G4, G5, G6, e G8 sono allocati nel **sito 3**. I frammenti G2 e G4 sono replicati in entrambi i siti.

Questa strategia di allocazione permette di eseguire il JOIN tra i frammenti locali IMPIEGATO o PROGETTO dei siti 2 e 3 e il frammento locale WORKS_ON completamente in locale.

Ad esempio, per il sito 2, l'unione dei frammenti G1, G4 e G7 restituisce tutte le tuple di WORKS_ON relativi ai progetti controllati dal dipartimento 5. Per il sito 3 vale lo stesso per i segmenti G2, G5 e G8.

Frammenti del sito 2

Segmento	Tabella	Condizioni di Selezione
(a)	G1	C1=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=5))
	G2	C2=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=4))
	G3	C3=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=1))
(b)	G4	C4=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=5))
	G5	Employees in Department 5
	G6	C6=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=1))
(c)	G7	C7=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=5))
	G8	C8=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=4))
	G9	C9=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=1))

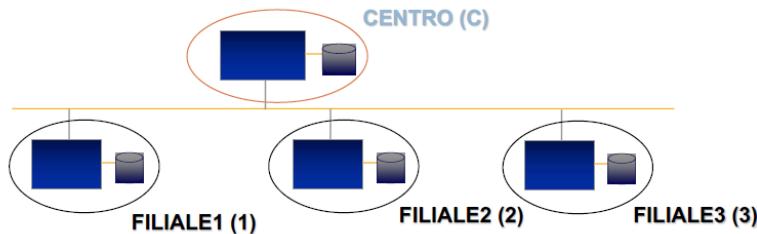
Frammenti del sito 3

Segmento	Tabella	Condizioni di Selezione
(a)	G1	C1=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=5))
	G2	C2=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=4))
	G3	C3=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=1))
(b)	G4	C4=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=5))
	G5	Employees in Department 5
	G6	C6=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=1))
(c)	G7	C7=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=5))
	G8	C8=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=4))
	G9	C9=C AND (PNO IN (SELECT PNUMBER FROM PROJECT WHERE DNUM=1))

Esempio: Conti correnti bancari

CONTO-CORRENTE (NUM-CC, NOME, FILIALE, SALDO)

TRANSAZIONE (NUM-CC, DATA, PROGR, AMMONTARE, CASUALE)



Frammentazione orizzontale principale

$$R_i = \sigma_{P_i}(R)$$

- CONTO1 = $\sigma_{Filiale=1}$ (CONTO-CORRENTE)
- CONTO2 = $\sigma_{Filiale=2}$ (CONTO-CORRENTE)
- CONTO3 = $\sigma_{Filiale=3}$ (CONTO-CORRENTE)

Frammentazione orizzontale derivata

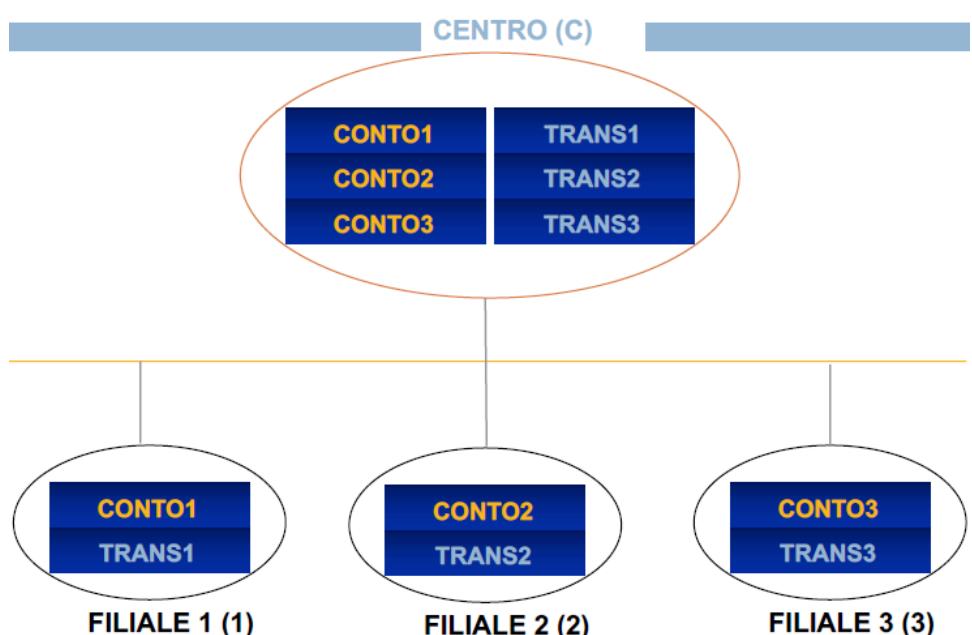
$$S_i = S \triangleright \triangleleft R_i$$

- TRANS1 = TRANSAZIONE $\triangleright \triangleleft$ CONTO1
- TRANS2 = TRANSAZIONE $\triangleright \triangleleft$ CONTO2
- TRANS3 = TRANSAZIONE $\triangleright \triangleleft$ CONTO3

Allocazione dei frammenti

Rete: 3 siti periferici e 1 sito centrale.

Allocazione: locale e centrale.



Livelli di trasparenza

Modalità per esprimere interrogazioni offerte dai DBMS commerciali (LIVELLI):

- **FRAMMENTAZIONE.** Il programmatore non si deve preoccupare se o no il database è distribuito o frammentato.
- **ALLOCAZIONE.** Il programmatore dovrebbe conoscere la struttura dei frammenti, ma non deve indicare la loro allocazione.
- **LINGUAGGIO.** Il programmatore deve indicare nella query sia la struttura dei frammenti che la loro allocazione.

Trasparenza di frammentazione

QUERY: Estrarre il saldo del conto corrente 45.

```
SELECT SALDO  
      FROM CONTO-CORRENTE  
      WHERE NUM-CC = 45
```

Trasparenza di allocazione

IPOTESI: Conto corrente 45 presso la Filiale 1 (locale).

```
SELECT SALDO  
      FROM CONTO1  
      WHERE NUM-CC=45
```

IPOTESI: Allocazione incerta: probabilmente allocato presso la Filiale 1.

```
SELECT SALDO FROM CONTO1  
      WHERE NUM-CC = 45  
      IF (NOT FOUND) THEN  
          (SELECT SALDO FROM CONTO2  
              WHERE NUM-CC = 45  
              UNION  
          SELECT SALDO FROM CONTO3  
              WHERE NUM-CC = 45)
```

Trasparenza di linguaggio

```
SELECT SALDO FROM CONTO1@1  
      WHERE NUM-CC = 45  
IF (NOT FOUND) THEN  
  (SELECT SALDO FROM CONTO2@c  
    WHERE NUM-CC = 45  
UNION  
  SELECT SALDO FROM CONTO3@c  
    WHERE NUM-CC = 45
```

Trasparenza di frammentazione

QUERY: Estrarre i movimenti dei conti con saldo negativo.

```
SELECT NUM-CC, PROGR, AMMONTARE  
FROM CONTO-CORRENTE AS C  
JOIN TRANSAZIONE AS T  
ON C.NUM-CC = T.NUM-CC  
WHERE SALDO < 0
```

Trasparenza di allocazione (join distribuito)

```
SELECT NUM-CC, PROGR, AMMONTARE  
FROM CONTO1 JOIN TRANS1 ON....  
WHERE SALDO < 0  
UNION  
SELECT NUM-CC, PROGR, AMMONTARE  
FROM CONTO2 JOIN TRANS2 ON....  
WHERE SALDO < 0  
UNION  
SELECT NUM-CC, PROGR, AMMONTARE  
FROM CONTO3 JOIN TRANS3 ON....  
WHERE SALDO < 0
```

Trasparenza di linguaggio

```
SELECT NUM-CC, PROGR, AMMONTARE  
      FROM CONTO1@1 JOIN TRANS1@1 ON....  
      WHERE SALDO < 0  
UNION  
SELECT NUM-CC, PROGR, AMMONTARE  
      FROM CONTO2@c JOIN TRANS2@c ON....  
      WHERE SALDO < 0  
UNION  
SELECT NUM-CC, PROGR, AMMONTARE  
      FROM CONTO3@c JOIN TRANS3@c ON....  
      WHERE SALDO < 0
```

Trasparenza di frammentazione

UPDATE: Sposta il cliente 45 dalla Filiale 1 alla Filiale 2.

```
UPDATE CONTO-CORRENTE  
      SET FILIALE = 2  
      WHERE NUM-CC = 45 AND FILIALE = 1
```

Trasparenza di allocazione (e replicazione)

```
INSERT INTO CONTO2  
      SELECT * FROM CONTO1 WHERE NUM-CC = 45  
INSERT INTO TRANS2  
      SELECT * FROM TRANS1 WHERE NUM-CC = 45  
DELETE FROM CONTO1 WHERE NUM-CC = 45  
DELETE FROM TRANS1 WHERE NUM-CC = 45
```

(Bisogna settare anche la FILIALE a 2)

Trasparenza di linguaggio

```
INSERT INTO CONTO2@2
```

```
SELECT * FROM CONTO1@1 WHERE NUM-CC = 45
```

```
INSERT INTO CONTO2@C
```

```
SELECT * FROM CONTO1@C WHERE NUM-CC = 45
```

```
INSERT INTO TRANS2@2
```

```
SELECT * FROM TRANS1@1 WHERE NUM-CC = 45
```

```
INSERT INTO TRANS2@C
```

```
SELECT * FROM TRANS1@C WHERE NUM-CC = 45
```

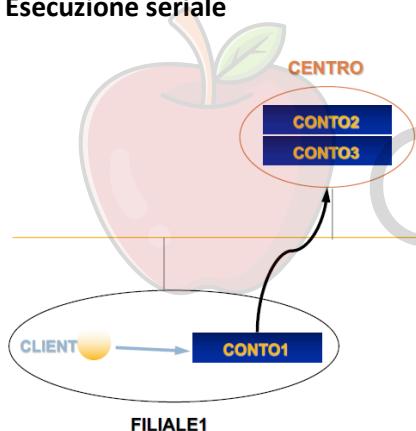
(In modo analogo: 2 coppie di DELETE)

Bisogna settare anche la FILIALE a 2

Efficienza

Ottimizzazione delle query. La modalità di esecuzione è **seriale** e **parallela**.

Esecuzione seriale



Esecuzione parallela



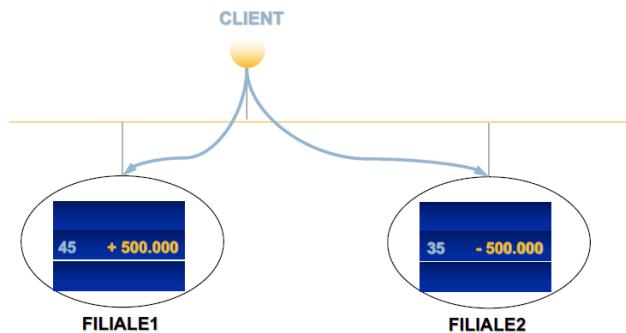
Transazioni distribuite

BEGIN TRANSACTION

```
UPDATE CONTO1@1
SET SALDO = SALDO + 500.000
WHERE NUM-CC = 45;
UPDATE CONTO2@2
SET SALDO = SALDO - 500.000
WHERE NUM-CC = 35;
```

COMMIT-WORK

END TRANSACTION

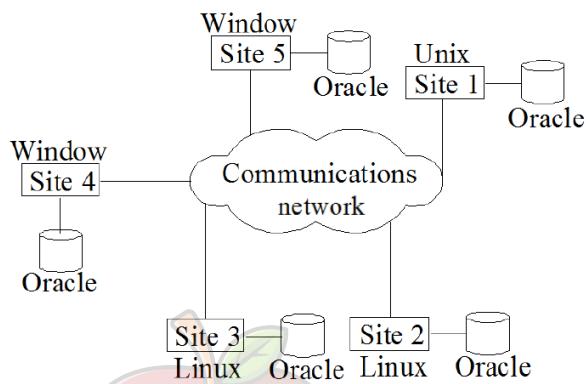


Tipi di DDB

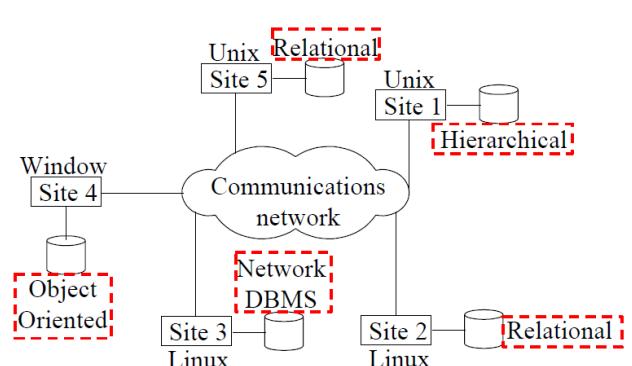
Si differenziano per diversi fattori:

- *Grado di omogeneità* del software dei DDBMS, dove:
 - **DDB omogenei**: tutti i server e tutti gli utenti usano lo stesso software.
 - **DDB eterogenei**, altrimenti.
- *Grado di autonomia locale*, dove:
 - Se il sito locale non può funzionare come un DBMS stand-alone non ha nessuna autonomia locale, e all'utente il sistema “appare” come un DBMS centralizzato.
 - Se alle transazioni locali è permesso l'accesso diretto a un server, ha qualche grado di autonomia locale.

DDB omogenei



DDB eterogenei



Sistemi di database **federati**: ogni server è un DBMS centralizzato e autonomo con:

- I suoi propri utenti locali;
- Transazioni locali;
- DBA;
- E, quindi, un alto grado di autonomia locale.

È presente una vista globale o schema della federazione di database.

Un sistema multidatabase (DB federati o FDB) non ha uno schema globale e interattivamente ne viene costruito uno (vista globale) in base alle necessità dell'applicazione.

In un FDB eterogeneo, i DB coinvolti possono essere relazionali, reticolari, gerarchici, ecc... Si rende necessario introdurre dei **traduttori di query**.

Sistemi per la gestione dei DB Federati

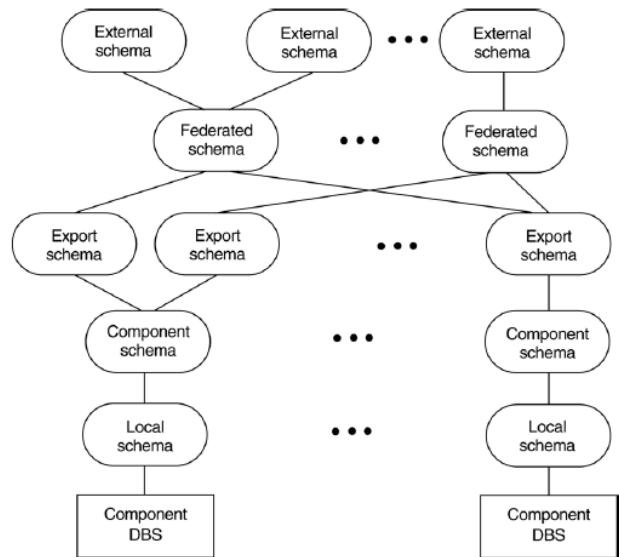
Vi sono diversi tipi differenti di *eterogeneità*:

- Differenze nei **modelli di dati**.
- Differenze nei **vincoli**: le modalità per la specifica dei vincoli varia da sistema a sistema.
- Differenze nei **linguaggi** di query: anche con lo stesso modello di dati i linguaggi e le loro versioni possono variare.
- Eterogeneità **semantica**: differenze di significato, interpretazione, uso degli stessi dati o di dati correlati.

Essa costituisce la principale difficoltà nella progettazione degli schemi globali di basi di dati eterogenee.

Lo schema a cinque livelli (FDBS)

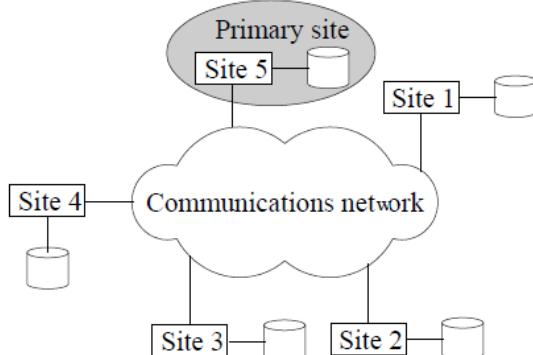
- Lo schema per un gruppo di utenti o un'applicazione.
- Schema globale risultante dall'integrazione di più schemi.
- Sottoinsieme dello schema componenti.
- Modello dati comune.
- Schema concettuale.



Controllo della concorrenza e Recovery

I database distribuiti incontrano un numero di controlli di concorrenza e problemi di recovery che non sono presenti nei database centralizzati. Ne vengono elencati alcuni di seguito:

- Trattamento di **copie multiple** di dati. Il controllo della consistenza deve mantenere una **consistenza globale**. Allo stesso modo il meccanismo di recovery deve recuperare tutte le copie e conservare la consistenza dopo il recovery.
- Fallimenti di **singoli siti**. La disponibilità del database non deve essere influenzata dai guasti di uno o due siti e lo schema di recovery li deve recuperare prima che siano resi disponibili.
- Guasto dei **collegamenti di comunicazione**. Tale guasto può portare ad una partizione della rete che può influenzare la disponibilità del database anche se tutti i siti sono in esecuzione.
- **Commit distribuito**. Una transazione può essere frammentata ed essere eseguita su un numero di siti. Questo richiede un approccio basato sul *commit a due fasi* per il commit della transazione.
- **Deadlock distribuito**. Poiché le transazioni sono processate su siti multipli, due o più siti possono essere coinvolti in un **deadlock**. Di conseguenza devono essere considerate le tecniche per il trattamento dei deadlock.
- Controllo della concorrenza distribuito basato su una **copia designata** dei dati. Si designa una particolare copia di ogni dato (**copia designata**). Tutte le richieste di **lock** ed **unlock** vengono inviate solo al sito che la contiene.
- Tecnica del **sito primario**. Un singolo sito è designato come **sito primario** il quale fa da coordinatore per la gestione delle transazioni.



Tecnica del sito primario

Nell'ottica della gestione delle transazioni il controllo della concorrenza ed il commit sono gestiti da questo sito. La tecnica del lock a due fasi (2PL) è usata per bloccare e rilasciare i data item. Se tutte le transazioni in tutti i siti seguono la politica delle due fasi allora viene garantita la serializzabilità.

Vantaggi: i data item sono bloccati (locked) solamente in un sito ma possono essere usati da qualsiasi altro sito.

Svantaggi: tutta la gestione delle transazioni passa per il sito primario che potrebbe essere **sovaccaricato**. Nel caso in cui il sito primario fallisce, l'intero sistema è inaccessibile.

Per assistere il recovery, un sito di backup viene designato come copia di backup del sito primario. Nel caso di fallimento del sito primario, il sito di backup funziona da sito principale.

Tecnica della copia primaria

In questo approccio, invece di un sito, una partizione dei data item è designata come copia primaria. Per bloccare un data item soltanto sulla copia primaria di quel data item viene eseguito il lock.

Vantaggi: le copie primarie sono distribuite su vari siti, e un singolo sito **non viene sovraccaricato** da un numero elevato di richieste di lock ed unlock.

Svantaggi: l'identificazione della copia primaria è complesso. Una directory distribuita deve essere gestita possibilmente in tutti i siti.

Recovery dal fallimento di un coordinatore

In entrambi gli approcci un **sito coordinatore** o un **sito copia** possono essere indisponibili. Questo richiede la selezione di un *nuovo coordinatore*.

- **Approccio del sito primario senza sito di backup.** Abortire e far ripartire tutte le transazioni attive in tutti i siti. Si elegge un nuovo coordinatore che inizia il processing delle transazioni.
- **Sito primario con copia di backup.** Sospende tutte le transazioni attive, designa il sito di backup come sito primario e identifica un nuovo sito di backup. Il nuovo sito primario riceve il compito di gestire tutte le transazioni per riprendere il processo.

Se il sito primario e quello di backup falliscono si utilizza un **processo di elezione** per selezionare un nuovo sito coordinatore.

Controllo della concorrenza basata sul Voting

In questo caso **non esiste la copia primaria del coordinatore**. Quindi:

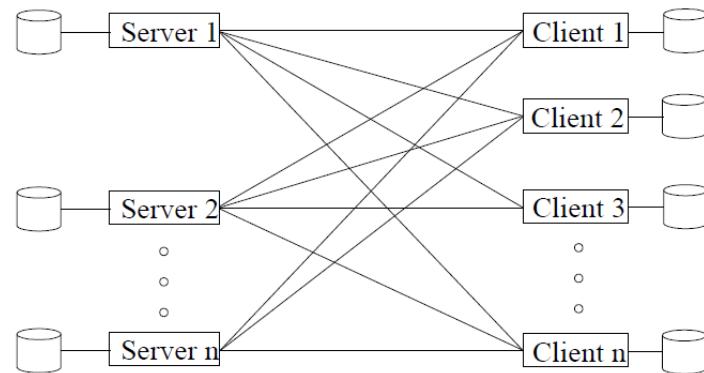
1. Spedire una richiesta di lock ai siti che hanno i data item.
2. Se la maggioranza dei siti concedono il lock allora la transazione richiedente ottiene il data item.
3. Le informazioni di lock (concesse o negate) sono spedite a tutti questi siti.
4. Per evitare un tempo di attesa inaccettabile, viene definito un periodo di **time-out**. Se la transazione richiedente non riceve alcuna informazione di voto allora la transazione viene abortita.

Architettura Client-Server

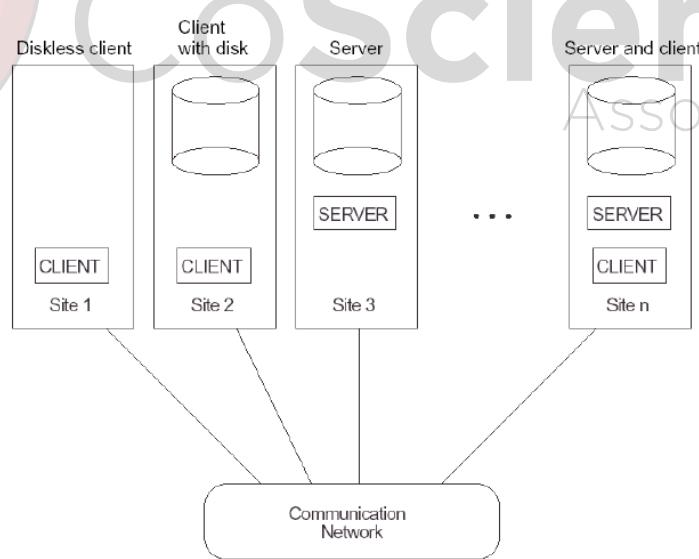
Viene discussa l'architettura Client-Server in generale e poi verrà applicata ai DBMS.

In un ambiente con un grande numero di PC, stampanti, ecc.... Si definiscono dei **server specializzati** con funzionalità specifiche. Ad esempio:

- Il file server mantiene i file delle macchine client.
- Il printer server gestisce la stampa su diverse stampanti.
- L'e-mail server gestisce la posta elettronica.



Le risorse fornite da server specializzati possono essere messe a disposizione di diversi **client**. Una macchina client fornisce all'utente l'appropriata interfaccia per utilizzare questi server impiegando la potenza di calcolo locale per eseguire applicazioni locali.



Architettura Client-Server per DBMS

Differenti approcci sono stati proposti su come suddividere le funzionalità tra client e server. Una possibilità è di includere le funzionalità in un DBMS centralizzato a livello server. Un **server SQL** è fornito al client (ogni client):

- Formula la query.
- Fornisce l'interfaccia utente.
- Fornisce funzioni di interfaccia dei linguaggi di programmazione.

SQL è uno standard e i server SQL di diversi produttori possono accettare query SQL.

I client possono collegarsi al dizionario dei dati che include la distribuzione dei dati fra i vari server SQL.

Interazione fra Client e Server

Elaborazione di una query:

1. Il client parsa una query e la decomponе in un certo numero di query ai siti indipendenti.
2. Ogni server processa la query locale e manda il risultato al sito client.
3. Il client combina il risultato delle sottoquery per produrre il risultato della query sottomessa in origine.

In questo approccio, il server SQL è detto anche **transaction server** o *back-end machine*. Il client è detto **application processor** o *front-end machine*.

In un tipico DDBMS, si è soliti dividere i moduli software su tre livelli:

- **Software server**, responsabile per la gestione dei dati locali di un sito.
- **Software client**, gestisce l'interfaccia utente, accede al catalogo del database e processa tutte le richieste che richiedono l'accesso a più di un sito.
- **Software di comunicazione**, fornisce le primitive di comunicazione che sono usate dal client per trasmettere comandi e dati tra i vari siti (server).

XML e basi di dati in Internet

HTML non idoneo per specificare dati strutturati che sono estratti dal database, quindi viene utilizzato un nuovo linguaggio XML come standard per la strutturazione e lo scambio di dati attraverso il Web. Esso può essere usato per fornire informazioni riguardanti la struttura e il significato dei dati in essi contenuti.

- **XML DTD** (Document Type Definition): linguaggi per la specifica della struttura dei documenti XML.
- **XSL** (eXtended Stylesheet Language): linguaggio di formattazione per specificare aspetti di formattazione.

Dati strutturati, semi strutturati e non strutturati

- **Dati strutturati**: informazioni memorizzate nei database, rappresentate in un formato rigido.
- **Dati non strutturati**: caratterizzati da una presenza molto limitata di informazioni relative ai tipi di dati. Sono dati conservati senza uno schema.
- **Dati semi strutturati**: dati che possono avere una struttura che può essere non identica per tutte le informazioni. Le informazioni sono inframmezzate ai dati. Questi tipi di dati sono anche **detti dati auto descrittivi**. Possono essere rappresentati come **grafi diretti**: i nomi dello schema sono rappresentati dalle **etichette** (o tag) sugli archi orientati, i nodi interni rappresentano oggetti individuali o gli **attributi composti**, i nodi foglia rappresentano i valori degli **attributi di tipo semplice** (atomico).

Modello dati XML (ad albero)

Documento XML: oggetto base in XML. Per costruire un documento XML si utilizzano elementi e attributi. Gli **elementi** sono le parti di un documento dotate di un senso proprio. Un elemento è individuato da un tag iniziale e un tag finale. I nomi dei tag sono racchiusi tra <...> e i tag di fine sono caratterizzati da uno 'slash' </...>. Gli **attributi** sono informazioni aggiuntive sull'elemento che non fanno effettivamente parte del contenuto. Essi sono posti nel tag iniziale dell'elemento.

Gli **elementi complessi** vengono costruiti in modo gerarchico (contengono altri elementi), mentre gli **elementi semplici** contengono solamente valori.

È naturale vedere la corrispondenza tra la rappresentazione XML testuale e la struttura ad albero. I **nodi interni** dell'albero rappresentano gli elementi complessi, i **nodi foglia** rappresentano gli elementi semplici.

Per questo motivo il modello XML è chiamato **modello ad albero** o **modello gerarchico**.

I documenti XML possono essere classificati in tre tipi:

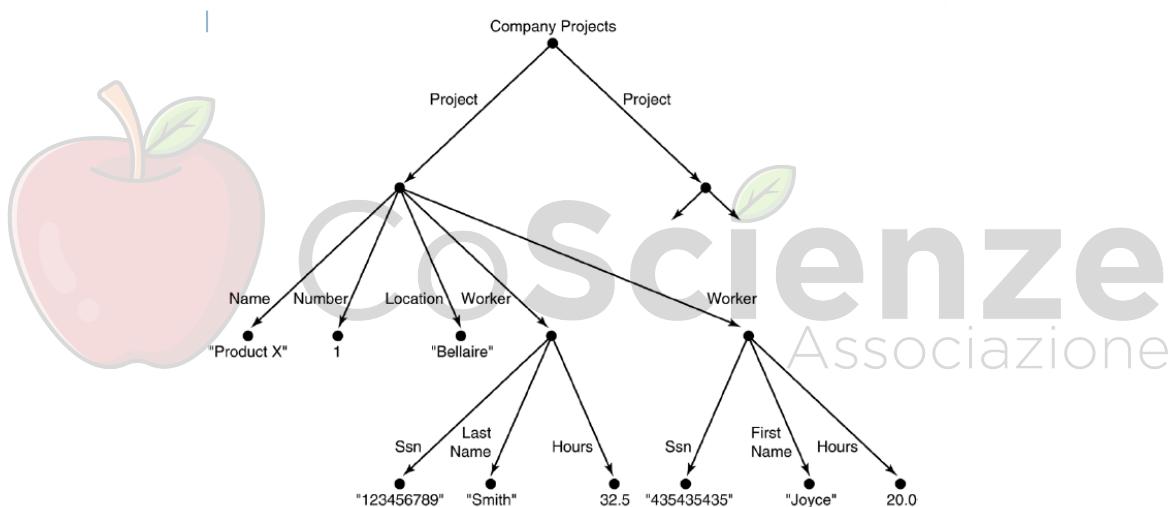
1. Documenti XML **incentrati sui dati**: documenti che hanno molti dati di dimensioni ridotte che seguono una specifica struttura e possono essere estratti da un database strutturato. Essi sono formattati come documenti XML per scambiarli o visualizzarli sul Web.
2. Documenti XML **incentrati sul documento**: documenti con grande quantità di testo (come news, articoli o libri). In questi documenti vi sono pochi, o nessun, dato strutturato.
3. Documenti XML ibridi: documenti che possono avere parti che contengono dati strutturati e altre parti prevalentemente testuali o non strutturate.

Documento XML ben formato (**well – formed**):

- Deve iniziare con la dichiarazione XML per indicare la versione XML utilizzata e altri attributi rilevanti;
- Deve seguire la struttura sintattica del modello ad albero, questo significa che deve contenere un **unico elemento radice**, e che ogni elemento deve contenere una coppia di tag di inizio e fine correlati tra loro, contenuti tra i tag dell'**elemento padre**;
- Un documento ben formato è **sintatticamente corretto**. Questo gli consente di essere processato da generici processori che attraversano il documento e creano la rappresentazione interna ad albero:
 - **DOM** (Document Object Model): consente ai programmi di manipolare la risultante rappresentazione ad albero corrispondente a documenti XML ben formati. Quando si usa DOM, l'intero documento deve essere analizzato anticipatamente.
 - **SAX**: consente di elaborare i documenti XML "al volo" notificando al programma di elaborazione quando si incontra un tag di inizio o fine.

Documento XML valido:

- Un criterio più forte per un documento è essere valido. Un documento valido è ben formato e deve essere scritto in modo che i nomi utilizzati nelle coppie di tag di inizio e fine siano coerenti con la struttura specificata nel file DTD o in un file XML schema.



Notazione XML DTD

- Un * che segue il nome dell'elemento significa che l'elemento può essere ripetuto **0 o più volte** nel documento. L'elemento in questo caso si dice essere **a valore multiplo (ripetuto) opzionale**.
- Un + che segue il nome dell'elemento significa che l'elemento può essere ripetuto **1 o più volte** nel documento. L'elemento può essere chiamato **obbligatorio a valore multiplo (ripetuto)**.
- Un ? che segue il nome dell'elemento significa che l'elemento può essere ripetuto **0 o 1 volta**. L'elemento può essere chiamato **a valore singolo (non ripetuto) opzionale**.
- Un elemento presente senza **nessuno** dei tre simboli precedenti deve comparire esattamente **1 volta**. Questi elementi sono chiamati **a valore singolo (non ripetuti) necessari**.
- Il tipo di un elemento viene specificato tra parentesi di seguito all'elemento stesso. Se le parentesi comprendono nomi di altri elementi, questi ultimi all'interno della struttura ad albero sono i figli dell'elemento a cui le parentesi si riferiscono. Se le parentesi includono la parola chiave #PCDATA o uno degli altri dati tipi di dato disponibili in XML DTD, l'elemento è un nodo foglia. PCDATA sta per parsed character data, analogo al tipo di dati stringa.
- Nella specifica degli elementi le parentesi possono essere nidificate.

- Un simbolo **barra** ($e_1 | e_2$) specifica che nel documento può comparire e_1 o e_2 .

La **dichiarazione di attributi** per ogni elemento dice: quali attributi può avere, che valore può assumere ciascun attributo, se esiste e qual è il valore di default.

Tipi di attributi

- CDATA: dati di tipo carattere;
- ($val_1 | val_2 | val_3$): un valore nella lista;
- ID: identificatore;
- IDREF, IDREFS: valore di un attributo di tipo ID nel documento (o insieme di valori);
- ENTITY, ENTITIES: nome (nomi) di entità;
- NMTOKEN, NMTOKENS: caso ristretto di CDATA (una sola parola o insieme di parole)

Vincoli sugli attributi

- #REQUIRED: il valore deve essere specificato;
- #IMPLIED: il valore può mancare;
- #FIXED "valore": se presente, deve coincidere con "valore";
- Default: si può specificare un valore come default, usato quando l'attributo è mancante

Limitazioni degli XML DTD

I tipi di dato del DTD non sono molto generali. DTD ha la sua particolare sintassi e pertanto richiede processori specializzati. Sarebbe vantaggioso specificare lo schema dei documenti XML usando le regole sintattiche di XML stesso in modo tale che gli stessi processori dei documenti XML possano processare le descrizioni dello schema XML. Tutti gli elementi DTD sono sempre forzati a seguire l'ordine del documento, pertanto non sono permessi elementi non ordinati. Un documento XML può dipendere da un solo DTD che definisce tutta la grammatica applicabile.

XML Schema

Lo scopo è definire gli elementi e la composizione di un documento XML in modo più efficace del DTD. XML Schema definisce regole riguardanti: elementi, attributi, gerarchia degli elementi, sequenza di elementi figli, cardinalità di elementi figli, tipi di dati per elementi e attributi, valori di default per elementi e attributi.

XSD

XSD (XML Schema Definition): schema di un tipo di documenti.

Gli XSD sono: estendibili (ammettono tipi riusabili definiti dall'utente), in formato XML, più ricchi e completi dei DTD, capaci di supportare tipi di dati diversi da PCDATA, capaci di gestire namespace multipli.

Documento XML con riferimento a XSD:

```
<?xml version="1.0"?>
<note
  xmlns="http://www.w3schools.com"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3schools.com/note.xsd">
  <to> Tove </to>
  <from> Jani </from>
  <head> Reminder </head>
  <body> Don't forget me this weekend! </body>
</note>
```

- **xmlns**: namespace di default.
- **xmlns:xsi**: URI che introduce l'uso dei tag di XML Schema.
- **xsi:schemaLocation**: dichiara dove reperire il file XSD (sempre attraverso URI).

Namespace XML

Definizione necessaria per identificare il particolare insieme di elementi (tag) del linguaggio XML schema usati specificando un file memorizzato in un sito Web.

```
<xs:schema  
xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Il nome del file è assegnato alla variabile **xs** usando l'attributo `xmlns`, questa variabile è usata come prefisso di tutti i comandi di XML schema.

Elementi semplici

Possono contenere solo testo (no elementi o attributi). Tipi: stringhe, numerici, date, time, boolean, ecc...

Esempi di definizione di elementi semplici in XSD:

```
<xs:element name="età" type="xs:integer"/>  
<xs:element name="cognome" type="xs:string"/>
```

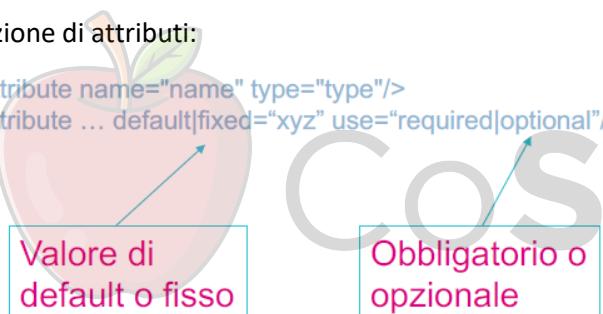
Esempi di elementi semplici XML:

```
<età> 65 </età>  
<cognome> Rossi </cognome>
```

Attributi

Definizione di attributi:

```
<xs:attribute name="name" type="type"/>  
<xs:attribute ... default|fixed="xyz" use="required|optional"/>
```



Valore di default o fisso

Obbligatorio o opzionale

Esempio di definizione di attributi:

```
<xs:attribute name="lang" type="xs:string"/>
```

Esempio di uso di un attributo:

```
<lastname lang="it"> qwerty </lastname>
```

Restrizioni

Consentono di dichiarare vincoli sui valori di un tipo elementare (detti tipi base):

```
<xs:element name="age">
<xs:simpleType>
<xs:restriction base="xs:integer">
<xs:minInclusive value="0"/>
<xs:maxInclusive value="100"/>
</xs:restriction>
</xs:simpleType>
</xs:element>
```

Enumerazione

Particolare tipo di restrizione che consente di enumerare i valori di un elemento o attributo:

```
<xs:element name="car">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="Audi"/>
<xs:enumeration value="Golf"/>
<xs:enumeration value="BMW"/>
</xs:restriction>
</xs:simpleType></xs:element>
```

Elementi complessi (complex elements)

Esistono quattro tipi di elementi complessi:

- vuoti (empty)
- contenenti solo altri elementi
- contenenti solo testo
- contenenti testo e/o altri elementi

```
<xs:complexType>
. . . element content . .
</xs:complexType>
```

Costrutto sequence

Sequenza (record): gli elementi devono apparire nell'ordine indicato:

```
<xs:complexType>
<xs:sequence>
<xs:element name="titolo" type="xs:string"/>
<xs:element name="autore" type="xs:string"
maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
```

La sequenza può contenere sia sotto elementi che attributi.

Costrutto all

Come la sequenza ma gli elementi possono apparire nel documento in qualsiasi ordine:

```
<xs:complexType>
<xs:all>
<xs:element name="titolo" type="xs:string"/>
<xs:element name="autore" type="xs:string"
maxOccurs="unbounded"/>
</xs:all>
</xs:complexType>
```

Costrutto choice

Sequenza (OR): gli elementi appaiono in alternativa nel documento:

```
<xs:complexType>
<xs:choice>
<xs:element name="capitolo" type="xs:string"
maxOccurs="unbounded"/>
<xs:element name="appendice" type="xs:string"
maxOccurs="unbounded"/>
</xs:choice>
</xs:complexType>
```

Elementi EMPTY

Gli elementi senza sotto elementi interni si dichiarano come complexType privi di sotto elementi (element):

```
<xs:element name="product">
<xs:complexType>
<xs:attribute name="prodid" type="xs:integer"/>
</xs:complexType>
</xs:element>
```

Specifiche della cardinalità

Indica la cardinalità dei sotto elementi. La sintassi prevede l'utilizzo di due attributi:

- **maxOccurs**: massimo numero di occorrenze. Per gli elementi multivaleure si deve impostare **maxOccurs="unbounded"**;
- **minOccurs**: minimo numero di occorrenze. Per i valori nulli si deve impostare **minOccurs=0**

Se non specificati, default=1:

```
<xs:complexType>
<xs:sequence>
<xs:element name="full_name" type="xs:string"/>
<xs:element name="child_name" type="xs:string"
maxOccurs="10" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
```

Definizione di tipi riusabili

Consente di definire tipi e riusarli per: tipare attributi oppure definire altri tipi.

Esempio di definizione di tipo (non di elemento):

```
<xs:complexType name="personinfo">
<xs:sequence>
<xs:element name="firstname" type="xs:string"/> <xs:element name="lastname"
type="xs:string"/>
</xs:sequence>
</xs:complexType>
```

Esempio di riuso del tipo per definire elementi/attributi:

```
<xs:element name="employee" type="personinfo"/>
<xs:element name="student" type="personinfo"/>
<xs:attribute name="member" type="personinfo"/>
```

Annotations and documentation

Gli elementi **xsd:annotation** e **xsd:documentation** sono utilizzati per fornire commenti ed altre descrizioni nel documento XML.

L'attributo **xml:lang** dell'elemento **xsd:documentation** specifica il linguaggio usato ("en" specifica la lingua inglese).

Specifiche delle chiavi

Per specificare una chiave primaria è utilizzato il tag **xsd:key**.

Per specificare le chiavi esterne è usato il tag **xsd:keyref**. Quando si specifica una chiave esterna, l'attributo **refer** del tag **xsd:keyref** indica la chiave primaria referenziata, mentre i tag **xsd:selector** e **xsd:field** specificano il tipo dell'elemento riferito e la chiave esterna.

Interrogazioni in XML

Come linguaggi di interrogazione XML sono emersi, tra le varie proposte, due standard:

- **XPath**: fornisce dei costruttori di linguaggio per la specifica di path expression, volti a identificare determinati nodi (o elementi) che corrispondono a particolari pattern contenuti nel documento XML.
- **XQuery**: linguaggio di interrogazione più generale che utilizza le espressioni XPath ma che possiede costrutti aggiuntivi.

XPath

Una espressione XPath restituisce una collezione di nodi che soddisfa pattern specificati in essa.

I nomi della espressione XPath sono i nomi dei nodi dell'albero del documento XML, che possono essere i nomi degli elementi o degli attributi, con l'aggiunta di condizioni di qualifica per limitarne il numero di risultati.

Nella specifica di un path sono utilizzati due separatori:

- Slash (/): inserito prima di un tag specifica che quel tag deve essere un figlio diretto del tag precedente;
- Doppio slash (//): indica che il tag può essere un discendente di qualunque livello

Si include il nome del file in ogni interrogazione XPath.

Path expression in XPath

L'idea è di usare una sintassi per "navigare" la struttura ad albero di un documento.

Una espressione XPath è una stringa contenente nomi di elementi e operatori di navigazione e selezione:

- . : nodo corrente;
- .. : nodo padre del nodo corrente;
- /: nodo radice, o figlio del nodo corrente;
- //: descendente del nodo corrente;
- @: attributo del nodo corrente;
- *: qualsiasi nodo;
- [p]: predicato (se l'espressione p, valutata, ha valore booleano);
- [n]: posizione (se l'espressione n, valutata, ha valore numerico).

Una path expression può iniziare con **doc(posizione_documento)** che restituisce l'elemento radice del documento specificato e tutto il suo contenuto.

A partire dalla radice del documento si possono specificare delle espressioni per estrarre il contenuto desiderato. Esempio:

`doc("libri.xml")/Elenco/Libro`

Restituisce la sequenza di tutti gli elementi di tipo Libro contenuti nel documento "libri.xml".

Condizioni su elementi/attributi

Esempio:

`doc ("libri.xml")/Elenco/Libro[Editore='Bompiani']/Titolo`

Restituisce la sequenza di tutti i titoli dei libri dell'editore 'Bompiani' che si trovano nel documento.

Ricerca di sotto-elementi a qualsiasi livello

Esempio:

`doc("libri.xml")//Autore`

Restituisce la sequenza di tutti gli autori che si trovano nel documento "libri.xml", annidati a qualunque livello.

Condizione sulla posizione dei sotto elementi e uso di wildcard

Esempio:

`doc ("libri.xml")/Elenco/Libro[2]/*`

Restituisce tutti i sotto elementi (*) contenuti nel secondo libro (Libro[2]) del documento "libri.xml".

Esempi XPath

1. **/company**: restituisce il nodo radice di **company** e tutti i suoi nodi discendenti, ciò implica che restituisce l'intero documento XML;
2. **/company/department**: restituisce tutti i nodi **department** ed il loro sottoalbero;
3. **//employee[employeeSalary > 70000]/employeeName**: restituisce tutti i nodi **employeeName** che sono figli diretti del nodo **employee**, dei nodi **employee** che abbiano un figlio **employeeSalary** il cui valore è maggiore di 70000;
4. **/company/employee[employeeSalary > 70000]/employeeName**: restituisce lo stesso risultato del (3), ma specificando l'intero nome del path;
5. **/company/project/projectWorker[hours ge 20.0]**: restituisce tutti i nodi **projectWorker** e i loro sottoalberi dei nodi che abbiano il nodo **hours** maggiore o uguale di 20.0

XQuery

Usa le espressioni XPath ma ha ulteriori costrutti. Una interrogazione XQuery è un'espressione complessa che consente di estrarre parti di un documento e costruire un altro documento. XQuery consente di specificare interrogazioni su uno o più documenti XML. La forma tipica di una interrogazione XQuery è conosciuta come **espressione FLWR**, che indica le principali clausole di XQuery:

- FOR: itera i valori delle variabili su sequenze di nodi;
- LET: variabili legate a collezioni di nodi;
- WHERE: esprime condizioni di qualificazione sui legami;
- RETURN: specificazione del risultato dell'interrogazione

Espressioni FLWR

In una singola XQuery si possono avere **0 o più** istanze della clausola **FOR**, **0 o più** istanze della clausola **LET**, **0 o 1** istanza di ciascuna delle clausole **WHERE** e **ORDER**, esattamente **1** istanza della clausola **RETURN**. Le variabili sono prefissate da \$, la clausola LET assegna una variabile a una particolare espressione.

Espressioni FOR

Esempio:

```
for $libro in doc("libri.xml")//Libro
    return $libro
```

La clausola **for** valuta la path expression (**doc("libri.xml")//Libro**), che restituisce una sequenza di elementi, e la variabile **\$libro** itera all'interno della sequenza, assumendo ad ogni iterazione il valore di un nodo diverso. La clausola **return** costruisce il risultato, in questo caso restituisce ogni valore legato a **\$libro**, cioè tutti i libri del documento.

Le espressioni FOR possono essere annidate.

```
for $libro in doc("libri.xml")//Libro
    for $autore in $libro/Autore
        return $autore
```

Semantica: per ogni valore di **\$libro**, per ogni valore di **\$autore** del libro corrente, inserisci nel risultato il **\$autore**.

Espressioni LET

Consentono di introdurre nuove variabili. Esempio:

```
let $libri := doc("libri.xml")//Libro
return $libri
```

La clausola **let** valuta l'espressione, **//Libro**, e assegna alla variabile **\$libri** l'intera sequenza restituita.

La valutazione di una clausola let assegna alla variabile un singolo valore, cioè l'intera sequenza dei nodi che soddisfano l'espressione.

Clausola WHERE

La clausola **WHERE** esprime una condizione, solo le tuple che soddisfano tale condizione vengono utilizzate per invocare la clausola **RETURN**. Le condizioni nella clausola WHERE possono contenere diversi predicati connessi da AND o OR. Il **not()** è realizzato tramite una funzione che inverte il valore di verità.

Esempio:

```
for $libro in doc("libri.xml")//Libro
where $libro/Editore="Bompiani"
and $libro/@disponibilità="S"
return $libro
```

Restituisce tutti i libri con Editore='Bompiani' e disponibilità='S'.

Clausola RETURN

Genera l'output di un'espressione FLWR che può essere: un nodo (**<Autore>F. Dürrenmatt</Autore>**), una foresta ordinata di nodi

```
(<Autore>J.R.R. Tolkien</Autore>
<Autore>Umberto Eco</Autore>
<Autore>F. Dürrenmatt</Autore>),
```

un valore testuale (PCDATA, esempio **F. Dürrenmatt**).

Può contenere dei costruttori di nodi, dei valori costanti, riferimenti a variabili definite nelle parti FOR e LET, ulteriori espressioni annidate.

Un **costruttore di elemento** consta di un tag iniziale e di un tag finale che racchiudono una lista opzionale di espressioni annidate che ne definiscono il contenuto.

Esempio:

```
for $libro in doc("libri.xml")//Libro
where $libro/Editore="Bompiani"
return <LibroBompiani>
      { $libro/Titolo }
    </LibroBompiani>
```

Restituisce:

```
<LibroBompiani><Titolo>Il Signore degli Anelli</Titolo></LibroBompiani>
<LibroBompiani><Titolo>Il nome della rosa</Titolo></LibroBompiani>
```

Esempio:

```
for $libro in doc("libri.xml")//Libro
where $libro>Editore="Bompiani"
return <Libro-Bompiani>
    { $libro/Titolo/text() }
</Libro-Bompiani>
```

text() estrae il solo contenuto PCDATA di un elemento.

Risultato:

```
<Libro-Bompiani>Il Signore degli Anelli</Libro-Bompiani>
<Libro-Bompiani>Il nome della rosa</Libro-Bompiani>
```

Ordinare il risultato

Esempio:

```
for $libro in doc("libri.xml")//Libro
order by $libro/Titolo
return
    <Libro>
        { $libro/Titolo,
          $libro/Editore }
    </Libro>
```

I libri vengono ordinati rispetto al titolo. I matching della variabile sono riordinati prima di essere passati alla clausola return per generare il risultato.

Funzioni di aggregazione

Esempio:

```
for $e in doc("libri.xml")//Editore
let $libro := doc("libri.xml")//Libro[Editore = $e]
where count($libro) > 100
return $e
```

Restituisce gli editori con più di 100 libri in elenco. La “cardinalità” del risultato, cioè il numero di editori restituiti, dipende da quante volte è eseguita la return, e questo, a sua volta, dipende dalla clausola for. (Ogni editore risultante, viene restituito oltre cento volte).

Esempio di XQuery

```
FOR $x IN
doc("www.company.com/info.xml")/company/project[projectNumber=5]/projectWorker,
$y IN doc("www.company.com/info.xml")/company/employee
WHERE $x/hours gt 20.0 AND $y.ssn=$x.ssn
RETURN <res> $y/employeeName/firstName, $y/employeeName/lastName, $x/hours </res>
```

Questa query illustra come può essere eseguita un’operazione di **join** usando più di una variabile. La variabile **\$x** è legata a ogni elemento **projectWorker** che è figlio del progetto con **projectNumber=5**, mentre la variabile **\$y** è legata a ogni elemento **employee**. La condizione di join confronta i valori **ssn** al fine di recuperare nome e cognome dell’impiegato e numero di ore.

Database NoSQL

RDBMS è l'acronimo di relational database management system e indica un database management system basato sul modello relazionale. Le motivazioni alla base dell'utilizzo di un RDBMS sono:

- Schema predefinito per lo storage di dati strutturati;
- Struttura BCNF già familiare;
- Strong consistency;
- Transazioni;
- Maturi e accuratamente testati;
- Facile adozione/integrazione;
- Basati sulle proprietà ACID (atomicity, consistency, isolation, durability);
- Data retrieval: SQL – versatile e potente;
- Scalabilità verticale: se volessimo rendere un database SQL scalabile, l'unica alternativa sarebbe quella di potenziare l'hardware sul quale il DBMS è installato

Cosa ha spinto a database NoSQL:

- **BIG USERS:** un gran numero di utenti combinato con pattern di gestione dei dati fortemente dinamici, stando guidando il bisogno di nuovi database scalabili. Con le tecnologie relazionali, è difficile raggiungere la scalabilità dinamica richiesta dalle applicazioni per garantire il livello di performance richiesto dagli utenti;
- **BIG DATA:** la rapida crescita dell'ammontare dei dati e la natura degli stessi.

Sempre più aziende innovative si affidano ai database NoSQL richiedendo una tecnologia che sia in grado di scalare con i milioni di "cose" (the internet of things) connesse ad Internet.

NoSQL non è uno specifico linguaggio, ma è il termine che raggruppa un insieme di tecnologie per la persistenza dei dati che funzionano in modo sostanzialmente diverso dai database relazionali, quindi non rispettano una o più caratteristiche dei RDBMS.

Possono avere le caratteristiche più disparate: alcuni non utilizzano il modello relazionale, altri usano tabelle e campi ma senza schemi fissi, alcuni non permettono vincoli di integrità referenziale, altri non garantiscono transazioni ACID, oppure ci sono varianti che combinano le precedenti.

Alcuni database NoSQL garantiscono solo alcune proprietà ACID. Per esempio, non sempre è garantita la **consistenza** (si parla in questo caso di **eventual consistency**), ossia ci può essere una certa latenza prima che una modifica al database sia visibile. Altri non garantiscono la **durabilità**: ad esempio, in alcuni database distribuiti il malfunzionamento di un nodo dopo una transazione potrebbe impedire la corretta sincronizzazione di tutta la rete.

Queste proprietà vengono rilassate per fornire performance migliori.

Caratteristiche dei NoSQL:

- **Non relazionali:** i database NoSQL sono **schemaless** e consentono di memorizzare attributi **on the fly**, anche senza averli definiti a priori, questo per consentire la memorizzazione di dati fortemente dinamici;
- **Distribuiti:** la flessibilità nella clusterizzazione e nella replica dei dati permette di distribuire su più nodi lo storage, in modo da realizzare potenti sistemi **fault tolerant**;
- **Scalabili orizzontalmente:** architetture enormemente scalabili che consentono di memorizzare e gestire una grande quantità di informazioni;
- **Open-source:** filosofia alla base del movimento NoSQL;
- **Grossi volumi di dati;**

- i database NoSQL sono generalmente ideati per richiedere una minore manutenzione rispetto a un sistema RDBMS che, invece, può essere mantenuto solamente con l'assistenza di amministratori esperti e costosi;
- velocità di risposta alle query;
- le proprietà ACID non sono richieste;
- CAP theorem.

Transazioni BASE

I NoSQL si basano sul modello BASE:

- **Basically Available:** garantire la disponibilità dei dati anche in presenza di fallimenti multipli. L'obiettivo è raggiunto attraverso un approccio fortemente distribuito;
- **Soft State:** abbandonano il requisito della consistenza dei modelli ACID quasi completamente. La consistenza è un problema dello sviluppatore e non deve essere gestita dal database.
- **Eventually Consistent:** l'unico requisito riguardante la consistenza è garantire che, ad un certo momento, nel futuro, i dati possano convergere ad uno stato consistente.

Brewer's CAP Theorem

Un sistema distribuito è in grado di supportare solamente due tra le seguenti caratteristiche:

- **Consistency:** tutti i nodi vedono lo stesso dato nello stesso tempo;
- **Availability:** ogni operazione deve sempre ricevere una risposta;
- **Partition tolerance:** capacità di un sistema di essere tollerante ad una aggiunta, una rimozione di un nodo nel sistema distribuito o alla disponibilità di un componente singolo.

Vantaggi e svantaggi del NoSQL

Un noto problema di quando si sviluppa con linguaggi orientati ad oggetti è il cosiddetto **O/R impedance mismatch**, ossia i due modelli, relazionale e ad oggetti, sono molto diversi tra loro (può generare problematiche quali il polimorfismo oppure la conversione dei tipi di dati).

Svantaggio più significativo dei database NoSQL è la carenza di tool: essendo una tecnologia abbastanza recente esistono pochi strumenti di gestione e sviluppo, a contrario del SQL.

Vantaggi

- **Strutturazione dei dati:** maggiore centralità alle informazioni e alla loro varietà, supportando l'uso di dati non omogenei pur mantenendo le possibilità di interrogazione, analisi ed elaborazione efficiente;
- **Scalabilità:** i database NoSQL sono generalmente basati su strutture fisiche che si prestano meglio alla distribuzione dei dati su più nodi di una rete (*sharding*), permettendone un'espandibilità maggiore;
- **Prestazioni:** la maggiore distribuzione dei dati sulle reti permette migliori performance;
- **Flessibilità nella progettazione:** la struttura flessibile usata nei database NoSQL non obbliga ad una stereotipazione dei dati durante la progettazione.

Svantaggi (sfide)

- **Maturità:** i sistemi RDBMS sono in esercizio da tanto tempo;
- **Supporto:** tutte le aziende che sviluppano e vendono RDBMS forniscono un alto livello di supporto alle aziende
- **Business intelligence:** spesso i tool per la BI non supportano connettività con i database NoSQL;
- **Amministrazione:** al giorno d'oggi sono necessarie grosse competenze per l'installazione e manutenzione dei database NoSQL;

- **Expertise:** è più semplice trovare un esperto amministratore RDBMS che NoSQL.

Conviene utilizzare database NoSQL quando:

- La struttura dati non è definibile a priori;
- I dati disposti nei vari oggetti sono molto collegati tra loro;
- È necessario interagire molto frequentemente con il database;
- Si necessita di prestazioni più elevate.

Non rinunciare al modello relazionale quando:

- DBMS relazionali consolidati da decenni, supportati da una gran varietà di strumenti;
- Si devono modellare dati molto strutturati;
- La duttilità del NoSQL non è un requisito fondamentale;

Classificare i DBMS NoSQL

I DBMS NoSQL possono essere classificati in quattro categorie:

- Document data store;
- Key – value data store;
- Graph – based data store;
- Column – oriented data store.

Document data store (es. MongoDB)

- Utilizzano dati non strutturati;
- Schema – less;
- Supporto a diversi tipi di documento;
- Ogni documento è identificato da una chiave primaria
- Scalabilità orizzontale

La rappresentazione dei dati avviene attraverso strutture simili ad oggetti, dette **documenti**, ognuno dei quali possiede delle proprietà che rappresentano le informazioni. I documenti non devono seguire una struttura rigida fissa. Il documento può essere visto come l'equivalente di un record delle tabelle. I documenti possono essere messi in relazione tra loro con dei riferimenti.

Key – value data store (es. Redis)

- Utilizza un associative array, chiave – valore, come modello per lo storage;
- Storage, update e ricerca basato sulle chiavi;
- Tipi di dati primitivi familiari ai programmati;
- Semplice;
- Veloce recupero dei dati;
- Grandi moli di dati

Vengono costruiti come dizionari o mappe, in cui viene inserita una coppia chiave (l'identificatore) – valore (l'informazione vera e propria). Il loro utilizzo principale è offrire la possibilità di effettuare ricerche rapide di singoli blocchi di informazione. Sono database che puntano tutto sulla velocità.

Graph – based data store (es. neo4j)

- Utilizza nodi (entità), proprietà (attributi) e archi (relazioni);

- Modello logico semplice e intuitivo;
- Ogni elemento contiene un puntatore all'elemento adiacente;
- Attraversamento del grafo per trovare i dati;
- Efficiente per la rappresentazione di reti sociali o dati sparsi;
- Relazioni tra i dati centrali

Le informazioni possono essere custodite sia nei nodi che negli archi. La forza di questa tipologia è tutto l'insieme del valore informativo che si può estrapolare ricostruendo i percorsi attraverso il grafo.

Column – oriented data store (es. cassandra)

- I dati sono nelle colonne anziché nelle righe;
- Un gruppo di colonne è chiamato famiglia e vi è un'analogia con le tabelle di un database relazionale;
- Le colonne possono essere facilmente distribuite;
- Scalabile;
- Performante;
- Fault – tolerant.

La rapida aggregazione che permettono è stata apprezzata dai grandi produttori di motori di ricerca e Social Network (alcune delle principali soluzioni di questo tipo, Cassandra, Big Table, SimpleDB, sono state realizzate da colossi come Facebook, Google e Amazon).

MongoDB

DB NoSQL orientato ai documenti, nato nel 2007 in California come servizio di un progetto più ampio, per poi diventare un prodotto indipendente e open – source. Memorizza i documenti in JSON.

I database a documenti seguono un modello che struttura le informazioni in aggregazioni di dati, i **documenti**. I documenti sono disposti in strutture dati lineari dette **collection** o **collezioni**, e possono essere collegati tra loro mediante riferimenti.

Il documento è fondamentalmente un albero che può contenere molti dati, anche annidati.

Le collezioni, in cui vengono raggruppati i documenti, possono essere anche eterogenee, non c'è uno schema fisso per i documenti.

Tra le collezioni non ci sono relazioni o legami garantiti da MongoDB.

Le caratteristiche chiave di MongoDB sono:

- Consente servizi di alta disponibilità, perché la replicazione di un database (**replica set**) può avvenire in modo molto semplice;
- Garantisce la scalabilità automatica, ossia la possibilità di distribuire (**sharding**) le collezioni in cluster di nodi.

Si adatta a molti contesti, in generale quando si manipolano grandi quantità di dati eterogenei e senza uno schema, non è, invece, opportuno quando si devono gestire molte relazioni tra oggetti.

Progettazione

1. Individuazione delle possibili entità;
2. Per quanto possibile, selezionare un insieme di proprietà che dovrebbero apparire nei documenti. Queste attività non sono obbligatorie, in quanto molti MongoDB (e altri database NoSQL), non richiedono una definizione apriori della struttura interna dei documenti. Su MongoDB i documenti sono rappresentati in BSON (formato binario derivato e molto simile a JSON). Ogni documento è identificato da un valore univoco, *ObjectID*;
3. Individuazione dei documenti da innestare in altri: innestando documenti in altri, si ha da un lato il vantaggio che abbiamo in un unico documento tutte le informazioni relative ad una entità, d'altro canto, però, potrebbe comportare una ridondanza dei dati

MongoDB è fortemente orientato all'interazione con le applicazioni e non contempla alcune funzionalità come il controllo sui valori inseriti.

Progettare database key – value

Sono ispirati alla struttura dati di dizionario. Tutto ciò che viene archiviato in questi database dovrà essere fornito sotto forma di coppia chiave/valore, dove il valore è l'informazione vera e propria, e la chiave è ciò che ne consente il recupero. Un database key – value può essere considerato come un'unica grande mappa (o dizionario). Il DBMS di questo tipo più diffuso è Redis.

I database di questo tipo sono stati spesso criticati perché troppo di nicchia. Vengono usati per operazioni di cache o immagazzinamento di dati di sessione. Ma si prestano anche ad altri scenari applicativi, ad esempio ogni tabella di un database relazionale vive di una struttura key – value dove la “key” è costituita dalla chiave primaria e il “value” è rappresentato dagli altri campi della stessa riga.

In un database key – value, la chiave è l'elemento centrale della memorizzazione, è necessario scegliere correttamente le chiavi. Le chiavi in Redis sono *binary safe*, pertanto possono essere delle sequenze alfanumeriche o il contenuto di un file non testuale, è importante, però, non utilizzare chiavi di dimensioni eccessive per non penalizzare le prestazioni in fase di ricerca. Si consiglia di definire chiavi leggibili, buona norma prevede di anteporre un prefisso seguito da un separatore (spesso si usa “:”).

Database a grafo

Questi si differenziano dai RDBMS per il modello di dati. Neo4j (2003) è il database a grafo più utilizzato, seguito da OrientDB.

Caratteristiche principali: modello di dati a grafo orientato con proprietà chiave – valore, supportate sia sui nodi che sulle relazioni (modello detto Property Graph); transazioni ACID; linguaggio di query dichiarativo denominato Cypher; driver per interagire con i linguaggi di programmazione più diffusi.

Modello Property Graph: i nodi e le relazioni che possono essere memorizzati in Neo4j possono avere un elenco di proprietà chiave – valore, solitamente la chiave è una stringa che rappresenta il nome della proprietà, mentre il valore può essere un numero (intero o reale), una stringa, un array di numeri o stringhe. Il modello prevede che i **nodi** abbiano un **ID** univoco assegnato automaticamente da Neo4j al momento della creazione, possano avere una o più **Label** per classificarli e indicizzarli. I nodi sono rappresentati da parentesi tonde, le proprietà si scrivono con la stessa sintassi di JSON, cioè un elenco di attributi *chiave:valore* separati da “,” e racchiusi in {}. Ogni **relazione** ha necessariamente un **tipo**, specificato dall'utente in fase di creazione, ha necessariamente una **direzione** (cioè una relazione va da un nodo a un altro) e non è possibile creare relazioni senza verso.

OrientDB

Database multi – model, modello ibrido con funzionalità NoSQL, prodotto e sviluppato da Orient Technologies. Supporta: **Object Model** (permette di definire classi di dati, supporta ereditarietà e polimorfismo), **Document Model** (permette di raccogliere i documenti in collezioni legate tra loro da link), **Graph Model** (raccoglie documenti in nodi collegati tra loro da relazioni), **Key – value Model** (immagazzinare i dati in strutture indicizzate tramite chiavi).

OrientDB è realmente Multi – Model, nel senso che i vari approcci sono veri modelli di gestione dei dati di cui il motore del DBMS è stato dotato. La varietà di usi cui si presta è uno dei fattori determinanti della sua diffusione.

Tra i vari paradigmi NoSQL non ne esiste uno migliore, ma a seconda delle circostanze si possono sfruttare i vantaggi di un approccio piuttosto che di altri.

Basi di dati semantiche, SPARQL e Linked Open Data

Arricchimento semantico della basi di dati

Presente fin dagli inizi dell'evoluzione del settore. L'uso dei modelli semantici è alla base di vari processi, quali:

- Durante la progettazione concettuale.
- Gestione dei dati e delle interrogazioni ad un livello astratto.
- Sistemi che supportano direttamente i modelli semantici.

Spinta allo sviluppo degli aspetti semantici direttamente nei sistemi di gestione dati è stata:

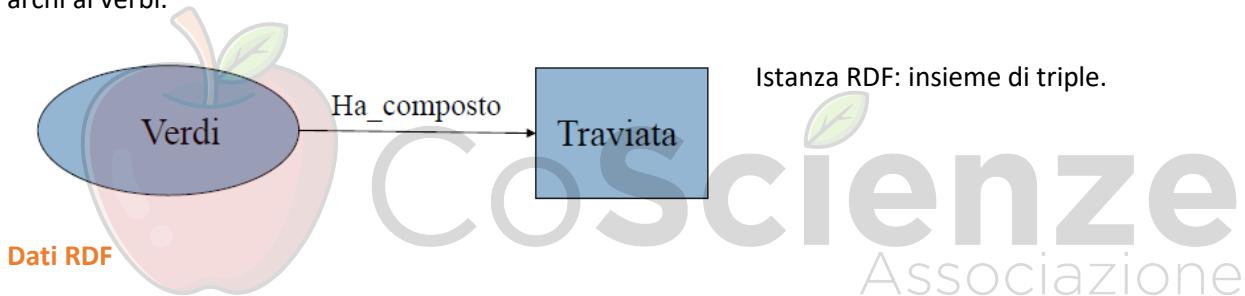
- Evoluzione dei linguaggi e delle basi dati ad oggetti verso la fine degli anni Ottanta;
- Evoluzione dei <<Semantic Web>>

Sviluppo delle cosiddette raccolte aperte e connesse <<Linked Open Data>>.

Modello RDF (Resource Description Framework)

È il modello astratto proposto dal W3C per esprimere affermazioni sul mondo. Le informazioni vengono rappresentate sotto forma di triple, che possono essere interpretate come **soggetto-verbo-oggetto**.

Una tripla può essere rappresentata come un grafo, in cui i nodi corrispondono ai soggetti e agli oggetti, gli archi ai verbi.



I dati rappresentabili in RDF sono di tre categorie:

- **IRI** (Internationalized Resource Identifier): sono stringhe che identificano univocamente tutte le risorse disponibili sul Web.
- **Letterali**: descrivono le risorse presenti nell'istanza RDF. È possibile assegnare ai letterali un tipo; se inclusi tra apici sono implicitamente di tipo *stringa*, mentre numeri senza apici, con segno, sono implicitamente di tipo *intero*.
- **Blank Nodes**: identificatori usati per rappresentare risorse anonime.

Nelle triple RDF il primo termine (soggetto) può essere un IRI o un blank node, il secondo termine (predicato) è un IRI, e il terzo termine (oggetto) può essere un IRI, un letterale o un blank node.

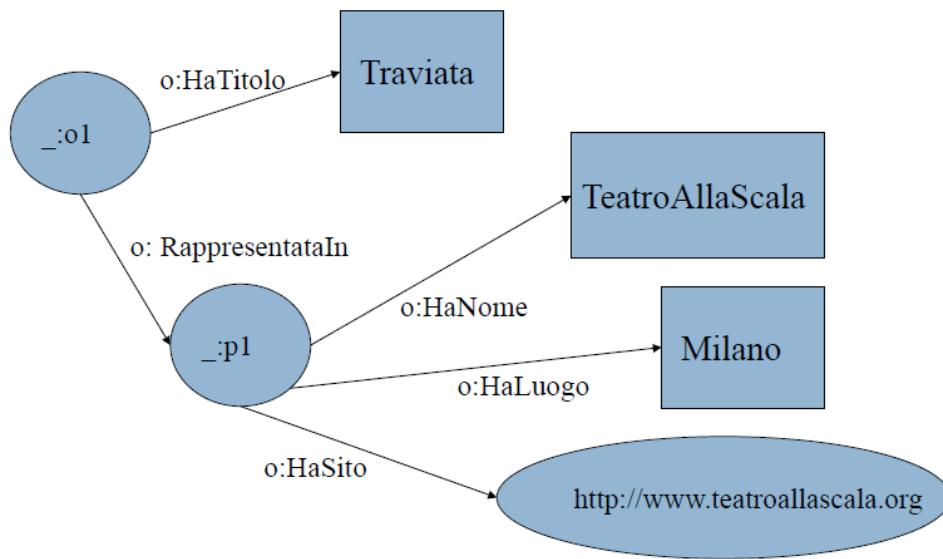
Namespace nel modello RDF

È possibile utilizzare uno o più vocabolari per i termini usati nell'istanza RDF; introdotti dall'istruzione <<prefix>> che li associa ad un prefisso da usare nelle definizioni e query:

@prefix o: <http://.....>

Nel definire un'istanza RDF, si possono indicare più vocabolari.

Esempio istanza RDF



Istanza RDF: Notazione N3

```

_:o1  o:HaTitolo 'Traviata';
      o:RappresentataIn _:p1.

_:p1  o:HaNome 'TeatroAllaScala';
      o:HaLuogo 'Milano';
      o:HaSito http://ecc...
  
```

Viene adottata la notazione **N3**, che utilizza la punteggiature per sintetizzare le parti comuni delle triple. Quando due triple condividono il soggetto si usa il separatore ‘;’. Quando condividono sia soggetto sia predicato, si utilizza il separatore ‘’.

Oltre alla notazione N3, è possibile utilizzare il formato XML per la definizione di una istanza RDF; tale notazione prende il nome di **RDF/XML** ed è stata standardizzata dal W3C.

Istanza RDF: Rappresentazione XML

```

<?xml version="1.0"?>
<rdf:RDF xmlns:o="http://www.polimi.it/ceri/opera"
           xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:ceri="http://www.polimi.it/ceri"/>

  <rdf:Description>
    <ceri:operaHaTitolo>Traviata</ceri:operaHaTitolo>
    <ceri:operaRappresentataIn>
      <rdf:Description>
        <ceri:operaHaNome>TeatroAllaScala</ceri:operaHaNome>
        <ceri:operaHaLuogo>Milano</ceri:operaHaLuogo>
        <ceri:operaHaSito rdf:resource="http://www.teatroallascala.org"/>
      </rdf:Description>
    </ceri:operaRappresentataIn>
  </rdf:Description>
</rdf:RDF>
  
```

Altro esempio di istanza RDF

```
@prefix o: <http://www.polimi.it/ceri/opera>
_:Ring o:compostoDa 'Wagner' .
_:Ring o:prodottoDa 'ScalaDiMilano' .
_:Ring o:condottoDa 'DanielBaremoim' .
_:Ring o:messolnScenaDa 'GuyCassiers' .
_:Ring o:haTitolo 'Ring'.
```

```
_:Ring o:comprende _:OroDelReno,
      _:Valchiria,
      _:Sigfrid,
      _:Crepuscolo.
```

```
_:OroDelReno o:haTitolo 'Oro Del Reno' ;
      o:dura 150 ;
      o:haPersonaggio 'Wotan' .
```

Altro esempio di istanza RDF

```
_:Valchiria o:haTitolo 'Valchiria' ;
      o:dura 310 ;
      o:haAtti 3 ;
      o:haPersonaggio 'Brunilde' ;
      o:haPersonaggio 'Wotan' ;
      o:haPersonaggio 'Sieglinde' .
```

```
_:Sigfrido o:haTitolo 'Sigfrido' ;
      o:dura 320 ;
      o:haAtti 3 ;
      o:haPersonaggio 'Brunilde' ;
      o:haPersonaggio 'Sigfrido' .
```

```
_:Crepuscolo o:haTitolo 'Crepuscolo degli Dei' ;
      o:dura 360 ;
      o:haAtti 3 ;
      o:haPrologo 'si' ;
      o:haPersonaggio 'Brunilde' ;
      o:haPersonaggio 'Sigfrido' ;
      o:haPersonaggio 'Waltraute' .
```

RDF Schema

Estensione di RDF come raccomandazione del W3C dal 2004. Esso aggiunge a RDF costrutti per descrivere metadati, ovvero struttura e proprietà di istanze RDF. I principali costrutti messi a disposizione da RDFS sono le classi e le proprietà:

- Una classe RDFS consente di definire insiemi di risorse di un'istanza RDF che hanno caratteristiche comuni.
- Una proprietà RDFS consente invece di definire il soggetto e l'oggetto dei predici in un'istanza RDF.

RDF Schema: costrutti

rdfs:Resource Tutto ciò che viene descritto in RDF è detto *risorsa*. Ogni risorsa è istanza della classe rdfs:Resource.

rdfs:Literal Sottoclasse di rdfs:Resource, rappresenta un letterale, una stringa di testo.

rdf:Property Rappresenta le proprietà. È sottoclasse di rdfs:Resource.

rdfs:Class Corrisponde al concetto di *tipo* e di *classe* della programmazione object-oriented. Quando viene definita una nuova classe, la risorsa che la rappresenta deve avere la proprietà rdf:type impostata a rdfs:Class.

rdfs:subClassOf Specifica la relazione di ereditarietà fra classi. Questa proprietà può essere assegnata solo a istanze di rdfs:Class. Una classe può essere sottoclasse di una o più classi (*ereditarietà multipla*)

rdfs:range (codominio) Usato come predicato di una risorsa r, indica le classi che saranno oggetto di un'asserzione che ha r come predicato.

rdfs:domain (dominio) Usato come predicato di una risorsa r, indica le classi (soggetto) a cui può essere applicata r.

rdfs:subPropertyOf Specifica la relazione di ereditarietà fra proprietà.

RDF Schema: Esempi

Esempio di classe:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .  
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .  
@prefix o: <http://www.polimi.it/ceri/opera> .  
o:Compositore rdf:type rdfs:Class.  
_:Wagner rdf:type o:Compositore.
```

Esempio di proprietà:

```
o:nome a rdfs:Property;  
        rdfs:domain o:Compositore;  
        rdfs:range rdfs:Literal.  
_:Ring a o:Opera.  
o:compostoDa a rdfs:Property;  
        rdfs:domain o:Opera;  
        rdfs:range o:Compositore.
```

È possibile definire in RDFS relazioni di generalizzazione tra classi.

```
o:Artista rdf:type rdfs:Class.  
o:Compositore rdfs:subClassOf o:Artista.
```

La semantica di RDFS definisce l'ereditarietà: se i compositori sono artisti e gli artisti sono persone, si deduce che i compositori sono persone; se le persone possiedono come proprietà una città e una data di nascita, anche i direttori d'orchestra hanno queste proprietà. Questi meccanismi di inferenza vengono chiamati **entailments**.

Altre estensioni semantiche

Una maggior ricchezza semantica rispetto ad RDFS è offerta dal Web Ontology Language (OWL). Esso è un linguaggio di markup per rappresentare esplicitamente le *ontologie* (ovvero il significato dei termini e relazioni tra i termini). L'obiettivo è supportare l'elaborazione automatica del contenuto delle informazioni dei documenti scritti in OWL ed il *reasoning* su di essi. Esistono tre versioni:

- OWL Lite pone delle restrizioni sui costrutti OWL utilizzabili che garantiscono un calcolo efficiente delle inferenze.
- OWL DL estende OWL Lite garantendo solo che tutte le inferenze possono essere calcolate in un tempo finito.
- OWL Full fornisce il massimo dell'espressività senza garantire la computabilità completa degli entailments.

SPARQL (Simple Protocol and RDF Query Language)

È un linguaggio di interrogazione simile a SQL proposto dal W3C per interrogare dati descritti in RDF, ricevendo il risultato in un formato XML. Esistono quattro tipi di query:

- SELECT per interrogazioni classiche.
- DESCRIBE, per ottenere una descrizione delle risorse presenti presso un database RDF interrogabili in SPARQL (detto <>endpoint<>).
- ASK, per sapere se specifici termini sono disponibili nell'endpoint.
- CONSTRUCT, per costruire un nuovo grafo RDF a partire da una interrogazione.

Vi sono due versioni del linguaggio:

- **1.0** (raccomandazione W3C del 2008);
- **1.1** (nuova versione più potente del 2013).

Sintassi

Una query **SPARQL 1.0** è composta da cinque parti:

- La clausola opzionale PREFIX per introdurre vocabolari.
- Il risultato prodotto dalla query (una delle quattro clausole SELECT, DESCRIBE, ASK, e CONSTRUCT).
- Gli endpoint consultati, tramite le clausole FROM e FROM NAMED.
- La parte centrale della query, introdotta dalla clausola WHERE consente di esprimere condizioni di pattern matching tra la query stessa e il grafo RDF su cui la query opera. Elemento centrale del pattern matching è il **triple pattern**.
- Modificatori opzionali, che includono i costrutti ORDER BY, DISTINCT e LIMIT.

Triple e graph patterns

Un **triple pattern** è una tripla nelle cui posizioni è possibile far comparire, in aggiunta a IRI, letterali e blank nodes, anche variabili, introdotte dal simbolo '?'; durante la valutazione delle query le variabili vengono <>legate<> (binding). Ad esempio

< ?o1 o:HaPersonaggio 'Sigfrido' >

Valutato sull'istanza RDF, la variabile o1? È legata ai blank node _:Sigfrido e _:Crepuscolo.
Possono essere combinati insieme vari triple pattern, formando un **graph pattern**.

Query SPARQL: 1

Estrarre i nomi delle opere che hanno Wotan come personaggio.

PREFIX o: <<http://www.polimi.it/ceri/opera>>

SELECT ?t

FROM <<http://www.polimi.it/ceri/ring.rdf>>

WHERE {

?o o:haPersonaggio 'Wotan' .

?o o:haTitolo ?t .

}

?t
'Oro del Reno'
'Valchiria'

Query SPARQL: 2

Estrarre i titoli delle opere e i nomi dei personaggi che hanno gli stessi personaggi che compaiono in Valchiria.

PREFIX o: <<http://www.polimi.it/ceri/opera>>

SELECT ?t, ?p

FROM <<http://www.polimi.it/ceri/ring.rdf>>

WHERE {

?o1 o:haPersonaggio ?p .

?o2 o:haPersonaggio ?p .

?o1 != ?o2 .

?o1 o:haTitolo ?t .

?o2 o:haTitolo 'Valchiria' .

}

?t	?p
'Oro del Reno'	'Wotan'
'Sigfrido'	'Brunilde'
'Crepuscolo degli Dei'	'Brunilde'

Query SPARQL: 3

Estrarre i titoli delle opere e i nomi dei personaggi di tutte le coppie di opere che hanno un personaggio in comune.

PREFIX o: <<http://www.polimi.it/ceri/opera>>

SELECT ?t1, ?t2, ?p

FROM <<http://www.polimi.it/ceri/ring.rdf>>

WHERE {

?o1 o:haPersonaggio ?p .

?o2 o:haPersonaggio ?p .

?o1 != ?o2 .

?o1 o:haTitolo ?t1 .

?o2 o:haTitolo ?t2 .

}

?t1	?t2	?p
'Oro del Reno'	'Valchiria'	'Wotan'
'Valchiria'	'Oro del Reno'	'Wotan'
'Valchiria'	'Sigfrido'	'Brunilde'
'Sigfrido'	'Valchiria'	'Brunilde'
...

Query SPARQL: 4

Estrarre i titoli delle opere e i nomi dei personaggi che hanno gli stessi personaggi che compaiono in Valchiria e filtrare le opere la cui durata è superiore a 300 minuti.

PREFIX o: <http://www.polimi.it/ceri/opera>

SELECT ?t, ?p

FROM <http://www.polimi.it/ceri/ring.rdf>

WHERE {

?o1 o:haPersonaggio ?p .

?o2 o:haPersonaggio ?p .

?o1 != ?o2 .

?o1 o:haTitolo ?t .

?o2 o:haTitolo 'Valchiria' .

?o2 o:dura ?d .

FILTER (?d > 300).

}

?t	?p
'Sigfrido'	'Brunilde'
'Crepuscolo degli Dei'	'Brunilde'

Query SPARQL: 5

Estrarre le opere che hanno Wotan come personaggio oppure durano più di 350 minuti.

PREFIX o: <http://www.polimi.it/ceri/opera>

SELECT ?t

FROM <http://www.polimi.it/ceri/ring.rdf>

WHERE {

{ ?o o:haTitolo ?t .

?o o:haPersonaggio ?p .

FILTER (?p = 'Wotan') .}

UNION

{ ?o o:haTitolo ?t .

?o o:dura ?m .

FILTER (?m > 350). }

}

?t
'Oro del Reno'
'Valchiria'
'Crepuscolo degli Dei'

Query SPARQL: 6

Estrarre titolo, durata, numero di atti e presenza di prologo di tutte le opere.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ?t, ?d, ?a, ?p
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o o:haTitolo ?t .
    ?o o:haDurata ?d .
    OPTIONAL {
        ?o o:haAtti ?a .
        ?o o:haPrologo ?p. }
}
```

?t	?d	?a	?p
'Oro del Reno'	150	NULL	NULL
'Valchiria'	310	3	NULL
'Sigfrido'	320	3	NULL
'Crepuscolo degli Dei'	360	3	'si'

Query SPARQL: 7

Estrarre il titolo di tutte le opere che hanno qualche personaggio diverso da Sigfrido.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ?t
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o o:haTitolo ?t
    ?o o:haPersonaggio ?p
    FILTER {
        ?p != 'Sigfrido'}
}
```

?t
'Oro del Reno'
'Valchiria'
'Sigfrido'
'Crepuscolo degli Dei'

Query SPARQL: 8

Estrarre le opere in cui non è presente il personaggio Sigfrido.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
```

```
SELECT ?t
```

```
FROM <http://www.polimi.it/ceri/ring.rdf>
```

```
WHERE {
```

```
    ?o o:haTitolo ?t
```

```
    OPTIONAL {
```

```
        ?o haPersonaggio ?p .
```

```
        FILTER (?p = 'Sigfrido')}
```

```
        FILTER (! BOUND(?p))
```

```
}
```

?t
'Oro del Reno'
'Valchiria'

Query SPARQL: Construct

Costruire nuove tuple RDF a partire dall'istanza RDF e costruendo legami per le variabili ?t e ?p, in particolare le coppie il cui primo elemento è il nome Ring e il secondo elemento è un personaggio di una delle quattro opere che lo compongono.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
```

```
CONSTRUCT { ?t haPersonaggio ?p . }
```

```
FROM <http://www.polimi.it/ceri/ring.rdf>
```

```
WHERE {
```

```
    ?r o:haTitolo ?t .
```

```
    ?r o:comprende ?o .
```

```
    ?o o:haPersonaggio ?p .
```

```
}
```

'Ring' haPersonaggio 'Wotan'.

'Ring' haPersonaggio 'Brunilde'.

'Ring' haPersonaggio 'Wotan'.

'Ring' haPersonaggio 'Sieglinde'.

'Ring' haPersonaggio 'Brunilde'.

'Ring' haPersonaggio 'Sigfrido'.

'Ring' haPersonaggio 'Brunilde'.

'Ring' haPersonaggio 'Sigfrido'.

'Ring' haPersonaggio 'Waltraute'.

Query SPARQL: Ask

Valuta se una query ha un risultato non nullo, cioè se alle variabili della query viene associata almeno una tupla di binding.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
```

```
ASK
```

```
FROM <http://www.polimi.it/ceri/ring.rdf>
```

La query ASK ha valore **false**.

```
WHERE {
```

```
    ?o o:haTitolo 'Walkiria' .
```

```
    ?o o:haPrologo ?p .
```

```
}
```

```
PREFIX o: <http://www.polimi.it/ceri/opera>
```

```
ASK
```

```
FROM <http://www.polimi.it/ceri/ring.rdf>
```

La query ASK ha valore **true**.

```
WHERE {
```

```
    ?o o:haTitolo 'Crepuscolo degli Dei' .
```

```
    ?o o:haPrologo ?p .
```

```
}
```

Query SPARQL: Describe

Estrae tutta l'informazione conosciuta relativamente alle risorse che soddisfano una query, in una forma definita dall'endpoint SPARQL.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
```

```
DESCRIBE ?o
```

```
FROM <http://www.polimi.it/ceri/ring.rdf>
```

```
WHERE {
```

```
    ?o o:haTitolo 'Valchiria' .
```

```
}
```

SPARQL 1.1

Essa introduce le clausole GROUP BY e HAVING. Inoltre, introduce l'operatore binario MINUS, che completa la copertura dell'algebra relazionale. Introduce la clausola NOT EXISTS, che ricorda le sottoquery di SQL. E come in SQL, sono disponibili le funzioni aggregate COUNT, SUM, MAX, MIN e AVG, cui si aggiunge la SAMPLE (per estrarre un valore arbitrario da un insieme di valori).

Aggregazione in SPARQL 1.1

Calcolare la durata totale delle quattro opere che formano il Ring.

PREFIX o: <http://www.polimi.it/ceri/opera>

SELECT ((SUM ?d) AS ?DurataDelRing)

FROM <http://www.polimi.it/ceri/ring.rdf>

WHERE {

 ?r o:haTitolo 'Ring' .

 ?r o:comprende ?o .

 ?o o:dura ?d .

}

GROUP BY ?r

?DurataDelRing

1140

Negazione in SPARQL 1.1

Estrarre le opere in cui non è presente il personaggio Sigfrido. Due modi alternativi:

PREFIX o: <http://www.polimi.it/ceri/opera>

SELECT ?t

FROM <http://www.polimi.it/ceri/ring.rdf>

WHERE {

 ?o o:haPersonaggio ?p .

 ?o o:haTitolo ?t .}

MINUS {

 ?o haPersonaggio 'Sigfrido' .

 ?o haTitolo ?t . }

PREFIX o: <http://www.polimi.it/ceri/opera>

SELECT ?t

FROM <http://www.polimi.it/ceri/ring.rdf>

WHERE {

 ?o o:haTitolo ?t .

 FILTER (NOT EXISTS ?o haPersonaggio 'Sigfrido'))

?t

'Oro del Reno'

'Valchiria'

Sottoquery in SPARQL 1.1

Calcolare quanti sono i personaggi che compaiono più di due volte. La soluzione calcola il risultato nidificando due query aggregate.

PREFIX o: <http://www.polimi.it/ceri/opera>

SELECT ((sum ?p) AS ?n)

FROM <http://www.polimi.it/ceri/ring.rdf>

WHERE

```
{ SELECT ?p  
  WHERE { ?r o:haTitolo 'Ring' .  
          ?r o:comprende ?o .  
          ?o o:haPersonaggio ?p . }  
  GROUP BY ?p  
  HAVING ( COUNT(?o) > 2 ) }
```

?n

3

SPARQL e interoperabilità

Tramite istruzioni PREFIX è possibile importare definizioni di standard, vocabolari e ontologie:

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

PREFIX dbcat: <http://dbpedia.org/resource/Category:>

PREFIX dbpprop: <http://dbpedia.org/property/>

PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>

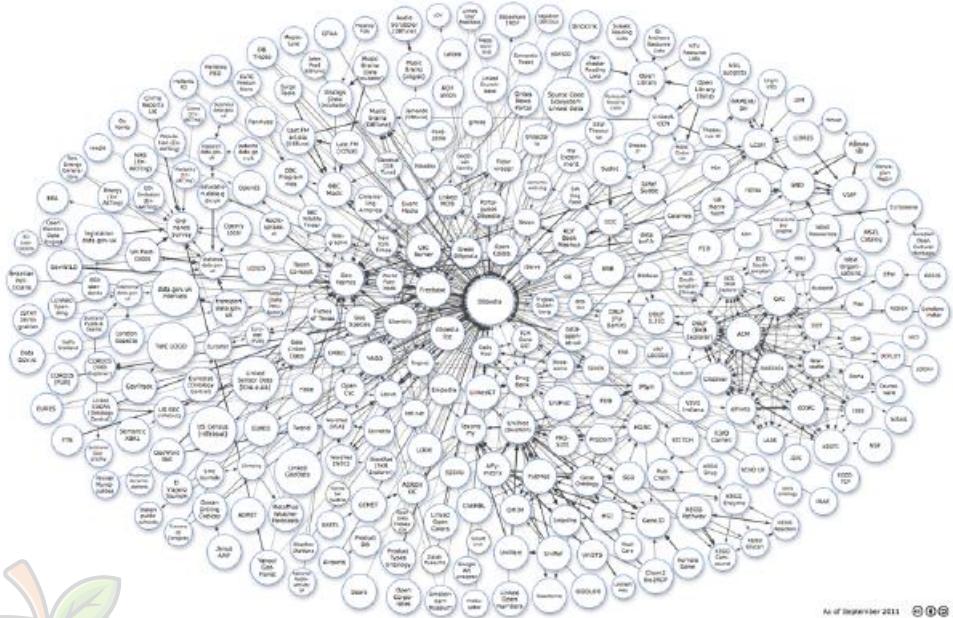
PREFIX yago: <http://dbpedia.org/class/yago/>

Inoltre, è possibile importare i dati da istanze RDF, tramite la clausola FROM NAMED. La clausola GRAPH consente di valutare la query su una particolare istanza RDF, oppure di determinare le istanze RDF che producono ciascun binding.

```
SELECT ?g, ?n  
FROM NAMED <http://www.deib.polimi.it/reports>  
FROM NAMED <http://www.dia.uniroma3.it/reports>  
FROM NAMED <http://www.ingegneria.unibg.it/reports >  
WHERE {  
  GRAPH ?g  
  { ?r scrittoDa ?p .  
    ?p haNome ?n .  
    ?r haTitolo 'Genomic Data Management' . } }
```

Linked e Open Data

I Linked Data sono risorse disponibili sul Web, descritte tramite triple RDF e collegate fra loro tramite riferimenti (link). Pubblicare risorse sotto forma di Linked Data vuol dire aderire ad una buona pratica di pubblicazione, usando le IRI e per identificare le risorse, lo standard http per trasferire dati tra diversi nodi del Web, e gli standard RDF, RDFS e OWL per descrivere le triple.



- Risorse interdominio: DBpedia (un dataset periodicamente estratto in modo automatico dai nodi di Wikipedia), Freebase, Yago e OpenCyc.
- Dati di tipo geografico: Geonames (vari milioni di nomi di luoghi) e LinkedGeoData (derivato dal progetto OpenStreetMap).
- Dati relativi ai media: BBC, New York Times e la Reuters.
- Dati relativi all’istruzione e alla ricerca: American Library of Congress, Open Library, DBLP.
- Dati relativi alle scienze della vita: Gene Ontology (che descrive i geni), UniProt (che descrive le proteine), Kegg (l’enciclopedia di Kyoto dei geni e dei genomi) e PubMed (che descrive le pubblicazioni scientifiche relative alla medicina).
- Dati relativi al commercio: GoodRelations (che descrive i vari aspetti del commercio elettronico).

Open Data

Sono dati amministrativi resi di dominio pubblico per aumentare la trasparenza delle amministrazioni nei riguardi dei cittadini. Tra i primi a pubblicare open data, gli Stati Uniti (www.data.gov) nel 2009, sotto la spunta dell’amministrazione Obama, e la Gran Bretagna (www.data.gov.uk).

Il fenomeno della pubblicazione dei dati aperti, talvolta non connessi alla rete dei linked data, è in continua evoluzione, e si citano ad esempio i siti dei comuni di Milano (www.datil.comune.milano.it) e di Roma (www.datil.comune.roma.it).

Data Warehouse

La maggior parte delle aziende dispone di enormi basi di dati contenenti dati di tipo operativo per questo c'è bisogno di **sistemi per il supporto alle decisioni** che permettano di: analizzare lo stato dell'azienda e prendere decisioni rapide e migliori. Gli **ambiti applicativi** di questi sistemi sono vari: industrie manifatturiere, servizi finanziari, telecomunicazioni, sanità.

Business intelligence

È una disciplina di supporto alla decisione strategica aziendale il cui obiettivo è quello di trasformare i dati aziendali in informazioni fruibili a diversi livelli di dettaglio e per applicazioni di analisi. Necessita un'adeguata infrastruttura hw e sw di supporto.

Knowledge discovery

È l'attività che consiste nell'individuare ed estrarre da enormi volumi di dati tutta la conoscenza utile alle attività più strategiche.

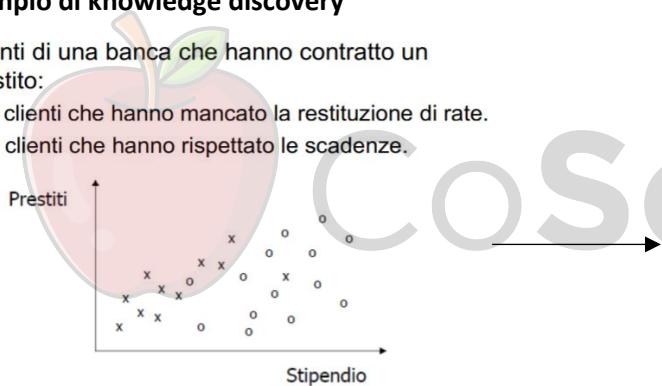
Dati: insieme di informazioni contenute in una base di dati o data warehouse

Pattern: espressione in un linguaggio opportuno che descrive in modo succinto le informazioni estratte dai dati: regolarità e informazione di alto livello.

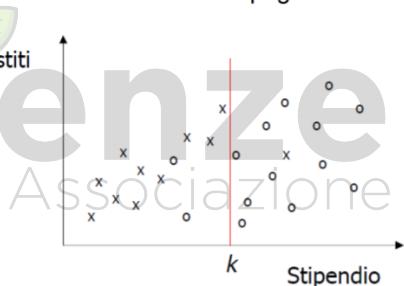
Esempio di knowledge discovery

Clienti di una banca che hanno contratto un prestito:

- x: clienti che hanno mancato la restituzione di rate.
- o: clienti che hanno rispettato le scadenze.



If stipendio < k € then mancati pagamenti.



Caratteristiche dei pattern

- **Validità:** i pattern scoperti devono essere validi su nuovi dati con un certo grado di certezza. Ad esempio, lo spostamento a destra del valore di k porta a una riduzione del grado di certezza
- **Novità:** misurata rispetto a variazione dei dati o della conoscenza estratta
- **Utilità:** ad esempio un aumento di profitto atteso dalla banca associato alla regola estratta
- **Comprensibilità:** misure di tipo: sintattico (numero di bit del pattern), semantico

Elaborazione dei dati (OLTP)

La modalità tradizionale di uso dei DBMS è caratterizzata da:

- Instantanea del valore corrente dei dati
- Dati dettagliati, rappresentazione relazionale
- Operazioni strutturate e ripetitive
- Accesso in lettura o aggiornamento di pochi record
- Transazioni brevi
- Isolamento, affidabilità e integrità sono critici
- Dimensione della base di dati >>100Mb - Gb

Analisi dei dati (OLAP)

E' la principale modalità di fruizione delle informazioni contenute in un DW. Consente agli utenti di esplorare interattivamente i dati sulla base del modello multidimensionale. L'elaborazione dei dati per il supporto alle decisioni è caratterizzata da:

- Dati di tipo storico;
- Dati consolidati e integrati
- Applicazioni ad hoc
- Accesso in lettura a milioni di record
- Interrogazioni di tipo complesso
- Consistenza dei dati prima e dopo le operazioni di caricamento periodico
- Dimensione della base di dati >> 100Gb – Tb

Data Warehousing

È una collezione di metodi, tecnologie e strumenti di ausilio al “knowledge worker” per condurre analisi dei dati finalizzate all'attuazione di processi decisionali e al miglioramento del patrimonio informativo. Sono quindi **sistemi di data warehousing**.

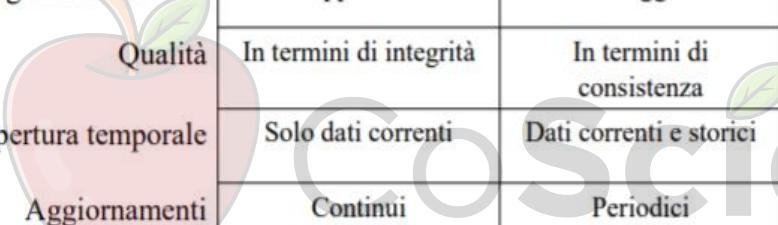
Data Warehouse

E' una base di dati per il supporto alle decisioni, che è mantenuta separatamente dalle basi di dati operative dell'azienda. E presenta le seguenti **caratteristiche**:

- Orientata ai soggetti di interesse: clienti, prodotti, vendite
- Integrata e consistente: si appoggia su più fonti eterogenee di dati
- Rappresentativa dell'evoluzione temporale e non volatile: aggiornato ad intervalli regolari

Differenza tra DB operazionali e Data Warehouse

	DB operazionali	DW
Utenti	Migliaia	Centinaia
Carico di lavoro	Transazioni predefinite	Interrogazioni di analisi ad hoc
Accesso	Centinaia di record in lettura e scrittura	Milioni di record per lo più in lettura
Scopo	Dipende dall'applicazione	Supporto alle decisioni
Dati	Elementari	Di sintesi



Integrazione dei dati
Qualità	Per applicazione	Per soggetto
Copertura temporale	In termini di integrità	In termini di consistenza
Aggiornamenti	Solo dati correnti	Dati correnti e storici
Modello	Continui	Periodici
Ottimizzazione	Normalizzato	Denormalizzato, Multidimensionale
Sviluppo	Per accessi OLTP su una frazione del DB	Per accessi OLAP su gran parte del DB
	A cascata	Iterativo

Data Warehouse e Data mart

Data Warehouse aziendale: contiene informazioni sul funzionamento di **tutta** l'azienda.

Data mart: sottoinsieme dipartimentale focalizzato su un settore prefissato. Ha una realizzazione più rapida ma richiede una progettazione attenta per evitare problemi di integrazione in seguito.

Caratteristiche architetturali Datawarehouse

- Separazione: dell'elaborazione analitica da quella transazionale
- Scalabilità: capacità di ridimensionamento a fronte della crescita del volume dei dati
- Estendibilità: possibilità di integrare nuove applicazioni senza riprogettare il sistema
- Sicurezza: controllo sugli accessi
- Amministrabilità: semplicità nell'amministrazione dei dati

Architettura DW ad un livello



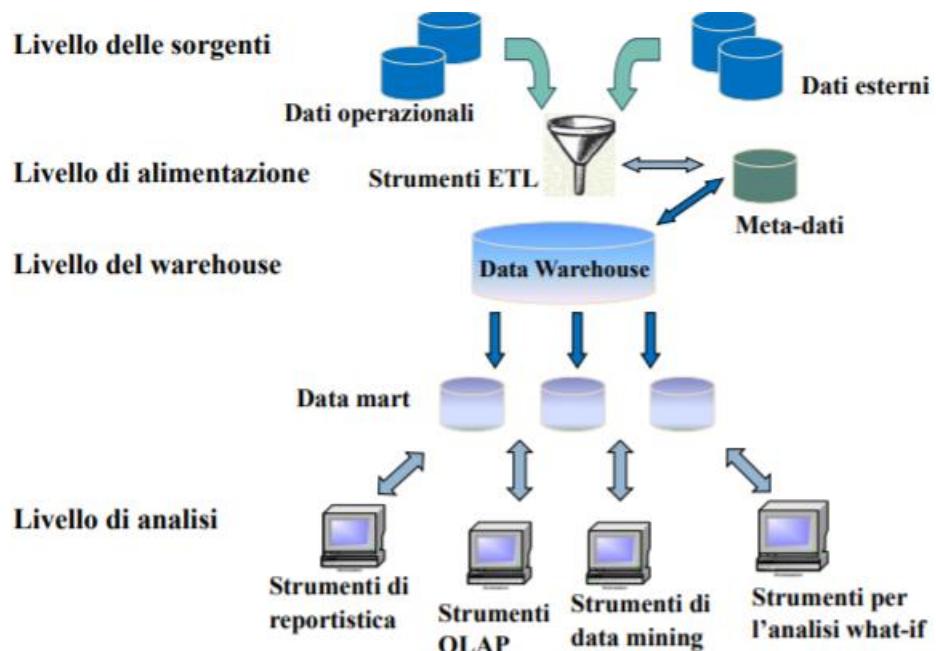
Caratteristiche:

- DW virtuale
- Minimizzazione dei dati memorizzati

Punti deboli:

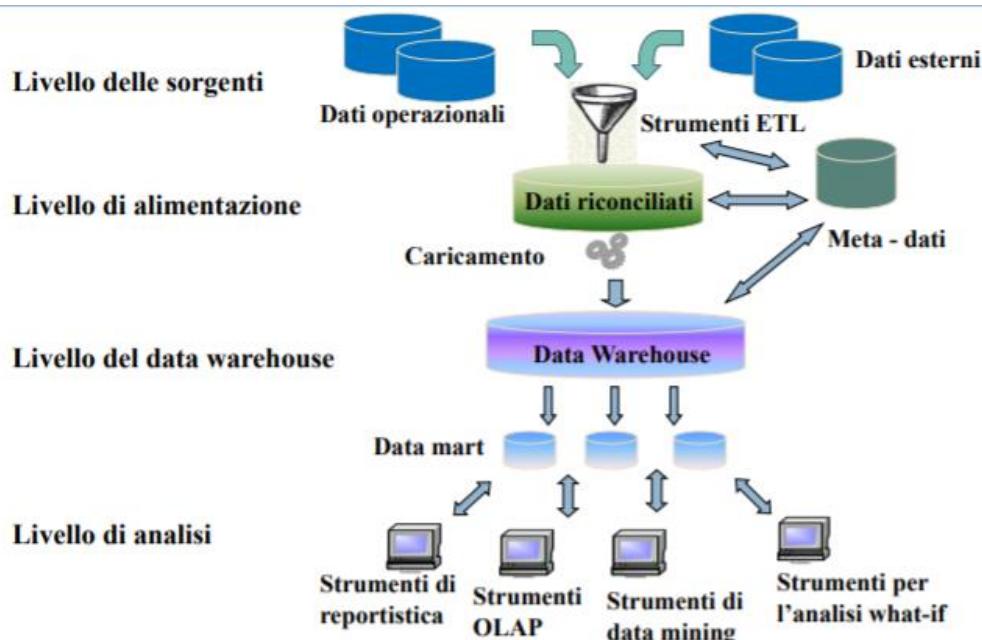
- Non rispetta il requisito di separazione tra l'elaborazione analitica OLAP e quella transazionale OLTP.

Architettura a due livelli



- **Livello delle sorgenti:** Fonti di dati eterogenei estratti dall'ambiente di produzione oppure provenienti da sistemi informativi esterni all'azienda.
- **Livello dell'alimentazione:** i dati memorizzati nelle sorgenti vengono estratti e ripuliti tramite strumenti **ETL** (Extraction, Transformation and Loading)
- **Livello del warehouse:** le informazioni vengono raccolte in un DW centralizzato e può essere consultato direttamente o utilizzato come sorgente per costruire i Data mart.
- **Livello di analisi:** Permette la consultazione efficiente e flessibile dei dati integrati per fini di stesura di report, di analisi e di simulazione

Architettura a tre livelli



Viene introdotto il livello dei dati riconciliati: ovvero il livello che materializza i dati operazionali ottenuti a valle del processo di integrazione e ripulitura dei dati sorgente.

Gli strumenti ETL

Il ruolo degli **strumenti ETL** è quello di alimentare una sorgente di dati che possa a sua volta alimentare il DW. Le operazioni di questi strumenti vengono definite con il termine di **riconciliazione**. Svolgono il proprio compito in 4 fasi:

1. Estrazione dei dati da sorgenti esterne
2. Pulizia dei dati (errori, dati mancanti o duplicati)
3. Trasformazioni e conversioni di formato
4. Caricamento e refresh periodico

La riconciliazione avviene: periodicamente e quando il DW viene popolato la prima volta.

ETL: estrazione

Consiste nell'estrazione di dati rilevanti dalle sorgenti

Tipi di estrazione:

- Estrazione statica: effettuata quando il DW viene popolato per la prima volta
- Estrazione incrementale: viene usata per l'aggiornamento periodico nel DW
- Guidata dalle sorgenti: consiste nel riscrivere le applicazioni operazionali per far sì che esse notifichino in modo asincrono le modifiche.

ETL: pulitura

Si occupa di migliorare la qualità dei dati, normalmente scarsa nelle sorgenti. Le sue **funzionalità** sono: Utilizza dizionari appositi per correggere gli errori di scrittura (**Correzione ed omogeneizzazione**) e applica regole del dominio applicativo per stabilire le corrette corrispondenze tra valori (**pulitura basata su regole**).

Alcune tipologie di errori: dati duplicati, mancanti, inconsistenza, valori impossibili

ETL: trasformazione

Converte i dati dal formato operazionale sorgente a quello del DW. Le sue **funzionalità** sono: operare sia a livello di formato di memorizzazione sia a livello di unità di misura al fine di uniformare i dati (**Conversione e normalizzazione**), stabilire corrispondenze tra campi equivalenti in sorgenti diverse (**Matching**) e ridurre il numero di campi e di record rispetto alle sorgenti (**Selezione**).

ETL: caricamento

Il caricamento dei dati avviene secondo due modalità:

- **Refresh**: I dati nel DW vengono riscritti integralmente, usata in abbinamento all'estrazione statica
- **Update**: Nel DW vengono aggiunti solo i cambiamenti occorsi ai dati sorgente, usata in abbinamento all'estrazione incrementare.

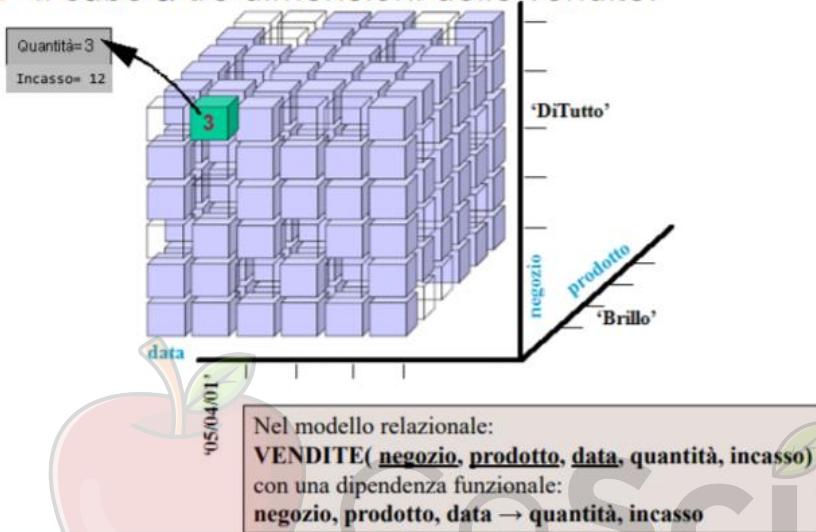
Struttura ed elaborazione dei dati

Il modello multidimensionale

È il donamento per la rappresentazione e l'interrogazione dei dati nei DW. Gli oggetti che influenzano il processo decisionale sono **fatti** di un'organizzazione (vendite, spedizioni, ricoveri...). I fatti di interesse sono rappresentati in **cubi** in cui:

- ogni cella contiene **misure** numeriche che quantificano il fatto da diversi punti di vista (**evento**)
- ogni asse rappresenta una **dimensione** di interesse per l'analisi
- ogni dimensione può essere la radice di una **gerarchia** di attributi usati per aggregare i dati memorizzati nei cubi di base.

□ Il cubo a tre dimensioni delle vendite:



A ciascuna dimensione del cubo è associata una gerarchia di livelli di aggregazione:

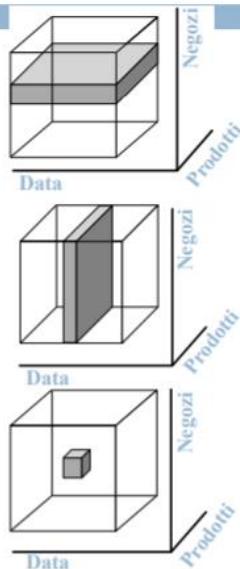
- prodotto -> tipo -> categoria
- negozio -> città -> regione
- data -> mese -> anno

Restrizione

Restringere i dati significa ritagliare una porzione dal cubo circoscrivendo il campo di analisi. La forma più semplice di restrizione è lo **slicing**. Consiste nel ridurre la dimensionalità del cubo fissando un valore per uno o più dimensioni.

Esempi di restrizione:

- ❖ Al manager regionale interessa la vendita dei prodotti in tutti i periodi nei **propri negozi**:
- ❖ Al manager finanziario interessa la vendita dei prodotti in tutti i negozi relativamente ad **un determinato periodo**:
- ❖ Il manager strategico si concentra su una categoria di prodotti, un'area regionale ed un orizzonte temporale medio:



Eventi e aggregazione

Un **evento primario** è una particolare occorrenza di un fatto, individuata da una ennupla caratterizzata da un valore per ciascuna dimensione. Ad esempio, un evento primario potrebbe essere il “05/04/01” nel negozio “DiTutto” è stata venduta una quantità 3 con incassi 12 del prodotto “Brillo”.

Un **evento secondario** è, dato un insieme di attributi dimensionali (pattern), ciascuna ennupla di questi valori che aggrega tutti gli eventi primari corrispondenti.

Le **gerarchie** definiscono il modo in cui gli eventi primari possono essere aggregati e selezionati significativamente per il processo decisionale. La dimensione in cui una gerarchia ha radice ne definisce la **granularità** più fine di aggregazione. Agli attributi dimensionali corrispondono granularità via via crescenti.

L'aggregazione richiede di definire un operatore adatto per comporre i valori delle misure che caratterizzano gli eventi primari in valori da abbinare a ciascun evento elementare.

Metadati

I metadati sono dati usati per descrivere altri dati. Indicano le sorgenti, descrivono la struttura dei dati nel DW, indicano il valore, l'uso e la funzione dei dati memorizzati, descrivono come i dati sono alterati e trasformati.

Il contenitore dei metadati è strettamente collegato al DW. Esistono due tipi di metadati:

- **Interni:** descrivono sorgenti, trasformazioni e politiche di alimentazione, sono di interesse per l'amministratore
- **Esterne:** descrivono le definizioni, quantità, unità di misura, aggregazioni significativa, sono di interesse per gli utenti

Reportistica

È una modalità di accesso ai dati del DW (l'altra modalità è OLAP) orientata ad utenti che devono accedere ad informazioni strutturate in modo pressoché invariabile nel tempo. Il progettista può formulare l'interrogazione e renderla disponibile nel tempo. Un report è definito da una interrogazione (es: selezione ed aggregazione) e da una presentazione (tabellare/grafica).

Operatori OLAP di restrizione (Slicing)

E' il processo per cui si fissa un valore per almeno uno degli attributi dimensionali e si escludono dall'analisi tutti quegli eventi che non presentano tale valore. Come risultato si ottiene un cubo con un numero di dimensioni inferiore almeno di uno rispetto al cubo sorgente.

The diagram shows two cubes. The top cube represents the original source cube with dimensions: Category, Year, Metrics, Customer, Region, and Dollar Sales. The bottom cube represents the result of slicing the top cube by the predicate Year = '1998', resulting in a smaller cube with dimensions: Category, Metrics, Customer, Region, and Dollar Sales. A red arrow points from the top cube to the bottom cube.

Category	Year	Metrics	Dollar Sales									
			Customer	Region	North-East	Mid-Atlantic	South-East	Central	South	North-West	South-West	England
Electronics	1997		\$ 138	\$ 1.774	\$ 384	\$ 138	\$ 2.346	\$ 2.554	\$ 2.184	\$ 566	\$ 199	\$ 7
	1998	↳	\$ 1.184	\$ 4.529	\$ 1.892	\$ 7.232	\$ 651	\$ 9.488	\$ 476	\$ 2.683	\$ 462	\$ 1
Food	1997		\$ 759	\$ 682	\$ 729	\$ 262	\$ 586	\$ 469	\$ 807	\$ 156	\$ 615	\$ 1
	1998	↳	\$ 538	\$ 925	\$ 959	\$ 677	\$ 213	\$ 1.503	\$ 261	\$ 165	\$ 175	\$ 1
Gifts	1997		\$ 2.532	\$ 1.355	\$ 1.854	\$ 1.413	\$ 2.535	\$ 2.132	\$ 1.904	\$ 908	\$ 375	\$ 10
	1998	↳	\$ 1.955	\$ 2.785	\$ 2.800	\$ 2.695	\$ 1.813	\$ 2.844	\$ 1.778	\$ 1.158	\$ 717	\$ 5
Health & Beauty	1997		\$ 624	\$ 640	\$ 1.317	\$ 647	\$ 588	\$ 754	\$ 654	\$ 143	\$ 292	\$ 3
	1998	↳	\$ 611	\$ 887	\$ 566	\$ 382	\$ 499	\$ 1.162	\$ 1.044	\$ 273	\$ 72	
Household	1997		\$ 5.354	\$ 4.112	\$ 5.410	\$ 4.446	\$ 3.058	\$ 3.974	\$ 2.654	\$ 3.545	\$ 2.875	\$ 1.9
	1998	↳	\$ 5.787	\$ 5.320	\$ 5.416	\$ 6.812	\$ 4.334	\$ 5.008	\$ 7.588	\$ 2.139	\$ 3.649	\$ 2.7
Kid's Korner	1997		\$ 201	\$ 398	\$ 485	\$ 186	\$ 409	\$ 323	\$ 396	\$ 105	\$ 34	\$
	1998	↳	\$ 247	\$ 422	\$ 441	\$ 380	\$ 221	\$ 592	\$ 290	\$ 198	\$ 19	\$
Travel	1997		\$ 624	\$ 505	\$ 564	\$ 386	\$ 300	\$ 978	\$ 416	\$ 48	\$ 38	
	1998	↳	\$ 608	\$ 559	\$ 1.096	\$ 611	\$ 464	\$ 316	\$ 573	\$ 257	\$ 198	\$

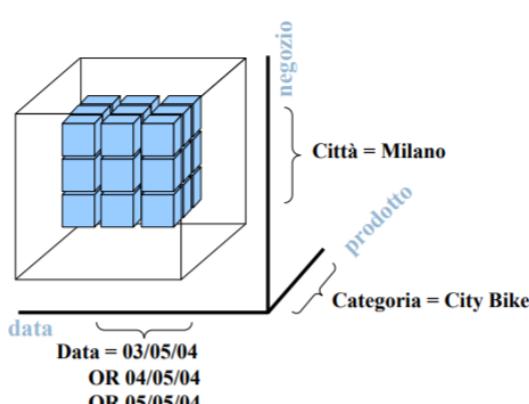
Category	Year	Metrics	Dollar Sales									
			Customer	Region	North-East	Mid-Atlantic	South-East	Central	South	North-West	South-West	England
Electronics	1998		\$ 1.184	\$ 4.529	\$ 1.892	\$ 7.232	\$ 651	\$ 9.488	\$ 476	\$ 2.683	\$ 462	\$ 702
Food	1998		\$ 538	\$ 925	\$ 959	\$ 677	\$ 213	\$ 1.503	\$ 251	\$ 155	\$ 175	\$ 100
Gifts	1998		\$ 1.955	\$ 2.785	\$ 2.800	\$ 2.695	\$ 1.813	\$ 2.844	\$ 1.778	\$ 1.158	\$ 717	\$ 66
Health & Beauty	1998		\$ 611	\$ 887	\$ 566	\$ 382	\$ 499	\$ 1.162	\$ 1.044	\$ 273	\$ 72	
Household	1998		\$ 5.787	\$ 5.320	\$ 5.416	\$ 6.812	\$ 4.334	\$ 5.008	\$ 7.588	\$ 2.139	\$ 3.649	\$ 2.7
Kid's Korner	1998		\$ 247	\$ 422	\$ 441	\$ 380	\$ 221	\$ 592	\$ 290	\$ 198	\$ 19	\$ 5
Travel	1998		\$ 608	\$ 559	\$ 1.096	\$ 611	\$ 464	\$ 316	\$ 573	\$ 257	\$ 198	\$ 5

Slicing sul predicato Year = '1998'.

Operatori OLAP di restrizione (Dicing)

Consiste nello stabilire per almeno una delle dimensioni di analisi un sottoinsieme di valori possibili per tale attributo e di escludere quei fatti che non sono associati a nessuno di tali valori.

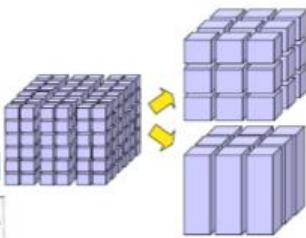
Città = "Milano"
Categoria = "City Bike"
Data = "03/05/04 OR
04/05/04 OR
05/05/04"



Operatori OLAP di aggregazione (Rollup)

Consiste nel passare da una visualizzazione ad un livello di dettaglio più fine ad una visualizzazione ad un livello di dettaglio meno accurato.

Roll-Up



Metrics	Dollar Sales	North-East	Mid-Atlantic	South-East	Central	South	North-West	South-West	England	France	Germany	Canada
Customer Region												
Month												
Jan 97	\$ 620	\$ 753	\$ 30	\$ 660	\$ 2,493	\$ 1,312	\$ 440	\$ 1,020	\$ 1,020	\$ 380	\$ 315	
Feb 97	\$ 258	\$ 252	\$ 800	\$ 975	\$ 160	\$ 582	\$ 744	\$ 310	\$ 799	\$ 118	\$ 357	
Mar 97	\$ 649	\$ 244	\$ 148	\$ 250	\$ 1,085	\$ 7,911	\$ 630	\$ 1,240	\$ 119	\$ 142	\$ 95	
Apr 97	\$ 767	\$ 588	\$ 100	\$ 180	\$ 226	\$ 506	\$ 601	\$ 1197	\$ 550	\$ 85		
May 97	\$ 1,250	\$ 45	\$ 936	\$ 156	\$ 664	\$ 810	\$ 101	\$ 135	\$ 200	\$ 177	\$ 235	
Jun 97	\$ 842	\$ 582	\$ 1,281	\$ 147	\$ 140	\$ 176	\$ 1,130	\$ 592	\$ 254	\$ 745		
Jul 97	\$ 3,935	\$ 260	\$ 486	\$ 1,393	\$ 2,965	\$ 2,965	\$ 1,050	\$ 1,537	\$ 175	\$ 65		
Aug 97	\$ 1,783	\$ 304	\$ 1,032	\$ 170	\$ 299	\$ 376	\$ 432	\$ 190	\$ 241	\$ 177	\$ 259	
Sep 97	\$ 581	\$ 778	\$ 3,598	\$ 287	\$ 440	\$ 1,652	\$ 1,071	\$ 313	\$ 210	\$ 252		
Oct 97	\$ 2,291	\$ 1,040	\$ 600	\$ 558	\$ 1,385	\$ 708	\$ 1,218	\$ 437	\$ 220	\$ 520	\$ 65	
Nov 97	\$ 39	\$ 1,602	\$ 1,082	\$ 1,187	\$ 842	\$ 759	\$ 740	\$ 232	\$ 101	\$ 1,037	\$ 37	
Dec 97	\$ 381	\$ 1,598	\$ 349	\$ 118	\$ 1,459	\$ 635	\$ 2,021	\$ 259	\$ 210	\$ 119	\$ 189	
Jan 98	\$ 311	\$ 1,174	\$ 2,634	\$ 3,130	\$ 954	\$ 2,083	\$ 1,351	\$ 747	\$ 405	\$ 447	\$ 1,141	
Feb 98	\$ 2,518	\$ 702	\$ 1,123	\$ 1,336	\$ 1,227	\$ 3,887	\$ 543	\$ 268	\$ 277	\$ 282		
Mar 98	\$ 2,459	\$ 1,523	\$ 1,178	\$ 4,708	\$ 1,480	\$ 3,514	\$ 1,948	\$ 1,705	\$ 276	\$ 1,158	\$ 83	
Apr 98	\$ 407	\$ 841	\$ 524	\$ 712	\$ 133	\$ 2,486	\$ 49	\$ 390	\$ 1,299	\$ 221	\$ 46	
May 98	\$ 667	\$ 1,721	\$ 440	\$ 148	\$ 80	\$ 1,110	\$ 303	\$ 104	\$ 657	\$ 65		
Jun 98	\$ 699	\$ 1,096	\$ 898	\$ 353	\$ 903	\$ 839	\$ 230	\$ 155	\$ 105	\$ 75		
Jul 98	\$ 586	\$ 1,897	\$ 412	\$ 225	\$ 400	\$ 361	\$ 1,628	\$ 267	\$ 1,011	\$ 41	\$ 184	
Aug 98	\$ 894	\$ 326	\$ 792	\$ 1,832	\$ 1,199	\$ 295	\$ 1,816	\$ 277	\$ 102	\$ 118	\$ 115	
Sep 98	\$ 338	\$ 3,179	\$ 505	\$ 427	\$ 99	\$ 2,976	\$ 885	\$ 125	\$ 85	\$ 1,110	\$ 10	
Oct 98	\$ 544	\$ 413	\$ 1,467	\$ 209	\$ 879	\$ 706	\$ 556	\$ 480	\$ 485	\$ 99	\$ 160	
Nov 98	\$ 871	\$ 469	\$ 1,471	\$ 2,066	\$ 701	\$ 716	\$ 986	\$ 1,327	\$ 154	\$ 440	\$ 381	
Dec 98	\$ 83n	\$ 2,096	\$ 1,720	\$ 3,042	\$ 395	\$ 1,740	\$ 1,943	\$ 1,143	\$ 380	\$ 207	\$ 118	

Roll-Up sulla gerarchia temporale.

Metrics	Dollar Sales	North-East	Mid-Atlantic	South-East	Central	South	North-West	South-West	England	France	Germany	Canada
Customer Region												
Quarter												
Q1 1997	\$ 1,526	\$ 1,249	\$ 978	\$ 1,885	\$ 3,650	\$ 4,855	\$ 1,834	\$ 2,552	\$ 1,920	\$ 543	\$ 553	
Q2 1997	\$ 2,979	\$ 1,415	\$ 2,664	\$ 1,582	\$ 1,130	\$ 1,906	\$ 884	\$ 1,393	\$ 1,402	\$ 515	\$ 975	
Q3 1997	\$ 3,016	\$ 1,772	\$ 5,076	\$ 2,050	\$ 1,443	\$ 2,311	\$ 2,321	\$ 508	\$ 575	\$ 782	\$ 325	
Q4 1997	\$ 2,711	\$ 5,030	\$ 2,025	\$ 1,961	\$ 3,601	\$ 2,112	\$ 3,976	\$ 918	\$ 531	\$ 1,575	\$ 291	
Q1 1998	\$ 5,288	\$ 3,399	\$ 4,935	\$ 9,174	\$ 3,601	\$ 9,484	\$ 3,844	\$ 2,720	\$ 979	\$ 1,897	\$ 1,224	
Q2 1998	\$ 1,773	\$ 3,658	\$ 1,862	\$ 1,213	\$ 1,115	\$ 4,635	\$ 352	\$ 724	\$ 2,110	\$ 391	\$ 121	
Q3 1998	\$ 1,818	\$ 5,402	\$ 1,709	\$ 2,485	\$ 1,704	\$ 3,632	\$ 4,329	\$ 679	\$ 1,198	\$ 1,269	\$ 809	
Q4 1998	\$ 2,051	\$ 2,968	\$ 4,664	\$ 5,917	\$ 1,775	\$ 3,162	\$ 3,485	\$ 2,750	\$ 1,005	\$ 846	\$ 539	

Operatori OLAP di aggregazione Drill-Down

Consiste nel passare da una visualizzazione ad un livello di dettaglio più grosso ad una visualizzazione ad un livello di dettaglio più accurato. Il dettaglio può essere aumentato o aumentando di una dimensione percorrendo la gerarchia (raggruppare per città e mese → raggruppare per negozio e mese) oppure aggiungendo una intera dimensione (raggruppare per prodotto → raggruppare per prodotto e città).

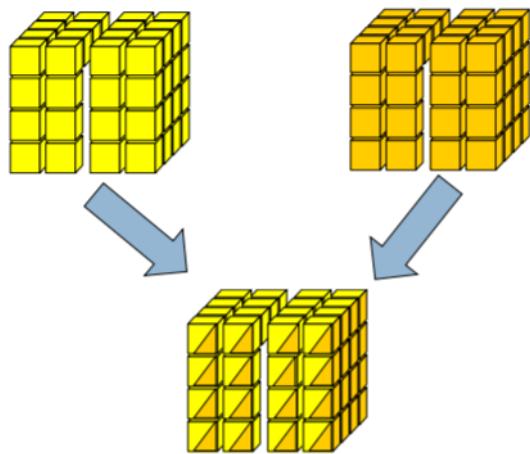
Metrics	Dollar Sales	North-East	Mid-Atlantic	South-East	Central	South	North-West	South-West	England	France	Germany	Canada
Customer Region												
Quarter												
Q1 1997	\$ 1,526	\$ 1,249	\$ 978	\$ 1,885	\$ 3,650	\$ 4,855	\$ 1,834	\$ 2,552	\$ 1,920	\$ 543	\$ 553	
Q2 1997	\$ 2,979	\$ 1,415	\$ 2,664	\$ 1,582	\$ 1,130	\$ 1,906	\$ 884	\$ 1,393	\$ 1,402	\$ 515	\$ 975	
Q3 1997	\$ 3,016	\$ 1,772	\$ 5,076	\$ 2,050	\$ 1,443	\$ 2,311	\$ 2,321	\$ 608	\$ 575	\$ 782	\$ 325	
Q4 1997	\$ 2,711	\$ 5,030	\$ 2,025	\$ 1,961	\$ 3,601	\$ 2,112	\$ 3,976	\$ 918	\$ 531	\$ 1,575	\$ 291	
Q1 1998	\$ 5,288	\$ 3,399	\$ 4,935	\$ 9,174	\$ 3,601	\$ 9,484	\$ 3,844	\$ 2,720	\$ 979	\$ 1,897	\$ 1,224	
Q2 1998	\$ 1,773	\$ 3,658	\$ 1,862	\$ 1,213	\$ 1,115	\$ 4,635	\$ 352	\$ 724	\$ 2,110	\$ 391	\$ 121	
Q3 1998	\$ 1,818	\$ 5,402	\$ 1,709	\$ 2,485	\$ 1,704	\$ 3,632	\$ 4,329	\$ 679	\$ 1,198	\$ 1,269	\$ 809	
Q4 1998	\$ 2,051	\$ 2,968	\$ 4,664	\$ 5,917	\$ 1,775	\$ 3,162	\$ 3,485	\$ 2,750	\$ 1,005	\$ 846	\$ 539	

Metrics	Dollar Sales	Arlin	San Pedro	Springfield	Chappel Hill	Scranburg	Pebble Beach	Martinsville	Maddon	Peoria	Lake Barkley	Alameda	Fingers Lake
Customer City													
Quarter													
Q1 1997	\$ 675												
Q2 1997					\$ 203								
Q3 1997					\$ 276								
Q4 1997	\$ 215	\$ 124			\$ 113	\$ 45		\$ 192	\$ 348				
Q1 1998									\$ 85	\$ 12			
Q2 1998									\$ 119	\$ 17			
Q3 1998	\$ 734												
Q4 1998													

Drill-Down sulla gerarchia del cliente.

Operatori OLAP di aggregazione Drill-Across

Il Drill-Across consiste nello stabilire un confronto tra due o più cubi correlati in modo da ottenere una visualizzazione comparata di due diverse misure e per il calcolo di misure derivate dai dati presenti sui cubi. Per fare ciò occorre che i metadati presenti nel DW siano coordinati.



Operatori OLAP di aggregazione Drill-Through

Consiste nel passaggio dai dati aggregati multi dimensionalmente del DW ai dati operazionali presenti nelle sorgenti o nel livello riconciliato.

Operatori OLAP di pivoting

L'operatore di pivoting consiste nel ruotare gli assi di visualizzazione del cubo dei fatti mantenendo invariato il livello di aggregazione ed il numero delle dimensioni: ciò incrementa la leggibilità delle stesse informazioni

Left Table (Original Data):

Category		Metrics	Dollar Sales	
			1997	1998
Electronics	Year	\$ 10.616	\$ 29.299	
Electronics	1997	\$ 10.616	\$ 29.299	
Electronics	1998	\$ 10.616	\$ 29.299	
Food	Year	\$ 5.300	\$ 5.638	
Food	1997	\$ 5.300	\$ 5.638	
Food	1998	\$ 5.300	\$ 5.638	
Gifts	Year	\$ 16.315	\$ 20.047	
Gifts	1997	\$ 16.315	\$ 20.047	
Gifts	1998	\$ 16.315	\$ 20.047	
Health & Beauty	Year	\$ 6.042	\$ 5.665	
Health & Beauty	1997	\$ 6.042	\$ 5.665	
Health & Beauty	1998	\$ 6.042	\$ 5.665	
Household	Year	\$ 38.383	\$ 50.391	
Household	1997	\$ 38.383	\$ 50.391	
Household	1998	\$ 38.383	\$ 50.391	
Kid's Korner	Year	\$ 2.559	\$ 2.943	
Kid's Korner	1997	\$ 2.559	\$ 2.943	
Kid's Korner	1998	\$ 2.559	\$ 2.943	
Travel	Year	\$ 4.497	\$ 4.792	
Travel	1997	\$ 4.497	\$ 4.792	
Travel	1998	\$ 4.497	\$ 4.792	

Right Table (Pivoted Data):

Category		Metrics	Dollar Sales	
			1997	1998
Electronics	Year	\$ 10.616	\$ 29.299	
Electronics	1997	\$ 10.616	\$ 29.299	
Electronics	1998	\$ 10.616	\$ 29.299	
Food	Year	\$ 5.300	\$ 5.638	
Food	1997	\$ 5.300	\$ 5.638	
Food	1998	\$ 5.300	\$ 5.638	
Gifts	Year	\$ 16.315	\$ 20.047	
Gifts	1997	\$ 16.315	\$ 20.047	
Gifts	1998	\$ 16.315	\$ 20.047	
Health & Beauty	Year	\$ 6.042	\$ 5.665	
Health & Beauty	1997	\$ 6.042	\$ 5.665	
Health & Beauty	1998	\$ 6.042	\$ 5.665	
Household	Year	\$ 38.383	\$ 50.391	
Household	1997	\$ 38.383	\$ 50.391	
Household	1998	\$ 38.383	\$ 50.391	
Kid's Korner	Year	\$ 2.559	\$ 2.943	
Kid's Korner	1997	\$ 2.559	\$ 2.943	
Kid's Korner	1998	\$ 2.559	\$ 2.943	
Travel	Year	\$ 4.497	\$ 4.792	
Travel	1997	\$ 4.497	\$ 4.792	
Travel	1998	\$ 4.497	\$ 4.792	

Approcci all'implementazione di un DW

Esistono due tipi di approcci:

- **ROLAP: Relational OLAP**

Viene utilizzato su DBMS relazionali, sono necessarie tipologie specifiche di schemi per traslare il modello multidimensionale su attributi, relazioni e vincoli di integrità. Presenta ridondanza e ha **basse prestazioni** dovute a costose operazioni di JOIN.

- **MOLAP: Multidimensional OLAP**

Viene utilizzato su DBMS multidimensionali, è un modello ad hoc ed accesso di tipo posizionale, le operazioni multidimensionali sono realizzabili in modo semplice senza ricorrere a JOIN questo porta ad **ottime prestazioni**.

Quando si realizza l'implementazione di un DW bisogna considerare anche:

- **Qualità:** la qualità di un processo misura la sua aderenza agli obiettivi degli utenti. I fattori che caratterizzano la qualità dei dati sono l'accuratezza, l'attualità, la completezza, la tracciabilità ecc..
- **Sicurezza:** controllo delle autorizzazioni e sulla mole di dati trasferiti dalle sorgenti
- **Evoluzione:** l'evoluzione delle informazioni nel DW nel tempo a livello dei dati e a livello di schema.

Ciclo di vita dei sistemi di Data Warehousing

Fattori di rischio

All'interno di un DW sono diversi i fattori di rischio:

- Legati alla gestione del progetto: scarsa disponibilità a condividere informazioni tra reparti
- Legati alle tecnologie: rapida evoluzione delle tecnologie, mancanza di standard
- Legati ai dati ed alla progettazione: risultati di scarso valore, causati da sorgenti instabili ed inaffidabili, specifica inaccurata dei requisiti
- Legati all'organizzazione: incapacità di coinvolgere attivamente l'utente finale nel progetto

Esistono diversi approcci per diminuire questi fattori di rischio:

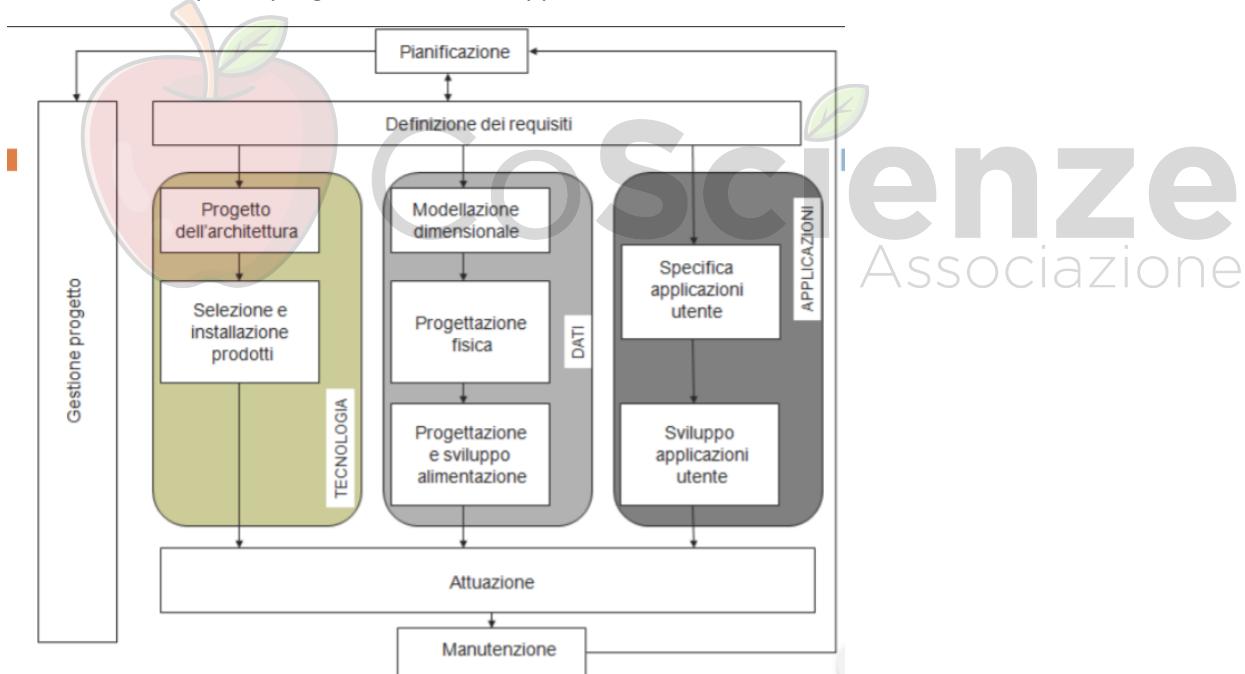
- **Approccio Top-Down:** con questo tipo di approccio è necessario analizzare i bisogni globali dell'intera azienda, pianificare lo sviluppo del DW e progettarlo e realizzarlo nella sua interezza.
 - **Vantaggi:** visione globale dell'obiettivo, DW consistente e ben integrato.
 - **Svantaggi:** Tempi lunghi di realizzazione, complessità nell'analisi e nella riconciliazione di tutte le fonti, impossibilità di prevedere le esigenze particolari delle diverse aree aziendali
- **Approccio Bottom-Up:** con questo tipo di approccio il DW è costruito in modo incrementale, i Data mart sono concentrati su una specifica area di interesse, il primo Data mart deve essere quello più strategico per l'azienda e deve ricoprire un ruolo centrale per l'intero DW. Questo approccio ha un particolare ciclo di vita:



- **Definizione degli obiettivi e pianificazione:** individuazione obiettivi e confini del sistema, stima delle dimensioni, valutazione dei costi e del valore aggiunto, scelta dell'approccio per la costruzione, analisi dei rischi e delle aspettative
- **Progettazione dell'infrastruttura:** scelte architettoniche e degli strumenti
- **Progettazione e sviluppo dei Data mart**

Business Dimensional Lifecycle (BDL)

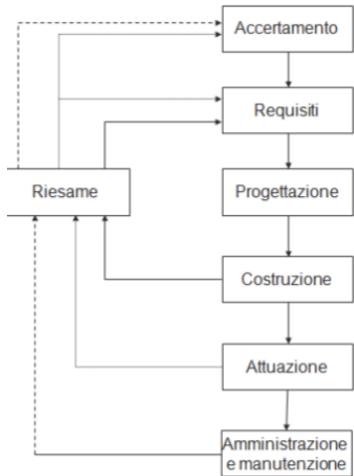
E' il ciclo di vita per la progettazione lo sviluppo e l'attuazione di DW.



- **Pianificazione:** scopi e confini del sistema, valutazione impatti organizzativi, stima costi e benefici, allocazione delle risorse
- **Definizione dei requisiti:** massima utilità e redditività del sistema, catturare i fattori chiave e trasformarli in specifiche. (diviso in fase dei dati, tecnologia e applicazioni)
- **Attuazione:** comporta l'effettivo avviamento del sistema sviluppato
- **Manutenzione:** assicura il supporto e la formazione degli utenti
- **Gestione del progetto:** occupa tutte le fasi del ciclo di vita, permette di mantenere le diverse attività sincronizzate.

Rapid Warehousing Methodology (RVM)

E' un'altra metodologia di sviluppo di DW. E' una metodologia iterativa ed evolutiva che suddivide grossi progetti in sotto progetti meno rischiosi chiamati **build**. Ogni build riprende l'ambiente di quello precedente, estendendolo con nuove funzionalità.



- **Accertamento:** fattibilità del progetto da parte dell'azienda, scopi, rischi e benefici.
- **Requisiti:** specifiche di analisi, del progetto e di architettura
- **Progettazione:** progetto logico e fisico dei dati, dell'alimentazione e selezione degli strumenti d'implementazione
- **Costruzione:** implementazione e popolazione del DW, sviluppo e collaudo front-end
- **Attuazione:** il sistema viene consegnato e avviato, gli utenti vengono addestrati
- **Amministrazione e manutenzione:** presente durante tutta la vita del sistema, estensione delle funzionalità, ridimensionamento dell'architettura
- **Riesame:** verifica dell'implementazione, accertamento che il sistema sia ben accettato dall'organizzazione

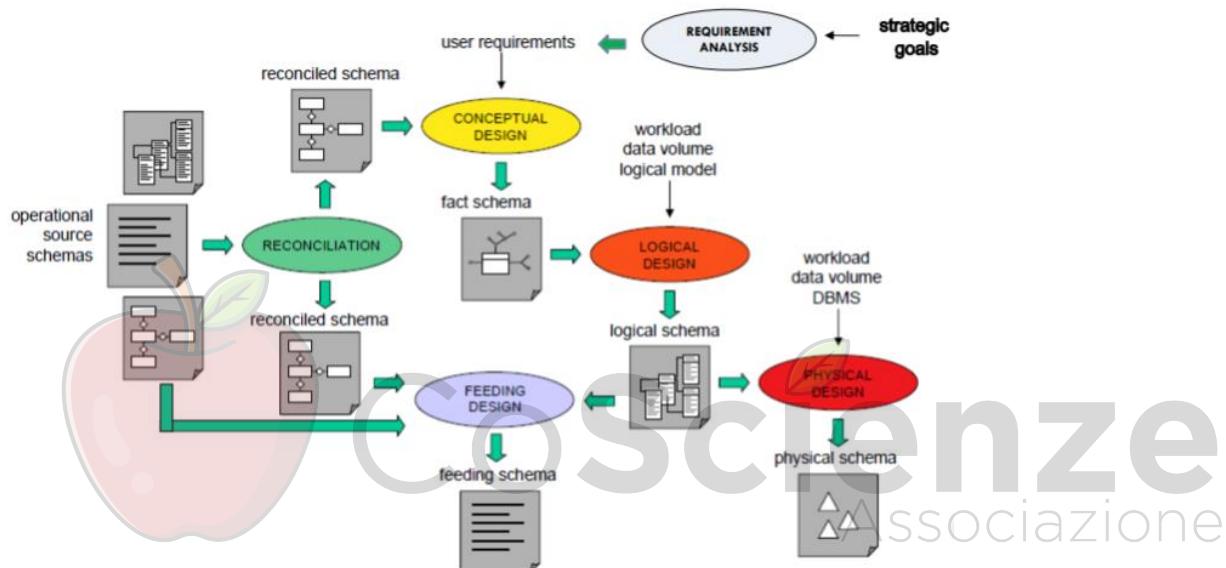
Progettazione di un Data mart

Fase	Ingresso	Uscita	Figure Coinvolte
Analisi e riconciliazione delle fonti	Schemi delle sorgenti	Schema riconciliato	Progettista; amministratore db operazionale;
Analisi dei requisiti	Obiettivi strategici	Specifiche dei requisiti; carico di lavoro preliminare	Progettista; utenti finali
Progettazione concettuale	Schema riconciliato; specifica dei requisiti	Schemi di fatto	Progettista; utenti finali
Raffinamento carico di lavoro, validazione schema concettuale	Schemi di fatto; carico di lavoro preliminare	Carico di lavoro; schemi di fatto validati	Progettista; utenti finali
Progettazione Logica	Schemi di fatto; modello logico target; carico di lavoro	Schema logico del Data mart	Progettista
Progettazione dell'alimentazione	Schemi delle sorgenti; schema riconciliato; schema logico del Data mart.	Procedure di alimentazione	Progettista; amministratori db operazionale
Progettazione fisica	Schema logico del Data mart; DBMS target; carico di lavoro	Schema fisico del Data mart	Progettista

- **Analisi e riconciliazione delle fonti dati:** analizzare e comprendere gli schemi delle sorgenti, normalizzazione, valutare la qualità dei dati
- **Analisi dei requisiti:** raccolta, filtro e documentazione dei requisiti, scelta dei fatti e granularità dei fatti (compromesso tra velocità e dettaglio)

- **Progettazione concettuale:** può essere usato il DFM(Dimensional Fact Model), creazione degli schemi di fatto.
- **Raffinamento del carico di lavoro e validazione dello schema concettuale:** formulazione delle interrogazioni direttamente sullo schema concettuale, verifica che le interrogazioni siano effettivamente esprimibili.
- **Progettazione logica:** scelta dell'implementazione (ROLAP o MOLAP), schemi logici, materializzazione delle viste.
- **Progettazione dell'alimentazione:** decisioni riguardanti il processo di alimentazione del livello riconciliato e del Data mart.
- **Progettazione fisica:** scelta degli indici per ottimizzare le prestazioni, riferimento ad un particolare DBMS, carico di lavoro e volume dei dati.

Schema del ciclo di vita del data mart

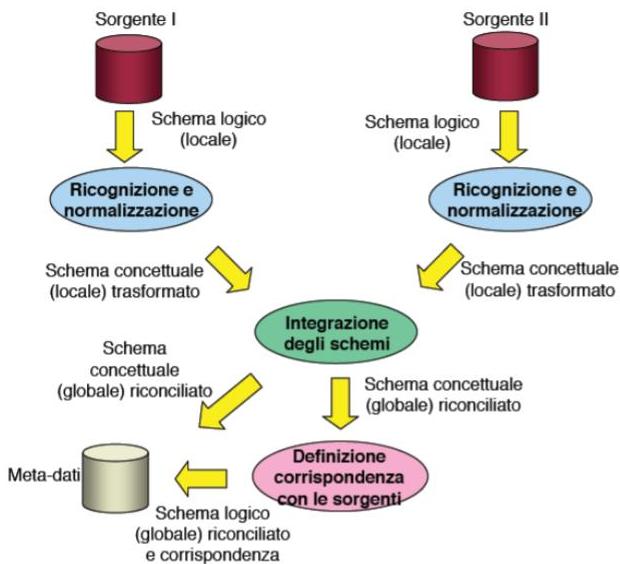


Analisi e riconciliazione delle fonti dati

Questa fase richiede di definire e documentare lo schema del livello dei dati operazionali, a partire da quale verrà alimentato il Dat mart. Riceve in ingresso gli schemi delle sorgenti e produce un insieme di metadati che modellano lo schema riconciliato e le corrispondenze tra gli elementi di quest'ultimo e quelli del sistema operazionale.

Le **figure coinvolte** sono: il progettista e l'amministratore dei database operazionali in quanto la sua conoscenza del dominio applicativo è indispensabile per normalizzare gli schemi.

A questo livello il modello architettonale di riferimento è quello a tre livelli, nel quale si suppone che il livello di dati riconciliato esiste, la scelta ricade su questo modello perché l'alimentazione diretta del DW è un compito troppo complesso per essere eseguito in modo atomico.



La figura dettaglia la fase di analisi e riconciliazione in caso di più sorgenti, di cui è noto il solo schema logico:

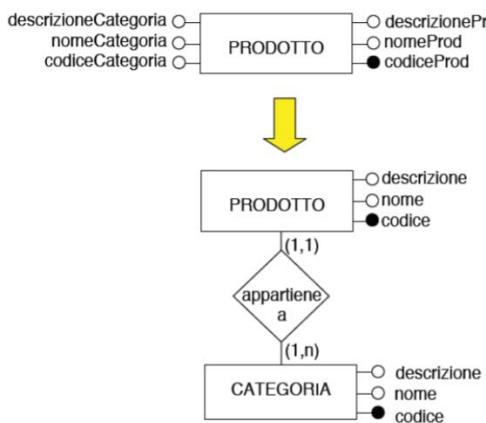
- **Ricognizione e normalizzazione** dei diversi schemi locali che produce un insieme di schemi concettuali, localmente consistenti e completi (va svolta anche in presenza di una sola sorgente dati, con più sorgenti viene ripetuta per ogni singolo schema locale)
- **Integrazione** che produce uno schema concettuale globalmente consistente
- Dallo schema ottenuto si effettua la progettazione logica dello schema riconciliato, per poi **definirne la corrispondenza** con schemi logici delle sorgenti.

Ricognizione e normalizzazione

Prima di procedere alla fase di progettazione concettuale il progettista deve acquisire conoscenza delle sorgenti operazionali attraverso attività di:

- **Ricognizione:** esame approfondito degli schemi locali mirato alla piena comprensione del dominio applicativo
- **Normalizzazione:** mira a correggere gli schemi locali al fine di modellare in modo più accurato il dominio applicativo

Il progettista deve verificare la completezza degli schemi. Le trasformazioni apportate allo schema non devono introdurre nuovi concetti, bensì rendere esplicativi tutti quelli ricavabili, oltre alle trasformazioni necessarie il progettista deve anche individuare eventuali porzioni degli schemi locali non utili al Data mart.

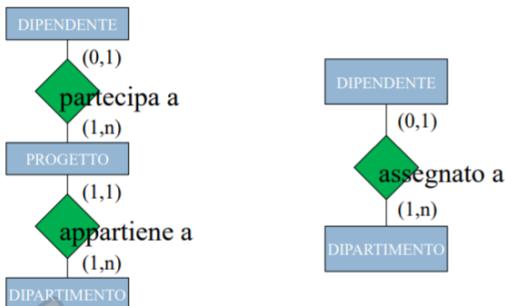


Integrazione

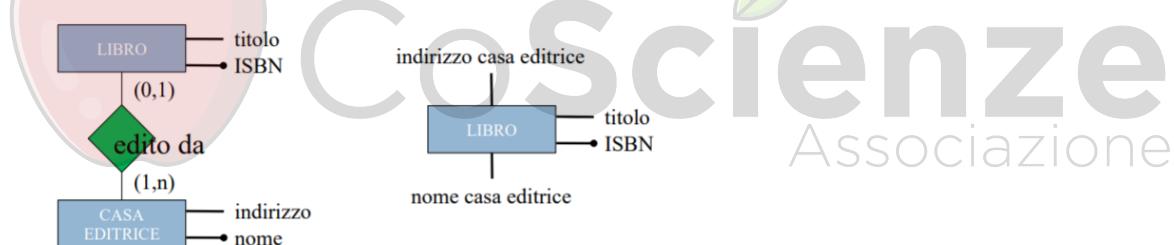
Consiste nella individuazione di corrispondenze tra i concetti degli schemi locali e nella risoluzione dei conflitti evidenziati. Lo scopo è di creare un unico schema globale i cui elementi sono correlati con i corrispondenti elementi degli schemi locali (**mapping**). In questa fase vanno anche identificati concetti distinti di schemi differenti che sono correlati attraverso proprietà semantiche (**proprietà inter-schema**). Per poter ragionare sui concetti degli schemi sorgente è necessario usare un unico formalismo (ER, UML, DTD ecc..) in modo da fissare i costrutti utilizzabili e la potenza espressiva.

Sono 5 i **problemi** individuabili durante l'integrazione:

- Diversità di prospettiva:** il punto di vista rispetto al quale diversi gruppi di utenti vedono uno stesso oggetto del dominio applicativo può differenziarsi in base agli aspetti per la funzione a cui essi sono preposti.

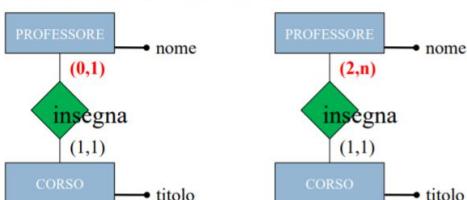


- Equivalenza dei costrutti del modello:** i formalismi di modellazione permettono di rappresentare uno stesso concetto utilizzando combinazioni diverse dei costrutti a disposizione



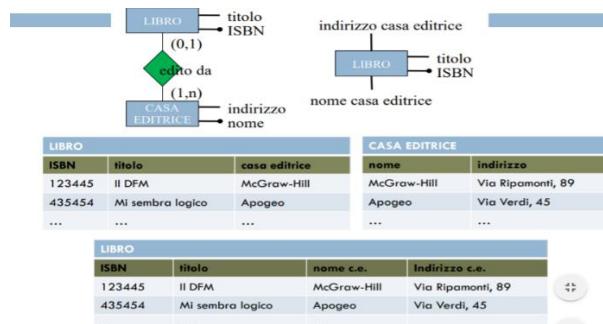
- Incompatibilità delle specifiche:** schemi diversi che modellano una stessa porzione del dominio applicativo racchiudono concetti diversi, in contrasto tra loro. Tali diversità possono coinvolgere ad esempio la scelta dei nomi, dei tipi di dati e dei vincoli di integrità.

- **Es:** in un caso un professore non può tenere più di un corso, nell'altro deve tenerne almeno 2.

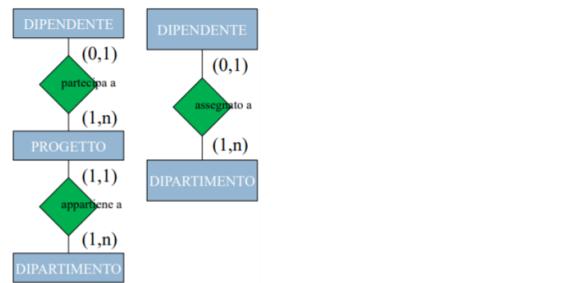


- Concetti comuni:** quattro sono le possibili relazioni esistenti tra due distinte rappresentazioni R_1 e R_2 di uno stesso concetto:

- **Identità:** si verifica quando vengono utilizzati gli stessi costrutti, il concetto è modellato dallo stesso punto di vista, quindi R_1 e R_2 coincidono.
- **Equivalenza:** si verifica quando R_1 e R_2 non sono le stesse poiché sono stati utilizzati costrutti diversi ma equivalenti. (sono equivalenti se le loro istanze possono essere messe in corrispondenza 1 a 1)



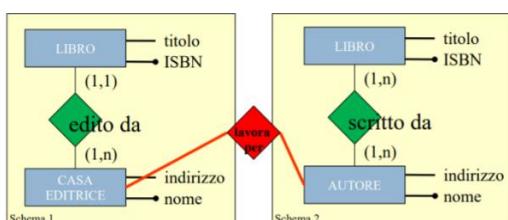
- **Comparabilità:** questa situazione si verifica quando R_1 e R_2 non sono né identici né equivalenti ma, i costrutti utilizzati e i punti di vista dei progettisti non sono in contrasto tra loro.



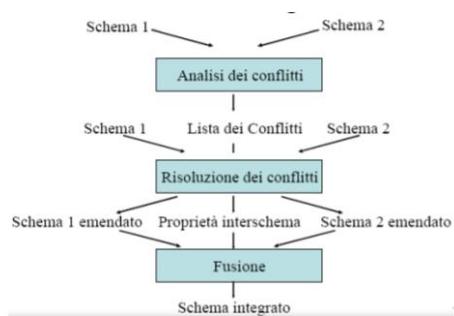
- **Incompatibilità:** questa situazione si verifica quando R_1 e R_2 sono in contrasto a causa dell'incoerenza nelle specifiche, ovvero quando la realtà modellata da R_1 nega quella modellata da R_2 .

Ad esclusione dell' "identità", tutti questi casi determinano dei conflitti ovvero quando due rappresentazioni R_1 ed R_2 dello stesso concetto non sono identiche.

5. **Concetti correlati:** a seguito dell'integrazione, molti concetti diversi, ma correlati, verranno a trovarsi nello stesso schema dando vita a nuove relazioni che non erano percepibili in precedenza. Tali relazioni sono dette **proprietà inter-schema**.



Sono 4 le fasi dell'integrazione:



1. **Preintegrazione:** Durante questa fase viene svolta l'analisi della sorgente dati, che porta a definire la politica generale dell'integrazione. Bisogna prendere decisioni sulle porzioni degli schemi che dovranno essere integrate e sulle strategie di integrazione.

2. **Comparazione degli schemi:** Questa fase consiste in un'analisi comparativa dei diversi schemi e mira a identificare correlazioni e conflitti tra concetti in essi espressi. I tipi di conflitti che possono essere evidenziati sono di diverso tipo:

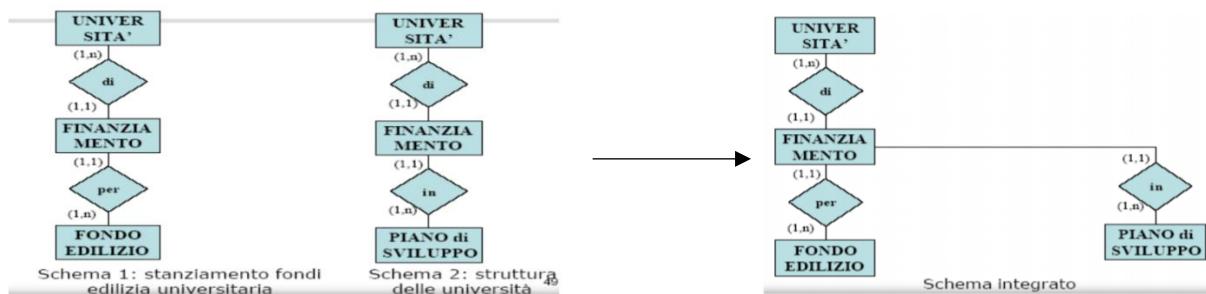
- **Di eterogeneità:** diversità dovute all'utilizzo di formalismi con diverso potere espressivo negli schemi sorgenti
- **Semantici:** due schemi modellano la stessa porzioni di mondo reale ma a un livello diverso di astrazione
- **Sui nomi:** differenze nelle terminologie utilizzate nei diversi schemi sorgenti (omonimie e sinonimie)
- **Strutturali:** scelte diverse nella modellazione di uno stesso concetto, questi conflitti possono a loro volta essere di vario tipo:
 - **Di tipo:** uno stesso concetto è modellato utilizzando due costrutti diversi
 - **Di dipendenza:** due o più concetti sono correlati con dipendenze diverse in schemi diversi
 - **Di chiave:** per uno stesso concetto vengono utilizzati identificatori diversi in schemi diversi
 - **Di comportamento:** diverse politiche di cancellazione/modifica dei dati vengono adottate per uno stesso concetto in schemi diversi.

3. **Allineamento degli schemi:** Lo scopo di questa fase è la risoluzione dei conflitti evidenziati al passo precedente, mediante primitive di trasformazione degli schemi sorgenti o dello schema riconciliato temporaneo. Alcune **primitive** tipiche riguardano il cambio dei nomi e dei tipi degli attributi, la modifica delle dipendenze funzionali e dei vincoli sugli schemi. È qui che il progettista definisce il **mapping** tra gli elementi degli schemi sorgenti e quelli dello schema riconciliato.

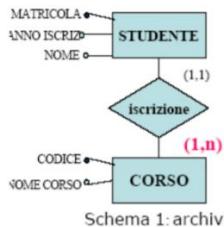
4. **Fusione e ristrutturazione degli schemi:** Nell'ultima fase gli schemi allenati vengono fusi per formare un unico schema riconciliato, solitamente si sovrappongono i concetti comuni a cui saranno collegati i rimanenti concetti degli schemi locali. Dopo questa operazione ulteriori trasformazioni permetteranno di migliorare lo schema rispetto a:

- **Leggibilità:** facilita e velocizza le successive fasi di progettazione
- **Completezza:** ricerca di proprietà inter-schema non evidenziate in precedenza
- **Minimalità:** eliminare ridondanza di concetti duplicati o comunque derivabili

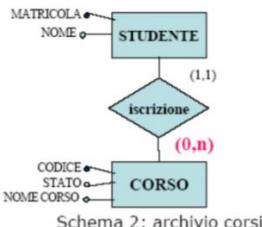
Esempio di schemi compatibili in seguito all'integrazione



Esempi di schemi incompatibili



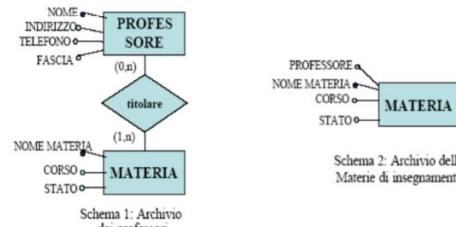
Schema 1: archivio studenti



Schema 2: archivio corsi

Nello schema 1 sono archiviati tutti gli studenti iscritti ad un corso universitario, mentre lo schema 2 include tutti i corsi attivati e quindi anche quelli a cui non è iscritto alcuno studente.

1. Nello schema integrato si sceglie la seconda soluzione, perché meno restrittiva.



Schema 2: Archivio delle Materie di insegnamento

Nello schema 1 sono archiviati tutti i professori che possono essere o meno titolari di materie di insegnamento, mentre nello schema 2 sono memorizzate tutte le materie.

2. Nello schema integrato si sceglie la prima soluzione, perché più completa e meno restrittiva.

Definizione delle corrispondenze

Il risultato dell'analisi delle sorgenti operazionali è composto da due elementi:

- **Schema riconciliato**, in cui sono stati risolti i conflitti presenti tra gli schemi locali
- **Insieme di corrispondenze** tra gli elementi presenti negli schemi sorgenti e quello dello schema destinazione (necessarie per la fase di progettazione degli ETL)

L'approccio per stabilire la corrispondenza tra i due livelli dell'architettura prevede che lo schema globale sia espresso in termini degli schemi sorgente detto **GAV (Global as view)**. Con questo approccio ad ogni concetto dello schema globale è associata una vista definita in base a concetti degli schemi sorgenti. Per fare ciò con il GAV si sostituisce ad ogni concetto dello schema globale la vista che lo definisce in termini di concetti degli schemi locali (**unfolding**).

Esempio: Mapping di tipo GAV

```

// DB1 Magazzino
ORDINI2001(chiaveO, chiaveC, data ordine, impiegato)
CLIENTE(chiaveC, nome, indirizzo, città, regione, stato)

.....
// DB2 Amministrazione
CLIENTE(chiaveC, partitalva, nome, telefono, fatturato)
FATTURE(chiaveE, data, chiaveC, importo, iva)
STORICO_ORDINI2000(chiaveO, chiaveC, data ordine, impiegato)

.....
CREATE VIEW CLIENTE AS
SELECT CL1.chiaveC, CL1.nome, CL1.indirizzo, CL1.città, CL1.regioni,
       CL1.statuto, CL2.partitalva, CL2.telefono, CL2.fatturato
  FROM DB1.CLIENTE AS CL1, DB2.CLIENTE AS CL2
 WHERE CL1.chiaveC = CL2.chiaveC;

CREATE VIEW ORDINI AS
SELECT * FROM DB1.ORDINI2001
UNION
SELECT * FROM DB2.STORICO_ORDINI2000;
  
```

Vedere slide 138 – 145 per esercizio intero sull'integrazione.

Analisi dei requisiti utente

Obiettivi dell'analisi dei requisiti utente

- Raccolta delle esigenze di utilizzo del Data mart espresse dagli utenti finali.
- Importanza strategica perché influenza tutte le decisioni da prendere durante le diverse attività, con un ruolo primario nel determinare:
 - o Schema concettuale dei dati del DW;
 - o Progetto dell'alimentazione;
 - o Specifiche delle applicazioni per l'analisi dei dati;
 - o Architettura del sistema;
 - o Piano di avviamento e formazione;
 - o Linee guida per la manutenzione e l'evoluzione del sistema;

Fonti dell'analisi dei requisiti utente

Le fonti da cui attingere per i requisiti sono i **business users**, i futuri utenti del data warehouse.

- Dialogo spesso infruttuoso a causa del differente linguaggio usato;
- Porre grande cura nella fase di analisi, la soddisfazione degli utenti dipende dall'accuratezza con la quale le loro richieste ottengono un'efficace risposta nel sistema.
- Sono previste sia **Interviste** che **Riunioni coordinate**

Gli aspetti tecnici vengono individuati mediante l'interazione con i gestori del sistema operazionale:

- I requisiti riguardano vincoli imposti al sistema e mirano a garantire livelli ottimali di prestazioni ed una facile integrazione con il sistema.

I fatti

I fatti sono concetti su cui gli utenti finali del Data mart baseranno il processo decisionale.

- Ogni fatto descrive una categoria di eventi che si verificano in azienda.

Caratteristiche che guidano il progettista verso la determinazione dell'insieme dei fatti per il Data mart.

- **Aspetti dinamici:** gli eventi che vengono descritti dal fatto devono avere una componente temporale;
- **Dominio applicativo;**
- **Tipo di analisi** che l'utente vuole eseguire (Un fatto di interesse per un Data mart potrebbe non esserlo per un altro)

Individuare i fatti non è sufficiente:

- Per ognuno di essi è necessario disporre di informazioni di contorno, definite con l'aiuto della documentazione del livello riconciliato:
 - Possibili dimensioni (granularità)
 - Possibili misure
 - Intervallo di storicizzazione

Granularità

Focalizzare le dimensioni di un fatto è importante per determinare la granularità, ovvero il più fine livello di dettagli a cui i dati saranno rappresentati nel Data mart.

La scelta della granularità di rappresentazione di un fatto nasce da un delicato compromesso tra due esigenze contrapposte:

- Raggiungere un'elevata **flessibilità di utilizzo**, che richiederebbe di mantenere la stessa granularità del livello operazionale;
- Conseguire **buone prestazioni** con la necessità di avere un consistente gradi di sintesi dei dati

Misure e intervallo di storicizzazione

La valutazione delle misure con cui quantificare ciascun fatto ha un ruolo preliminare e orientativo, in quanto la definizione dettagliata delle misure da abbinare al fatto è rimandata alla successiva fase di progettazione concettuale.

È l'arco temporale che gli eventi memorizzati nel Data mart devono coprire

- Valori tipici variano da 3 a 5 anni

Caso di studio

Data mart per la gestione di approvvigionamenti e vendite in una catena di supermercati.

Requisiti utente:

Fatto	Possibili dimensioni	Possibili misure	Storicità
inventario di magazzino	prodotto, data, magazzino	quantità in magazzino	1 anno
vendite	prodotto, data, negozio	quantità venduta, importo, sconto	5 anni
linee d'ordine	prodotto, data, fornitore	quantità ordinata, importo, sconto	3 anni

Caso di studio (2)

Carico di lavoro preliminare: Raccolta delle specifiche relative alle interrogazioni di analisi più frequenti sul Data mart.

Fatto	Interrogazione
Inventario di magazzino	<ul style="list-style-type: none"> • quantità media di prodotto presente mensilmente in tutti i magazzini • andamento giornaliero delle scorte complessive per ogni tipo di prodotto • prodotti per i quali è stata esaurita la scorta di magazzino contemporaneamente in almeno un'occasione durante la settimana scorsa.
Vendite	<ul style="list-style-type: none"> • incasso totale giornaliero di ciascun negozio • per un negozio, incassi relativi alle diverse categorie di prodotti durante un certo giorno • riepilogo annuale degli incassi per regione relativamente ad un dato prodotto • quantità totali di ciascun tipo di prodotto venduto durante l'ultimo mese
Linee d'ordine	<ul style="list-style-type: none"> • quantità totale ordinata annualmente presso un certo fornitore • importo giornaliero ordinato nell'ultimo mese per un certo tipo di prodotto • sconto massimo applicato da ciascun fornitore durante l'ultimo anno per ciascuna categoria di prodotto

Modellazione Concettuale

Modellizzazione concettuale

Affronta il problema della traduzione dei requisiti in termini di un modello astratto, indipendente dal DBMS.

- Non esistono standard di modello o di processo;

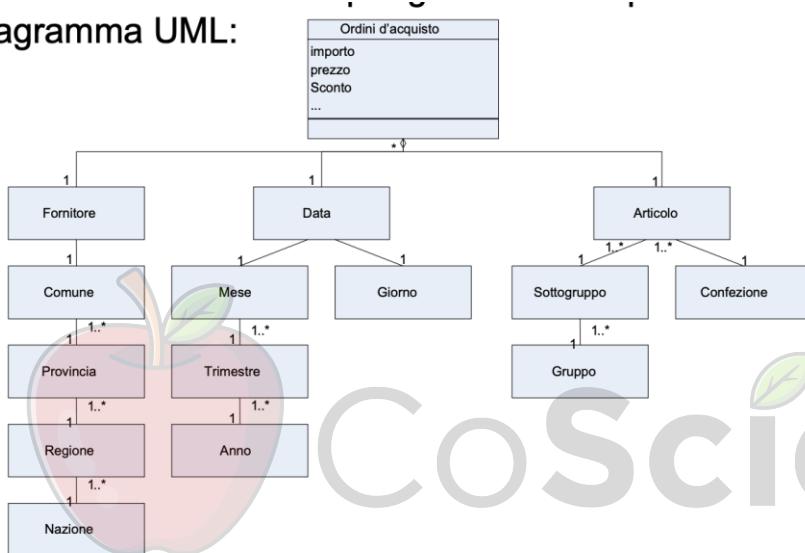
Il modello Entity-relationship (ER) è diffuso come strumento di modellizzazione concettuale dei Data mart.

- Esso è però orientato alle associazioni tra i dati e non alla sintesi;
- È sufficientemente espressivo per rappresentare la maggior parte dei concetti, ma non è in grado di mettere in luce il modello multidimensionale.

Modellazione concettuale in UML

Un modello concettuale per gli ordini d'acquisto basato su diagramma UML:

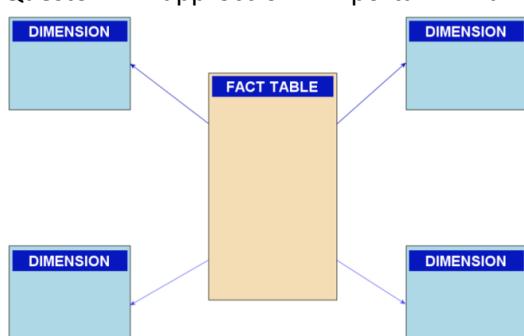
diagramma UML:



Lo schema a Stella (Star Schema)

Lo schema a stella è un modello logico che può essere usato per la modellazione concettuale.

- Usare lo schema a stella per i Data mart equivale a modellare uno schema logico per un DB relazionale.
- Questo approccio porta a schemi fortemente de normalizzati.



Dimensional Fact Model (DFM)

È un modello concettuale concepito per il supporto allo sviluppo di Data mart.

È una specializzazione del modello multidimensionale per applicazioni di Data warehousing.

È un modello concettuale grafico per Data mart, pensato per:

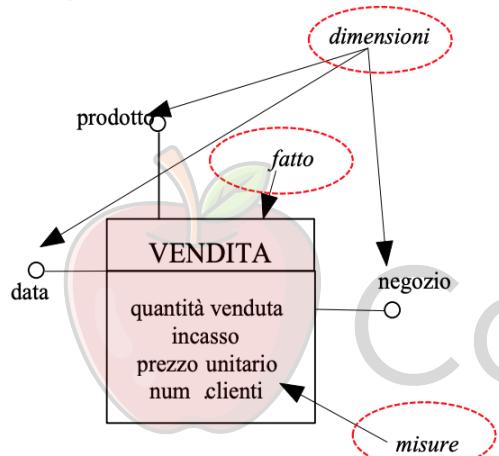
- Supportare efficacemente il progetto concettuale;
- Creare un ambiente su cui formulare in modo intuitivo le interrogazioni dell'utente;
- Permettere il dialogo tra progettista e utente finale per raffinare le specifiche dei requisiti;
- Creare una piattaforma stabile da cui partire per il progetto logico (indipendentemente dal modello logico target);
- Restituire una documentazione a posteriori espressiva e non ambigua.

La rappresentazione concettuale generata dal DFM consiste in un insieme di schemi di fatto.

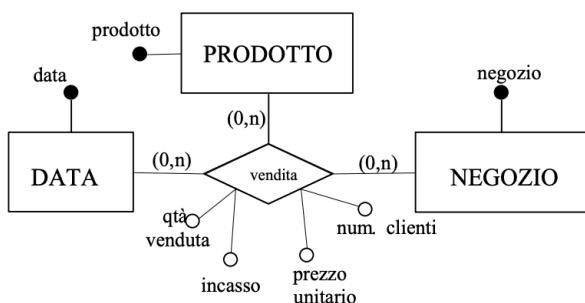
- Gli elementi di base modellati dagli schemi di fatto sono i fatti, le misure, le dimensione e le gerarchie.

Esempio DFM

Semplice schema di fatto per le vendite:



Schema ER corrispondente:



IL DFM: costrutti di base

Un **fatto** è un concetto di interesse per il processo decisionale:

- Tipicamente modella un insieme di eventi che accadono nell'impresa (ES: vendite, spedizioni)
- È essenziale che un fatto abbia aspetti dinamici ovvero evolva nel tempo.

Una **misura** è una proprietà numerica di un fatto e ne descrive un aspetto quantitativo di interesse per l'analisi (ES: ogni vendita è misurata dal suo incasso)

- Le misure vengono in genere usate per effettuare calcoli.

Una **dimensione** è una proprietà con dominio finito di un fatto e ne descrive una coordinata di analisi (dimensioni, tipiche per il fatto vendite sono prodotto, negozio, data)

- Un fatto esprime una associazione molti-a-molti tra le dimensioni.
- I fatti hanno natura dinamica, rappresentata da una dimensione temporale.

Un evento primario è una particolare occorrenza di un fatto, individuata da una ennupla costituita da un valore per ciascuna dimensione.

- A ciascun evento primario è associato un valore per ciascuna misura.
- Esempio: il giorno 10/10/2001 il negozio 'DiTutto' ha venduto 10 confezioni di detersivo Brillo per un incasso complessivo di 25€

DFM: Attributi Dimensionali

Gli **attributi dimensionali** sono le dimensioni e gli attributi che le descrivono.

- Esempio: un prodotto è descritto da tipo, categoria, marca.
- Le relazioni tra gli attributi dimensionali sono espresse dalle gerarchie.

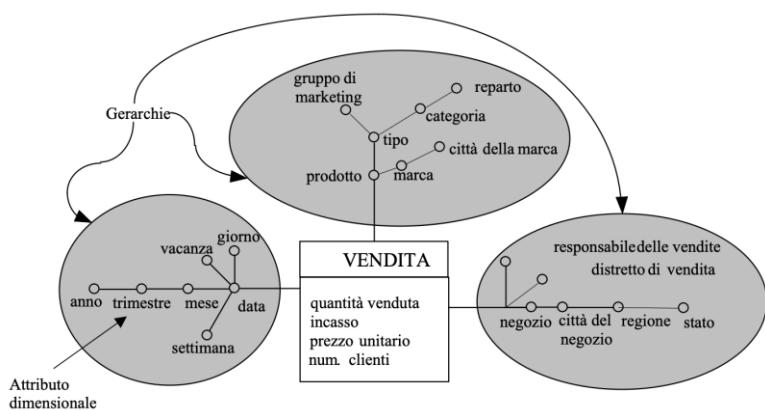
Una **gerarchia** è un albero direzionale i cui nodi sono attributi dimensionali e i cui archi modellano associazioni molti-a-uno tra coppie di attributi dimensionali.

- Racchiude una dimensione, posta alla radice dell'albero, e tutti gli attributi dimensionali che la descrivono.

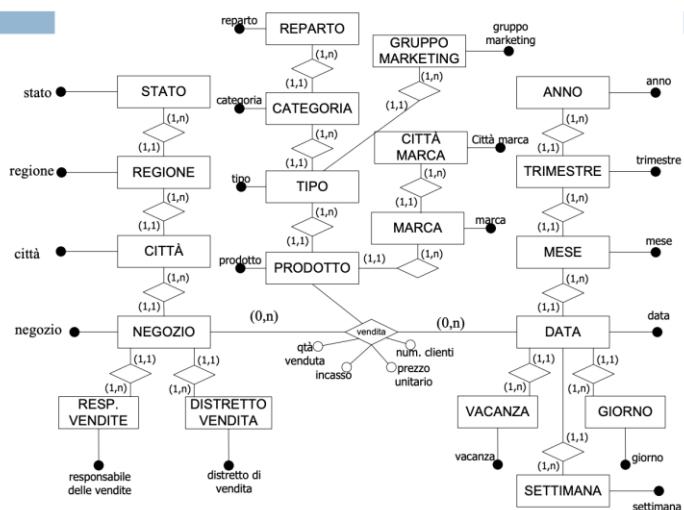
ATTENZIONE: Gerarchie ER != Gerarchie DFM

DFM Arricchito

Schema di fatto arricchito per la vendita:



Schema ER del DFM Vendita



Naming conventions

Tutti gli attributi dimensionali in ciascuno schema di fatto devono avere nomi diversi;

Eventuali nomi uguali dicono essere differenziati qualificandoli con il nome di un attributo dimensionale che li precede nella gerarchia.

- Ad esempio, **warehouse city** è la città in cui si trova un magazzino, mentre **store city** è la città in cui si trova un negozio

I nomi degli attributi non dovrebbero riferirsi esplicitamente al fatto a cui appartengono.

- Es. si evitino shipped product e shipment date

Attributi con lo stesso significato in schemi diversi devono avere lo stesso nome

Evento secondario

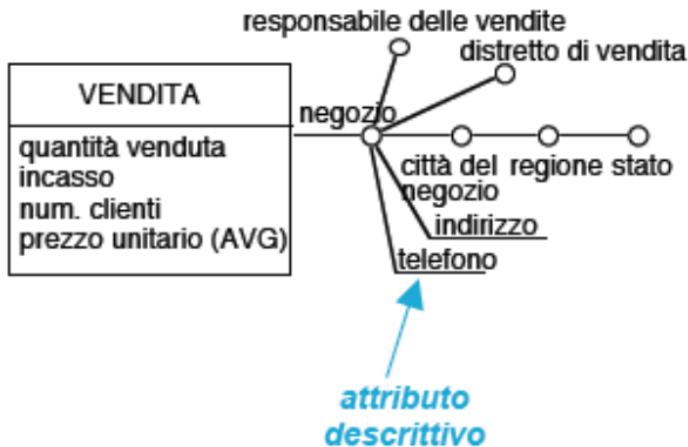
Dato un insieme di attributi dimensionali, ciascuna ennupla di valori individua un **evento secondario** che aggrega tutti gli eventi primari corrispondenti.

- A ciascun evento secondario è associato un valore per ciascuna misura, che riassume in sé tutti i valori della stessa misura negli eventi primari corrispondenti.
- ES. Le vendite possono essere raggruppate a seconda della categoria dei prodotti venduti, oppure a seconda del mese in cui sono effettuate le vendite, oppure a seconda della città in cui si trova il negozio, ecc...

Attributi descrittivi

Attributi descrittivi: specificano le proprietà degli attributi dimensionali di una gerarchia, e sono determinati tramite dipendenze funzionali.

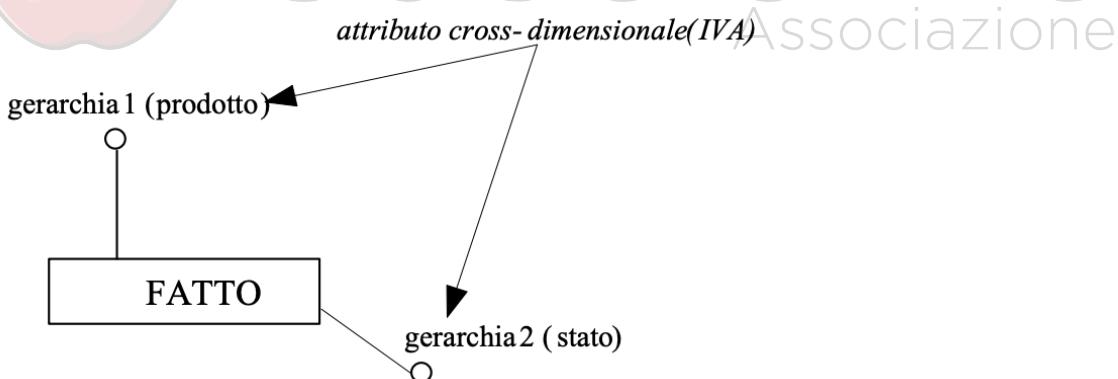
- Non possono essere usati per l'aggregazione poiché hanno spesso domini con valori continui.
- Un attributo descrittivo non può essere usato per identificare singoli eventi né per effettuare calcoli;
- Es. numero di telefono di un negozio, indirizzo di un negozio.



Attributi cross-dimensional

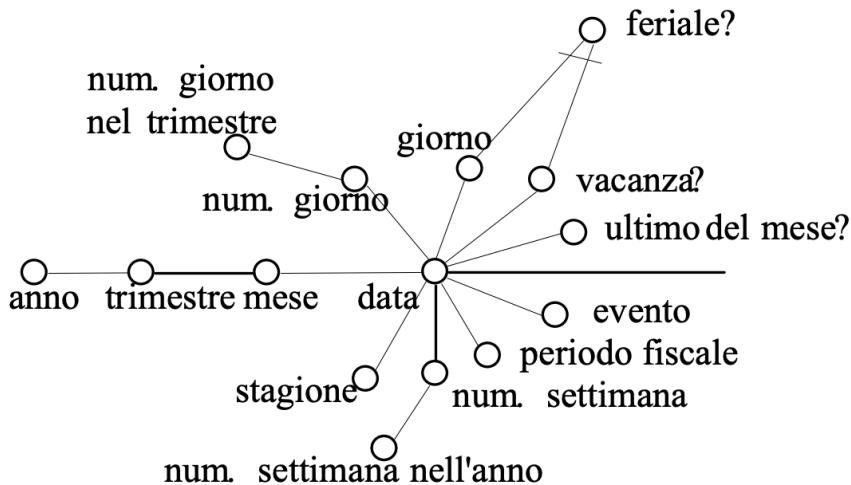
Attributi cross-dimensional: sono attributi dimensionali o descrittivi il cui valore è determinato mediante la combinazione di due o più attributi dimensionali.

- Gli attributi dimensionali possono appartenere anche a gerarchie diverse.
- ES. l'IVA su un prodotto dipende sia dalla categoria del prodotto che dallo stesso in cui esso è venduto.



Esempio di gerarchia temporale completa

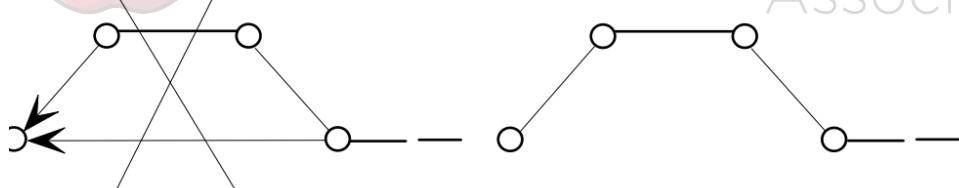
L'attributo booleano **feriale?** È determinato congiuntamente dal giorno e dal booleano vacanza?



Convergenza

La **convergenza** riguarda la struttura delle gerarchie.

- Sullo schema di fatto le convergenze sono denotate da due o più archi, in genere appartenenti alla stessa gerarchia, che terminano nello stesso attributo dimensionale.
- In presenza di una gerarchia che non ha una struttura ad albero non è più possibile determinare univocamente il verso degli archi e, per fare ciò, gli archi convergenti devono essere orientati.
- Attributi apparentemente uguali non determinano sempre una convergenza.
- Se uno dei percorsi alternativi non comprende attributi intermedi, non ha ragione di esistere: la convergenza è infatti del tutto ovvia grazie alla transitività delle dipendenze funzionali:

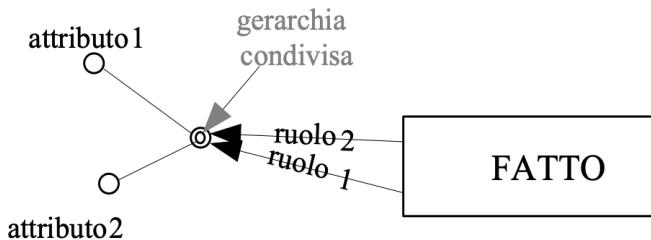


Convergenza ridondante e sua rappresentazione corretta

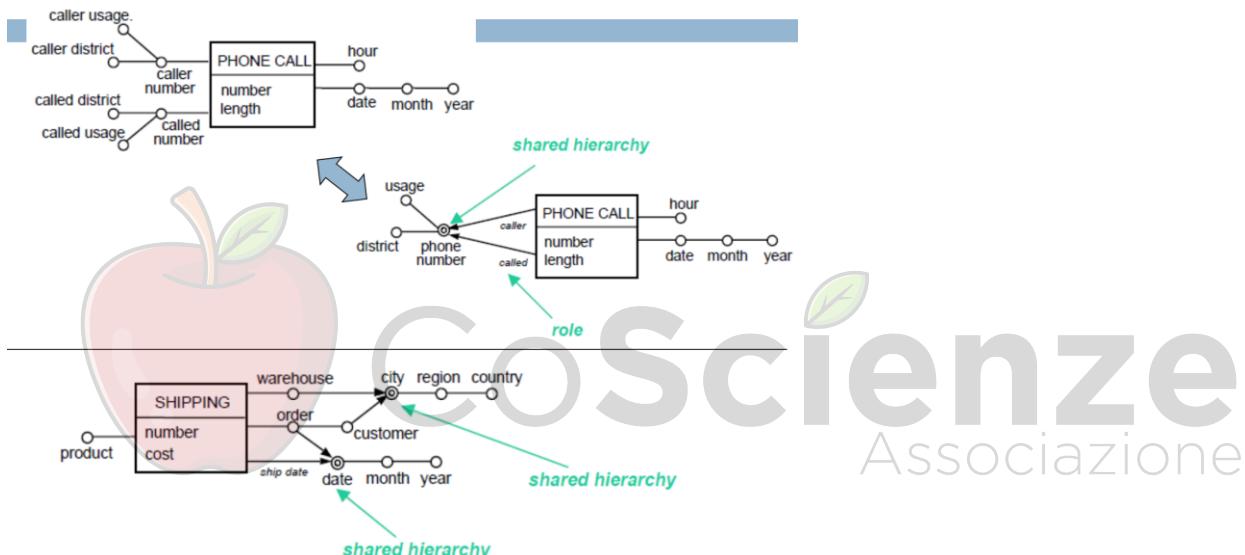
Gerarchie condivise

Negli schemi di fatto spesso si rende necessario duplicare intere porzioni di gerarchie e ciò comporta l'uso di diversi nomi per evitare ambiguità.

Tramite le **gerarchie condivise** si introduce una notazione grafica abbreviata che migliora la leggibilità dello schema. Si introducono quando si hanno significati diversi per lo stesso tipo di dati e il significato viene espresso inserendo il ruolo sull'arco entrante della gerarchia



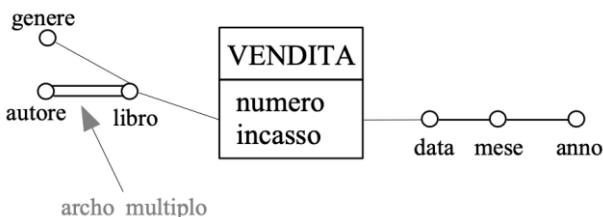
Esempio: Gerarchie condivise



Archi multipli

Un arco multiplo tra due attributi a e b indica che ad ogni singolo valore di a possono corrispondere valori di b

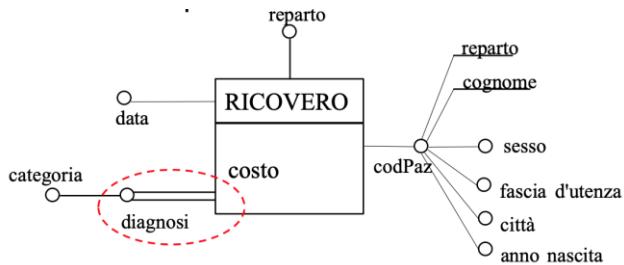
ES: Schema di fatto per le vendite dei libri:



Il significato di un arco multiplo che va da un attributo autore ad un attributo libro sta nel fatto che tra autore e libro esiste una **associazione M-N**

Aggregazione su dimensioni

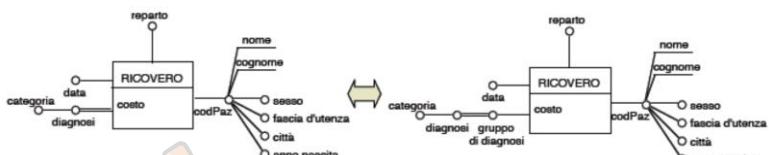
Es: Schema di fatto per ricoveri:



Nel momento in cui un attributo entra in una dimensione piuttosto che in un attributo qualsiasi, il caso diventa più complesso.

Infatti, è possibile aggregare i ricoveri in base alle diagnosi in uscita, ma anche selezionare le diagnosi in base ai ricoveri.

Alternativamente



Archi opzionali

Vengono impiegati per modellare associazioni dello schema di fatto non definite per un particolare sottoinsieme di eventi.

- Si rappresenta con un trattino sull'arco

Se r è l'arco opzionale, bisogna distinguere se esso determina un attributo o una dimensione.

- Se r determina la dimensione di allora essa è opzionale, ossia esistono alcuni eventi primari identificati solo dalle altre dimensioni.
Esempio: la promozione su un prodotto, identificato da una dimensione, vale solo per alcune combinazioni di **prodotto-negoziodata**

Copertura tra archi opzionali

Se esistono più archi opzionali uscenti da uno stesso attributo è possibile definire la copertura, ossia stabilire una relazione tra le diverse opzionalità.

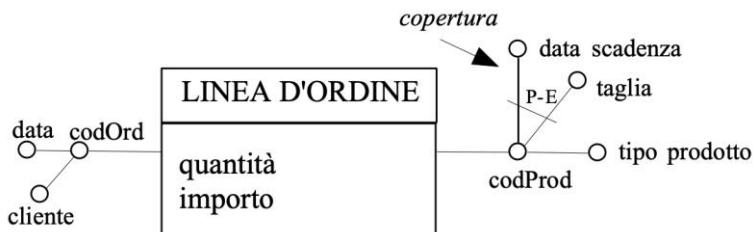
Sia a un attributo dimensionale con archi opzionali verso i propri figli b₁,...,b_m. Allora la copertura si dice:

- **Totale** se per ogni valore di a è sempre associato almeno un valore dei figli o **parziale** se esistono valori di a per i quali tutti i figli sono indefiniti,
- **Esclusivo** se per ogni valore di a si ha al massimo un valore per uno dei figli o **sovraposta** se invece esistono valori di a abbinati a due o più figli

I tipi di copertura sono identificati da T-E, T-S, P-E, P-S.

Esempio di copertura

Copertura per un insieme di archi opzionali.



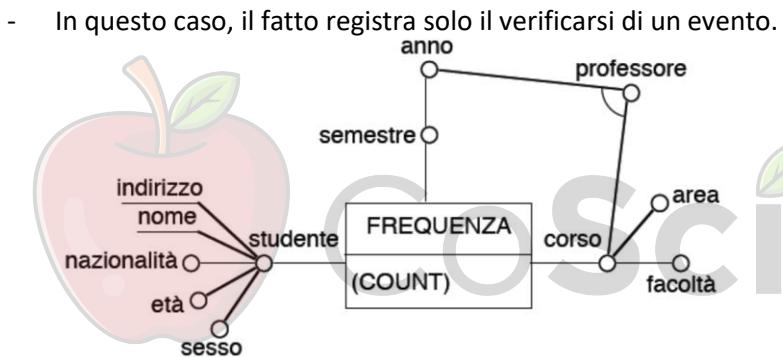
Prodotti di tre tipi: alimentari, abbigliamento, casalinghi.

Quindi **data scadenza** e **taglia** sono definiti solo per alimentari

E casalinghi rispettivamente: la copertura è P-E.

Schemi di fatto vuoti

Uno schema di fatto si dice vuoto se non ha misure:



Additività

L'aggregazione richiede di definire un operatore adatto per comporre i valori delle misure che caratterizzano gli eventi primari in valori da abbinare a ciascun evento secondario

Misure additive

Una misura è detta **additiva** su una dimensione:

- Se i suoi valori possono essere aggregati lungo la corrispondente gerarchia tramite l'operatore di somma;

Una misura è **non-additiva**:

- Se non può essere aggregata lungo una data gerarchia tramite l'operatore di somma.

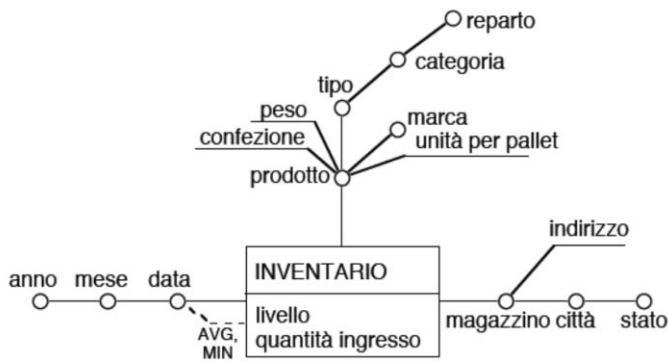
Una misura è **non-aggregabile**:

- Se non può essere aggregata lungo una qualsiasi gerarchia tramite l'uso di qualsiasi operatore di aggregazione.

Una misura **non-additiva** e **non-aggregabile** se nessun operatore di aggregazione può essere usato su di essa.

Esempio

Il livello di inventario non è additivo sul tempo, ma lo è sulle altre dimensioni:



Tipi di misure additive

Tre tipi:

Di flusso (periodo temporale, come incasso mensile, num_prodotti)

- Può essere valutata cumulativamente alla fine di un periodo di tempo;
- Può essere aggregata tramite tutti gli operatori standard;

di livello (particolari instanti di tempo, come gli abitanti di una città)

- Valutata in un preciso istante (snapshot)
- Non additiva lungo la dimensione temporale

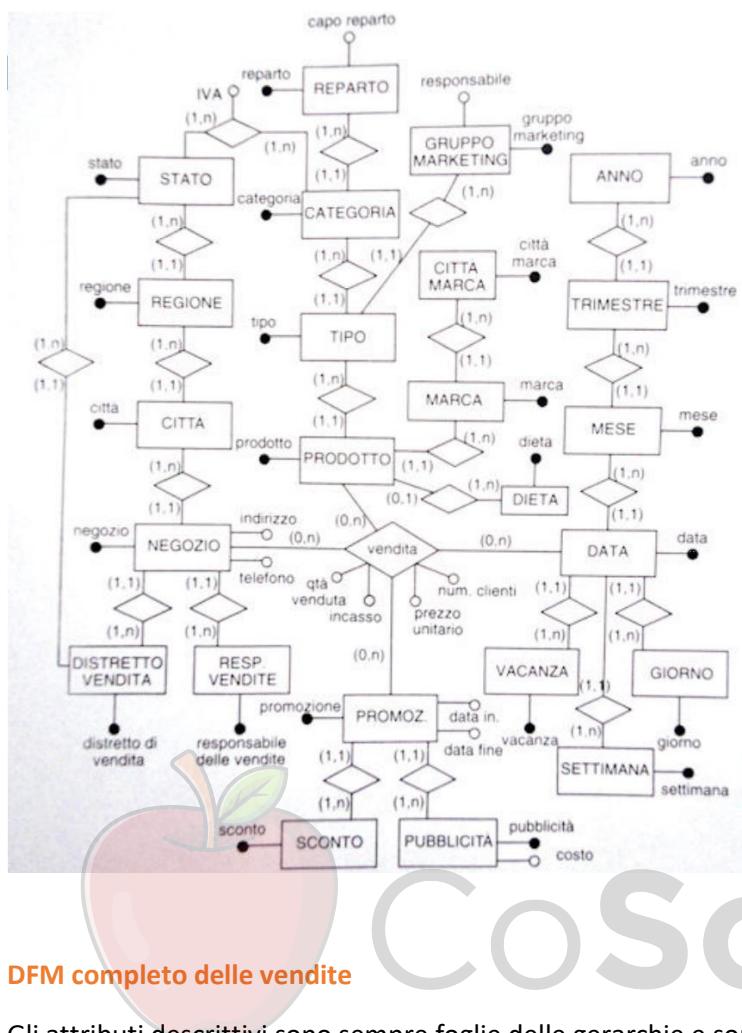
Unitarie (in particolari instanti di tempo, ma in termini relativi come il prezzo unitario di un prodotto o una percentuale di sconto)

- Valutata ad un certo istante ed espressa con termini relativi;
- Non additiva lungo qualsiasi dimensione;

Gerarchie temporali Gerarchie non temporali

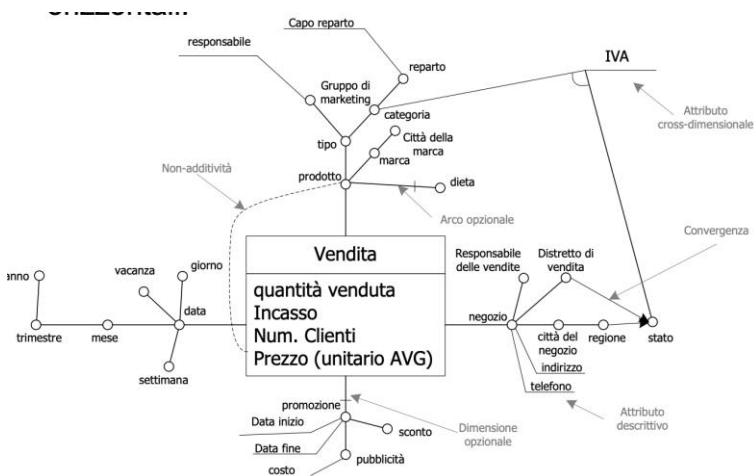
Misure di flusso	SUM, AVG, MIN, MAX	SUM, AVG, MIN, MAX
Misure di livello	AVG, MIN, MAX	SUM, AVG, MIN, MAX
Misure unitarie	AVG, MIN, MAX	AVG, MIN, MAX

Schema ER del DFM vendite



DFM completo delle vendite

Gli attributi descrittivi sono sempre foglie delle gerarchie e sono rappresentati nel DFM da linee orizzontali



Eventi

Definiamo evento un'istanza che popola uno schema di fatto;

Gli eventi possono essere aggregati rispetto i valori degli attributi lungo le gerarchie;

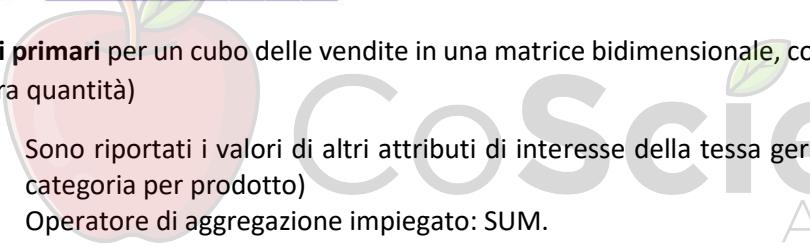
Le misure degli eventi aggregati sono ottenute aggregando le misure degli eventi corrispondenti nello schema dei fatti originale:

- Gli operatori di aggregazione standard: SUM, MIN, MAX, AVG, COUNT

L'aggregazione computa le misure con una granularità grezza rispetto a quella nello schema dei fatti originale:

- La riduzione del dettaglio è usualmente ottenuta risalendo nella gerarchia.

Aggregazione di eventi



year [1999 | 2000]
quarter [I '99 | II '99 | III '99 | IV '99 | I '00 | II '00 | III '00 | IV '00]

category	type	product	1999				2000			
			I '99	II '99	III '99	IV '99	I '00	II '00	III '00	IV '00
home cleaning	Washing powder	Brillo	100	90	95	90	80	70	90	85
		Sbianco	20	30	20	10	25	30	35	20
		Lucido	60	50	60	45	40	40	50	40
		Manipulite	15	20	25	30	15	15	20	10
food	soap	Scent	30	35	20	25	30	30	20	15
		Latte F Slurp	90	90	85	75	60	80	85	60
		Latte U Slurp	60	80	85	60	70	75	65	
		Yogurt Slurp	20	30	40	35	30	35	35	20
		Bevimi	20	10	25	30	35	30	20	10
		Colissima	50	60	45	40	50	60	45	40

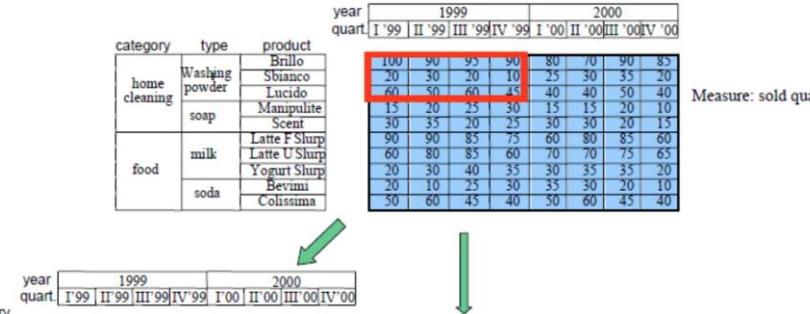
Measure: sold quantity

category	type	product	1999				2000			
			I '99	II '99	III '99	IV '99	I '00	II '00	III '00	IV '00
home clean			225	225	220	200	190	185	215	170
food			240	270	280	240	245	275	260	195

Eventi primari per un cubo delle vendite in una matrice bidimensionale, con dimensioni prodotto e trimestre (misura quantità)

- Sono riportati i valori di altri attributi di interesse della tessa gerarchia (anno per trimestre, tipo e categoria per prodotto)
- Operatore di aggregazione impiegato: SUM.

Eventi secondari sul pattern tipo e trimestre



year [1999 | 2000]
quarter [I '99 | II '99 | III '99 | IV '99 | I '00 | II '00 | III '00 | IV '00]

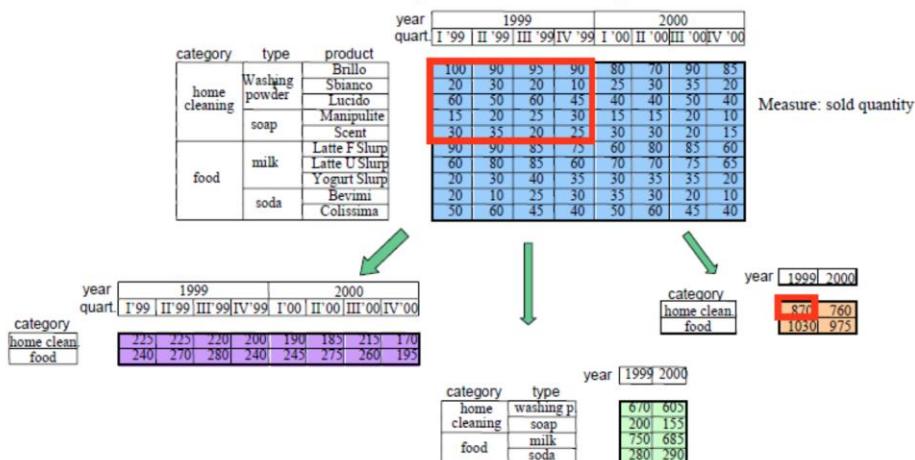
category	type	product	1999				2000			
			I '99	II '99	III '99	IV '99	I '00	II '00	III '00	IV '00
home cleaning	Washing powder	Brillo	100	90	95	90	80	70	90	85
		Sbianco	20	30	20	10	25	30	35	20
		Lucido	60	50	60	45	40	40	50	40
		Manipulite	15	20	25	30	15	15	20	10
food	soap	Scent	30	35	20	25	30	30	20	15
		Latte F Slurp	90	90	85	75	60	80	85	60
		Latte U Slurp	60	80	85	60	70	75	65	
		Yoguri Slurp	20	30	40	35	30	35	35	20
		Bevimi	20	10	25	30	35	30	20	10
		Colissima	50	60	45	40	50	60	45	40

Measure: sold quantity

category	type	product	1999				2000			
			I '99	II '99	III '99	IV '99	I '00	II '00	III '00	IV '00
home clean			225	225	220	200	190	185	215	170
food			240	270	280	240	245	275	260	195

category	type	product	1999 2000	
			washing p	
home cleaning	washing p	670	605	
home cleaning	soap	200	155	
food	milk	750	685	
food	soda	280	290	

Eventi secondari sul pattern categoria e anno



Aggregazione di misure non-additive

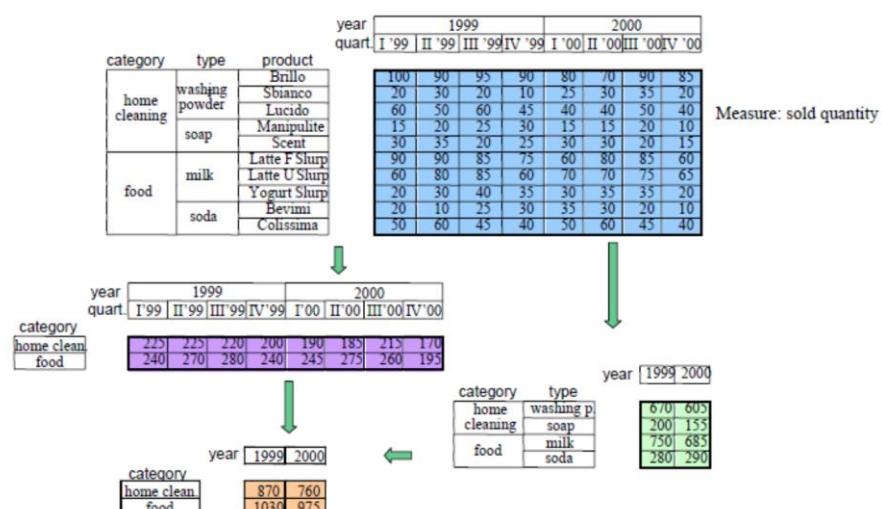
Per le misure non additive distinguiamo due casi:

Caso 1: operatore di aggregazione uguale per tutte le dimensioni.

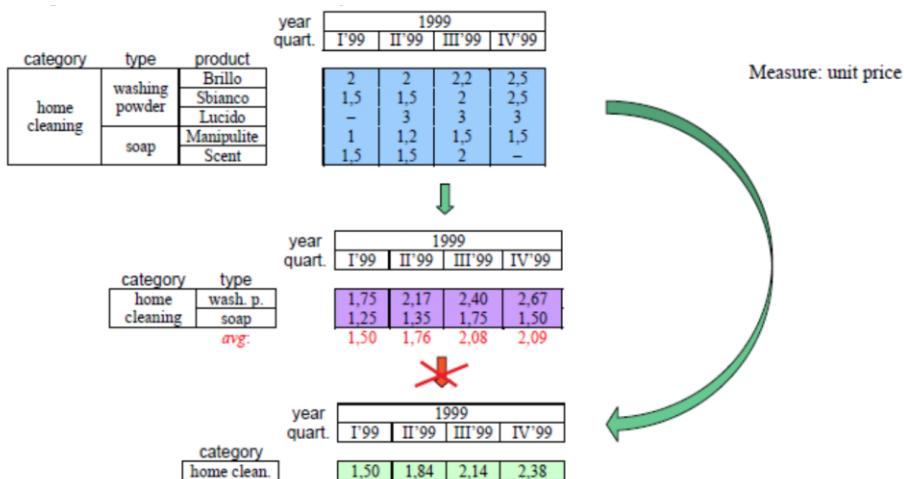
Gli operatori di aggregazione sono classificabili in:

- **Distributivi:** calcolo degli aggregati a partire da aggregati parziali (ES: SUM, MIN, MAX).
- **Algebrici:** utilizzo di misure di supporto per il calcolo di aggregati a partire da aggregati parziali (ES: AVG, deviazione standard, baricentro).
- **Olistici:** non permettono di calcolare aggregati a partire da aggregati parziali attraverso un numero finito di misure di supporto (ES: mediana, rango), per cui gli eventi secondari devono necessariamente essere calcolati a partire dagli eventi primari.

Operatore distributivo: SUM



Operatore algebrico: AVG



- Esempio della misura *prezzo unitario* di VENDITA, aggregabile tramite AVG su tutte le dimensioni: si nota immediatamente che la corretta aggregazione sul pattern **{categoria, trimestre}** non è ottenibile dall'aggregazione sul pattern **{tipo, trimestre}** a meno di aggiungere una nuova misura che conti il numero di eventi primari che definiscono ciascun evento secondario.

Aggregazione di misure non-additive (2)

Caso 2: operatore di aggregazione differenti lungo le diverse dimensioni.

- Eventi primari per uno schema dell'inventario con dimensioni magazzino e data sulla misura livello in riferimento al singolo prodotto.

città	magazzino	mese	marzo 1999								
		data	1/3/99	2/3/99	3/3/99	4/3/99	5/3/99	6/3/99	7/3/99	8/3/99	9/3/99
Roma	RM-Eur	10	10	8	4	20	20	15	15	12	
	RM-Centro	5	4	4	4	2	2	2	10	10	
	RM-Trastevere	14	14	14	12	20	20	20	20	16	
Milano	MI-Est	4	2	2	2	10	10	10	8	8	
	MI-Ovest	4	20	20	15	15	12	12	10	9	

- Eventi secondari sul pattern {mese, magazzino}

Aggregazione: min

mese marzo 1999

città	magazzino	
Roma	RM-Eur	4
	RM-Centro	2
	RM-Trastevere	12
Milano	MI-Est	2
	MI-Ovest	4

- Eventi secondari sul pattern {data, città}

Aggregazione: sum

mese	marzo 1999									
data	1/3/99	2/3/99	3/3/99	4/3/99	5/3/99	6/3/99	7/3/99	8/3/99	9/3/99	
città	Roma	29	28	26	20	42	42	37	45	38
	Milano	8	22	22	17	25	22	22	18	17

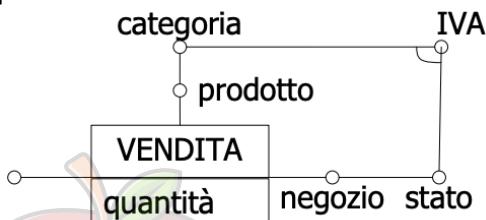
Aggregazione in presenza di convergenze e attributi cross-dimensional

Una convergenza nello schema di fatto è del tutto trasparente ai fini dell'aggregazione

Per verificare la semantica dell'aggregazione in presenza di attributi cross-dimensional dobbiamo risalire agli eventi primari che includono l'attributo cross-dimensionale

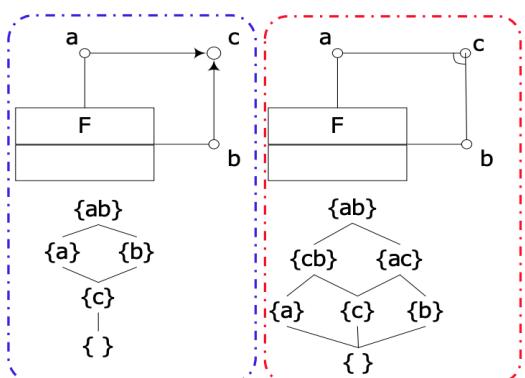
L'attributo cross-dimensionale IVA determinato da categoria e stato

- Ciascun evento primario è associato ad un prodotto e ad un negozio (quindi ad una categoria e ad un posto)
- Essendo definito univocamente un valore di IVA per ogni evento primario, gli eventi secondari sui pattern che includono IVA risultano immediatamente determinati



Aggregazione in presenza di convergenze e attributi cross-dimensional

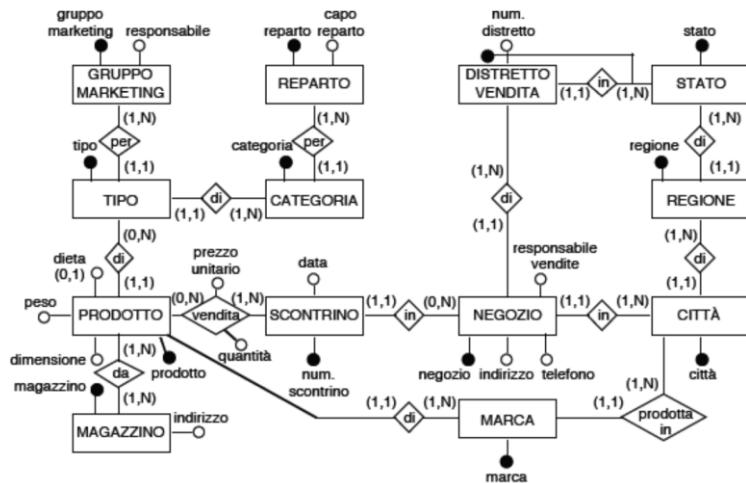
Differenza tra convergenza e attributo cross-dimensional:



Reticoli di roll-up in presenza di convergenza e cross-dimensionalità

Progettazione Concettuale

Esempio delle vendite (dopo la fase di integrazione)



Progettazione concettuale

Progettazione concettuale guidata dai dati

La tecnica per la progettazione concettuale di un Data mart a partire dalle sorgenti operazionali, secondo il DFM, consiste nei seguenti passi:

- 1) Definizione dei fatti;
- 2) Per ciascun fatto.
 - a. Costruzione dell'albero degli attributi
 - b. Potatura e innesto dell'albero degli attributi
 - c. Definizione delle dimensioni/misure.
 - d. Creazione dello schema di fatto

Definizione dei fatti

I fatti sono concetti di interesse primario per il processo decisionale.

In uno schema ER un fatto può corrispondere a:

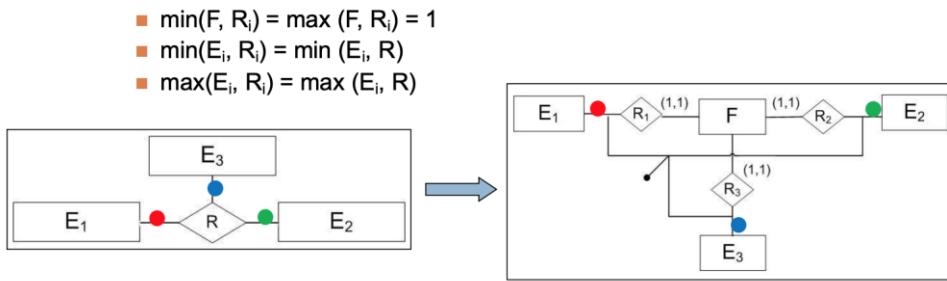
- **Un'entità F;**
- **Un'associazione n-aria R tra le entità E1, E2,..., En**

Processo di reificazione

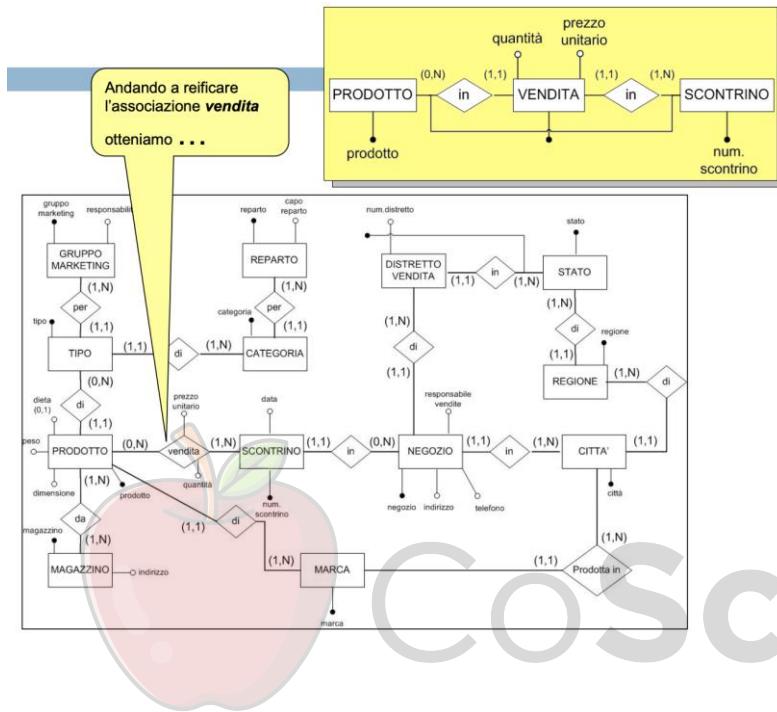
Se il fatto corrisponde ad **un'entità** del modello ER, non è richiesta alcuna modifica.

Se il fatto corrisponde ad una **relazione** n-aria R, è preferibile trasformare R in un'entità F che possieda tutti gli attributi della relazione e la cui chiave è data dall'unione delle chiavi delle singole entità coinvolte.

- Ciascuno dei rami di R viene sostituito con un'associazione binaria Ri tra F ed Ei tale che :



Esempio di reificazione



Linee guida nella scelta dei fatti

Le entità che rappresentano archivi aggiornati frequentemente (es. VENDITA) sono buoni candidati per la definizione dei fatti.

Le entità che rappresentano proprietà strutturali del dominio, corrispondenti ad archivi quasi statici (es. NEGOZIO e CITTA'), non lo sono.

In realtà, tale regola non è sempre valida in quanto la scelta del fatto dipende in maniera significativa sia dal dominio applicativo che dal tipo di analisi che l'utente intende eseguire.

Ciascun fatto identificato sullo schema sorgente diviene la radice di un differente schema di fatto.

Nel caso in cui diverse entità siano candidate ad esprimere lo stesso fatto, conviene sempre scegliere come fatto F l'entità, a partire dalla quale, è possibile costruire l'albero che include il maggior numero di attributi.

Costruzione albero degli attributi

Albero degli attributi: data un'entità F designata come fatto, si definisce albero degli attributi quello che soddisfa i seguenti requisiti:

- Ogni vertice corrisponde ad un attributo semplice o composto dello schema sorgente;
- La radice corrisponde all'identificativo di F.
- Per ogni vertice v, l'attributo corrispondente determina funzionalmente tutti gli attributi che corrispondono ai discendenti di v.

Algoritmo per la costruzione dell'albero degli attributi

```

root = nuovoVertice(ident(F)); /* ident(F) è l'identificatore di F, la radice
                               dell'albero è etichettata con l'identificatore
                               dell'entità scelta come fatto */
traduci(F,root);

procedura traduci(E,v);
//E è l'entità corrente dello schema sorgente, v il vertice corrente dell'albero
{
    per ogni attributo a di E tale che a!=ident(E)
        aggiungiFiglio(v, nuovoVertice(a))
        // aggiunge al vertice v un figlio a

    per ogni entità G connessa ad E da una associazione R tale che max(E,R)=1
    {
        per ogni attributo b di R
            aggiungiFiglio(v, nuovoVertice(b));
            // aggiunge al vertice v un figlio b
        prossimo = nuovoVertice(ident(G));
        //crea un nuovo vertice con il nome dell'identificatore di G...
        aggiungiFiglio(v, prossimo);
        // ... lo aggiunge a v come figlio ...
        traduci(G, prossimo);
        //... e innesca la ricorsione
    }
}

```

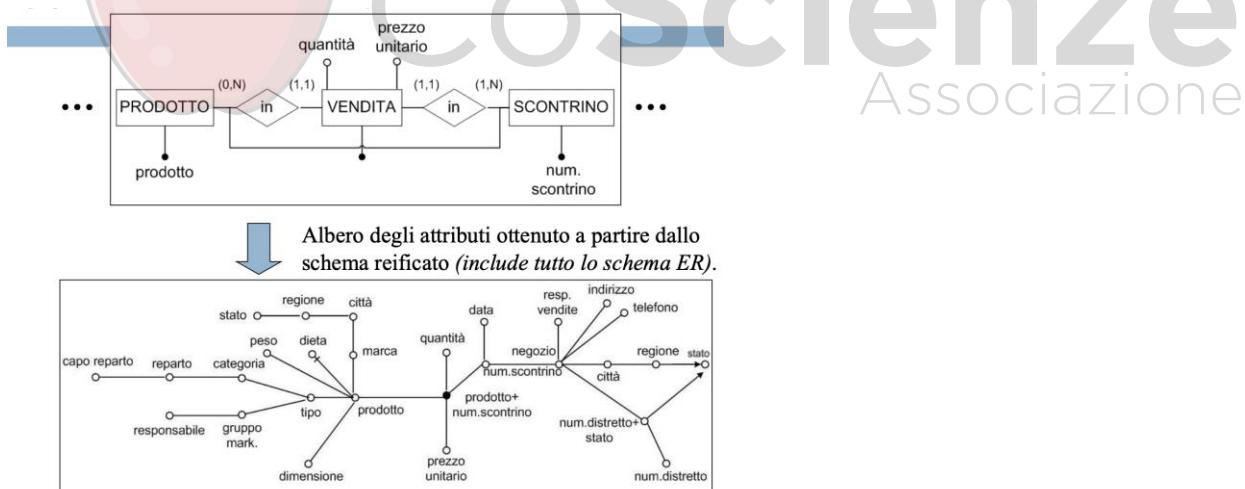
La procedura proposta naviga ricorsivamente le dipendenze funzionali espresse dagli identificatori e dalle associazioni ...-a-1 dello schema ER.

- L'entità a partire dalla quale viene innescato il processo è quella scelta come fatto.

Quando si esamina un'entità E si crea nell'albero un vertice v corrispondente all'identificatore di E, e gli si aggiunge un vertice per ogni altro attributo di E.

Per ogni associazione R da E verso un'entità G, con cardinalità massima 1, si aggiungono a v tanti figli quanti sono gli attributi di R, per poi ripetere il procedimento per G.

Applicazione dell'algoritmo

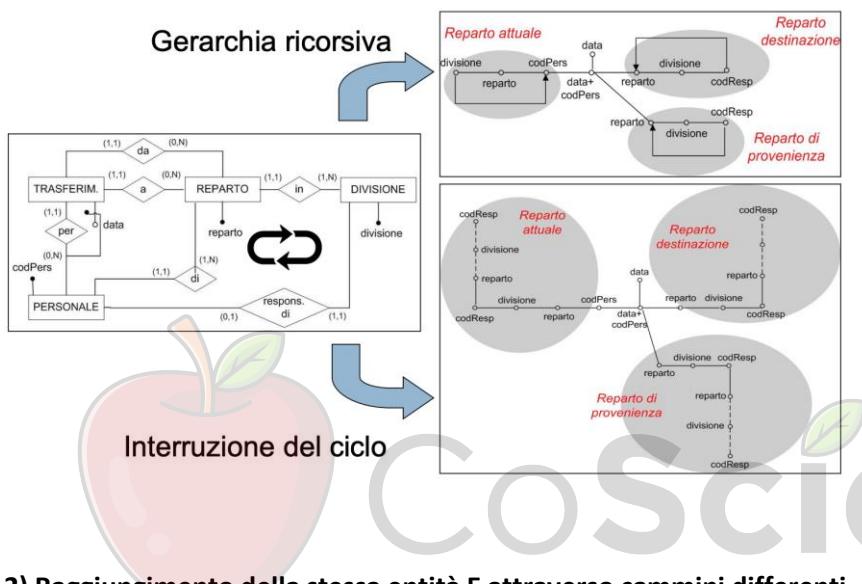


Algoritmo per la costruzione dell'albero degli attributi

Eccezioni: l'applicazione dell'algoritmo è condizionata da specifiche caratteristiche strutturali dello schema sorgente:

1. Presenza di un ciclo associazioni ... -a-1(loop);
2. Raggiungimento della medesima entità E attraverso cammini differenti.
3. Presenza di associazionia-molti e di attributi multipli.
4. Presenza di associazioni o attributi opzionali.
5. Presenza di associazioni n-arie.
6. Presenza di gerarchie di specializzazione.
7. Presenza di attributi composti.

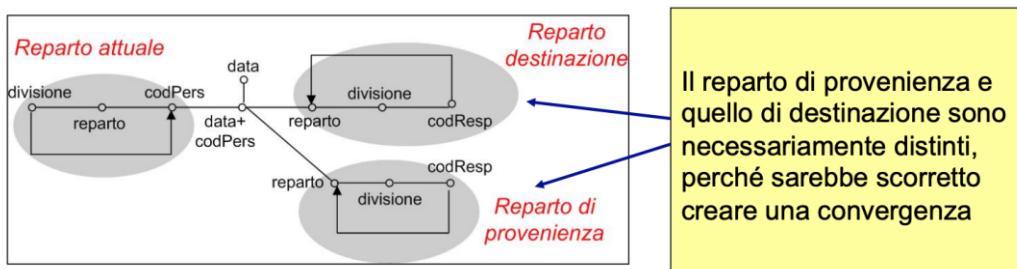
1) Presenza di un ciclo di associazioni molti a uno/ uno a uno

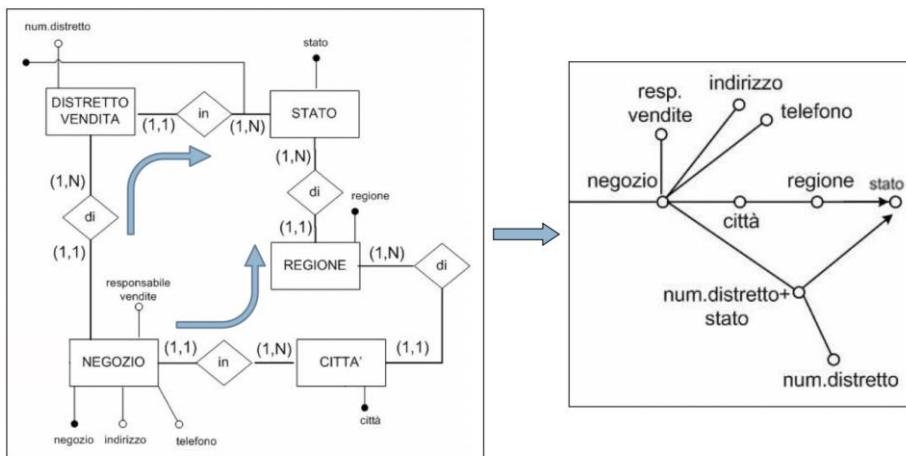


2) Raggiungimento della stessa entità E attraverso cammini differenti

Se si raggiunge due volte la stessa entità E attraverso cammini differenti, vengono generati nell'albero due vertici omologhi v' e v'' .

- Se ogni istanza di F determina E (i.e. $F \rightarrow E$) indipendentemente dal cammino seguito, allora v' e v'' possono coincidere (*si genera una convergenza*).

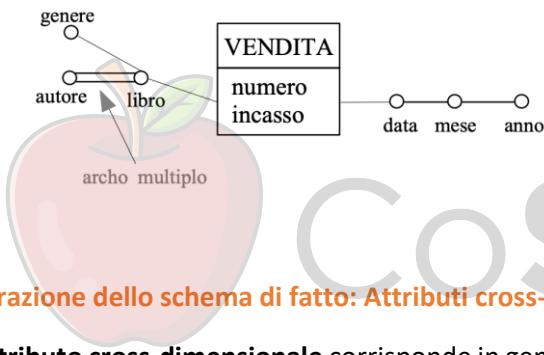




3) Presenza di associazioni ...-a-molti e di attributi multipli

Eventuali associazioni ...-a-molti ($\max(E,R) > 1$) e attributi multipli presenti nello schema ER non vengono inseriti automaticamente nell'albero degli attributi:

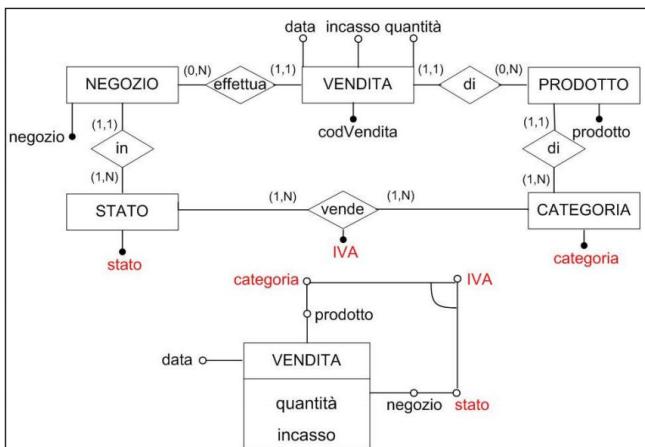
- Esse possono generare attributi cross-dimensional o archi multipli disegnati manualmente dal progettista sullo schema di fatto al termine della progettazione concettuale



Generazione dello schema di fatto: Attributi cross-dimensional

Un attributo cross-dimensionale corrisponde in genere a un attributo posto su un'associazione molti-a-molti R dello schema ER;

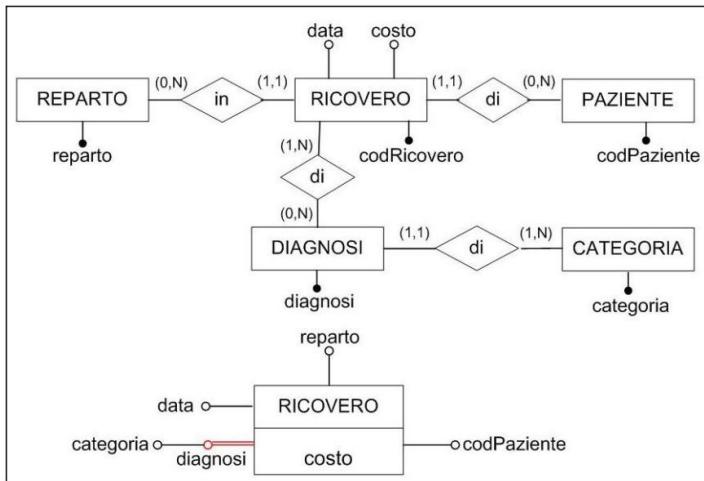
I suoi padri nello schema di fatto corrispondono allora agli identificatori delle entità coinvolte in R



Generazione dello schema di fatto: Archi multipli

Un arco multiplo corrisponde ad un'associazione R...-a-molti da un'entità E ad un'entità G.

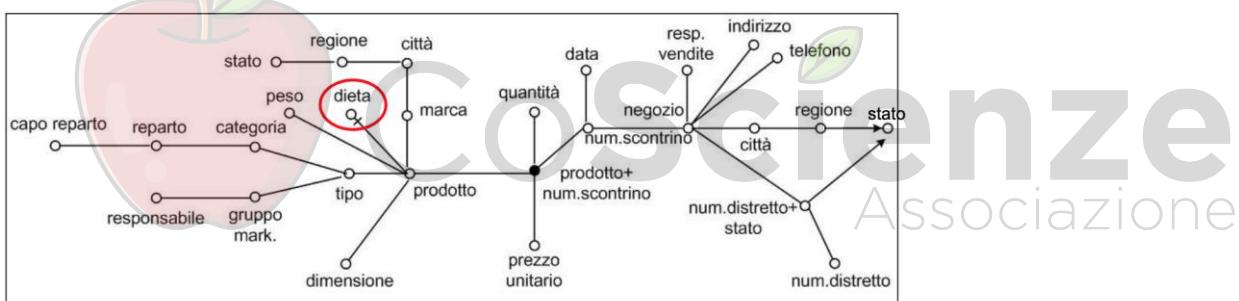
- Nello schema di fatto esso potrà connettere l'identificatore di E con un attributo di R o di G.



4) Presenza di associazioni o attributi opzionali

Gli **attributi opzionali** ($\min(E,R) = 0$) portano a collegamenti opzionali.

L'esistenza di un collegamento opzionale deve essere sottolineata nell'albero degli attributi con un trattino sugli archi corrispondenti ad associazioni o attributi opzionali nello schema ER:

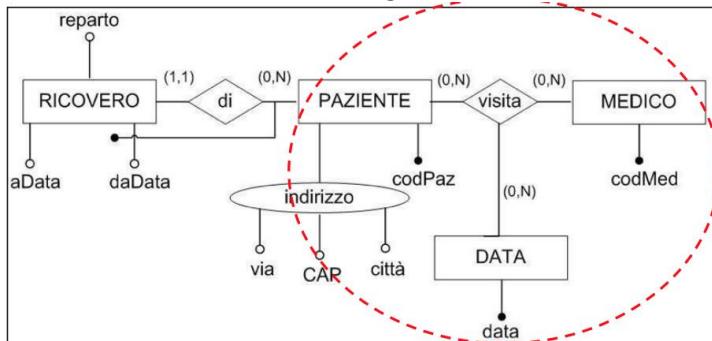


5) Presenza di associazioni n-arie

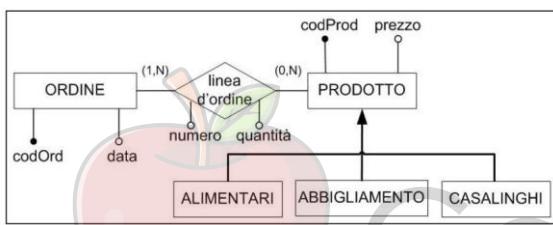
Eventuali associazioni n-arie saranno trasformate in n associazioni binare attraverso il processo di reificazione.

Molte delle associazioni n-arie hanno molteplicità massima maggiore di 1 sui rami (dunque sono inserite nell'algoritmo):

- Questa situazione porta ad n-associazioni binarie 1-a-molti che non possono essere inserite automaticamente nell'albero degli attributi.



6) Presenza di gerarchie di specializzazione

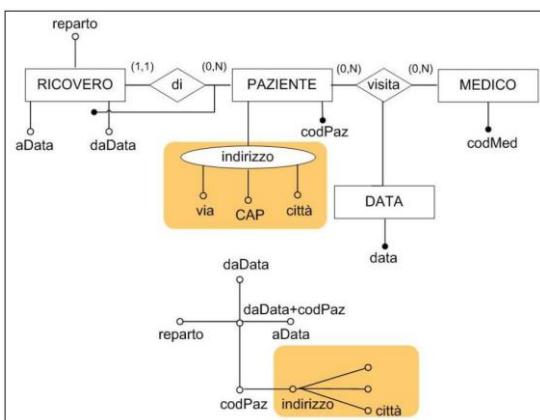


Le gerarchie dell'ER possono essere trattate dall'algoritmo come delle semplici **associazioni 1-a-1 opzionali** tra le super entità e la sotto entità.

In alternativa è possibile limitarsi ad aggiungere al noto corrispondente alla chiave della super entità un figlio che funga da discriminatore tra le diverse sotto entità (sotto entità unite con la super entità)

7) Presenza di attributi composti

In presenza di un attributo composto c, che consiste degli attributi semplici a₁,...,a_n tale attributo viene inserito nell'albero degli attributi come un vertice c con figli a₁,...,a_n.



Potatura e innesto dell'albero degli attributi

In genere, non tutti gli attributi dell'albero sono di interesse per il Data mart.

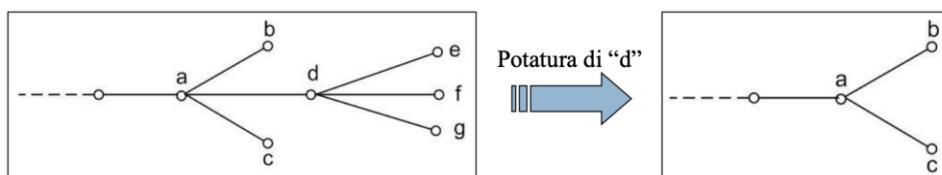
ES:

- Il numero di fax di un cliente difficilmente potrà rivelarsi utile ai fini decisionali per cui il Data mart è progettato.

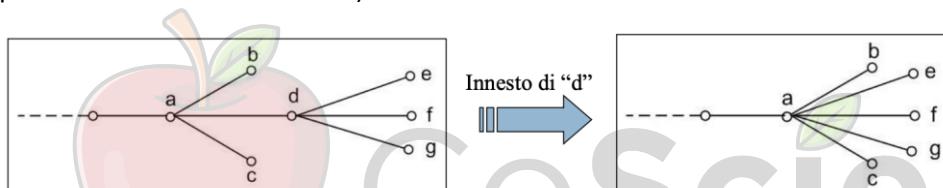
È dunque possibile manipolare l'albero al fine di eliminare e/o aggiungere livelli di dettaglio

Potatura di un vertice v: si effettua eliminando l'intero sottoalbero radicato in v.

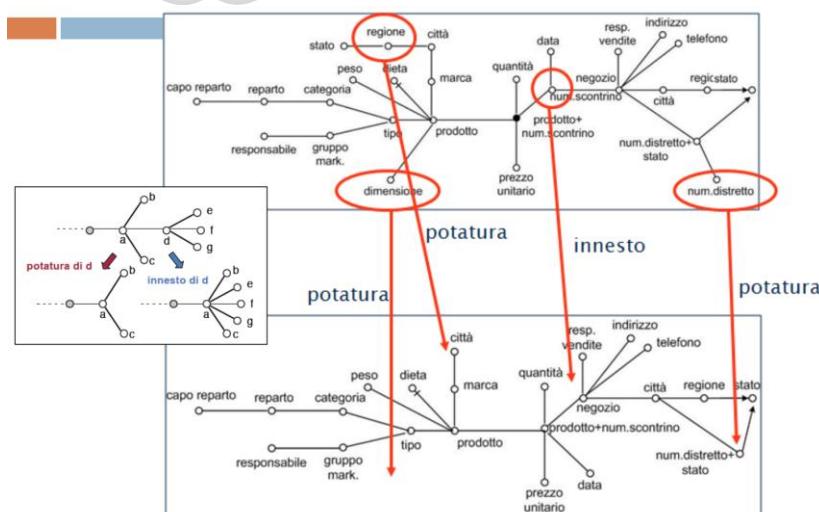
- Gli attributi eliminati non verranno inclusi nello schema di fatto.
- Non potranno essere usati per aggregare dati.



Innesto di un vertice v: viene utilizzato quando, sebbene un vertice esprima un'informazione non interessante, è necessario mantenere nell'albero i suoi discendenti (che verranno collegati direttamente al padre del vertice da innestare):



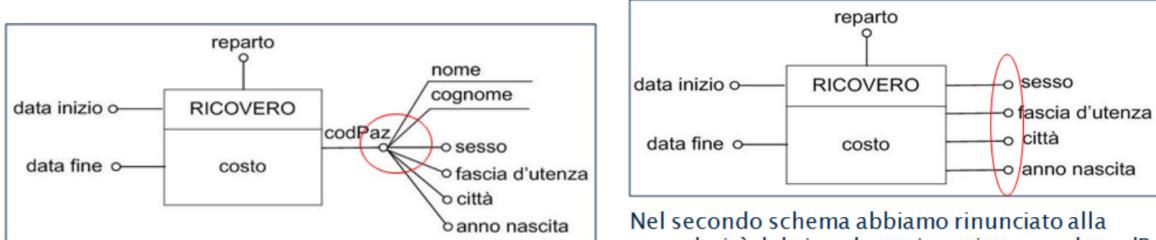
Esempi di potatura ed innesto



Definizione delle dimensioni

Le dimensioni devono essere scelte nell'albero degli attributi tra i vertici figli della radice; possono corrispondere ad attributi discreti o a intervalli di valori di attributi discreti o continui.

- ES. in un DW in ambito sanitario, un classico problema riguarda il mantenimento o meno della granularità del paziente.



Nel primo schema la dimensione codPaz è mantenuta (*grana transazionale*)

Nel secondo schema abbiamo rinunciato alla granularità del singolo paziente innestando codPaz e introducendo le dimensioni sesso, fascia d'utenza, città e anno di nascita (**grana temporale**)

In un DW non siamo interessati, di solito, ad interrogazioni di natura operazionale (che sono prerogativa dei DB relazionali).

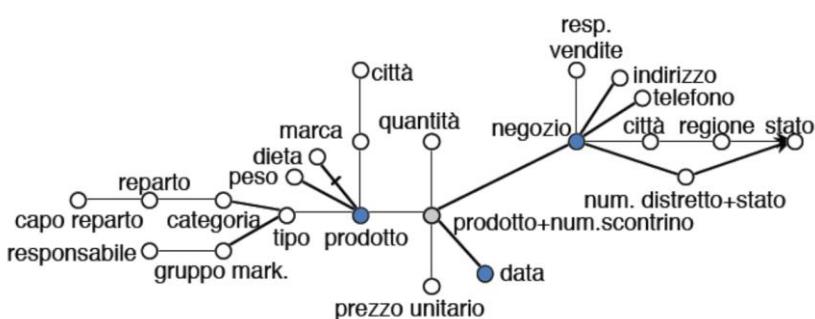
- Si preferisce in generale una grana temporale.

Tuttavia, se è necessario mantenere una granularità massima, si procede nel seguente modo:

- Duplicare il vertice radice nell’albero degli attributi; il nuovo vertice sarà collegato alla vecchia radice tramite un’associazione 1-a-1 e la radice non avrà altri archi uscenti.
 - Scegliere le dimensioni:
 - 1) Marcare come dimensione l’unico figlio diretto della radice (schema di fatto monodimensionale), oppure
 - 2) Trasformare in figli diretti della radice alcuni attributi dell’albero (schema di fatto multidimensionale)

L'esempio delle vendite

Definizione delle dimensioni:



Il tempo

Il tempo è un fattore chiave nella progettazione di un DW.

Gli schemi sorgente possono essere classificati rispetto al tempo come:

- **Snapshot:** descrivono lo stato corrente del dominio applicativo; vengono mantenute solo le versioni dei dati correnti che rimpiazzano continuamente le precedenti. Il tempo viene aggiunto manualmente come dimensione nello schema di fatto.

- **Storici:** descrivono l'evoluzione del dominio applicativo durante un intervallo di tempo; anche le vecchie versioni dei dati continuano ad essere mantenute. Pertanto, in tal caso il tempo diventa un ovvio candidato alla definizione di una dimensione nello schema di fatto.

Valid Time e Transaction Time

- **Valid time:** istante in cui l'evento si verifica nel mondo aziendale.
- **Transaction time:** istante in cui l'evento è memorizzato nel database.

Uso di valid e transaction time

Non necessariamente entrambe le coordinate temporali (valid e transaction) devono essere mantenute.

La scelta di quale debba essere mantenuta dipende dal tipo di interrogazioni, che possono essere:

- 1) Interrogazioni che richiedono il tempo di validità;
ES: in quali mesi gli studenti preferiscono iscriversi ad un certo corso;
- 2) Interrogazioni che richiedono il tempo di transazione.
ES: confrontare il numero totale degli iscritti con quello degli anni precedenti.
- 3) Interrogazioni che richiedono entrambi i tempi.
ES: stabilire qual è il ritardo medio nella trasmissione di pagamenti.

Modellazione del tempo nel DFM

A seconda del tipo di interrogazione il tempo viene modellato concettualmente in maniera diversa:

- 1) **Modellazione del solo tempo di validità:** soluzione che riflette la semantica del tempo comunemente adottata negli schemi di fatto.
Consente solo interrogazioni del primo tipo non essendo disponibili, prima dell'aggiornamento retrospettivo, i valori che riflettono la situazione reale.
- 2) **Modellazione del solo tempo di transazione:** soluzione sconsigliata se non per i casi in cui il tempo di transazione ha una semantica rilevante all'interno del dominio applicativo.
- 3) **Modellazione di entrambi i tempi:** è la soluzione più generale e consente la formulazione di tutti e tre i tipi di interrogazione.

Definizione delle misure

Se tra le dimensioni compaiono tutti gli attributi che costituiscono un'entità fatto (schema a grana transazionale), allora le misure corrispondono ad attributi numerici che siano figli della radice.

Se lo schema è a grana temporale, le misure devono essere definite applicando, ad attributi numerici dell'albero, funzioni di aggregazione (SUM, AVG, MAX, MIN) che operano su tutte le istanze di F corrispondenti a ciascun evento primario:

- Scegliere come misura un attributo che non è figlio diretto della radice significa rinunciare ad una dipendenza funzionale.

Glossari per il calcolo delle misure

Occorre definire un glossario che associa ciascuna misura ed un'espressione che descrive come essa possa essere calcolata.

Esempio: Il glossario delle vendite potrebbe essere il seguente:

quantità venduta = $\text{SUM}(\text{VENDITA.quantità})$

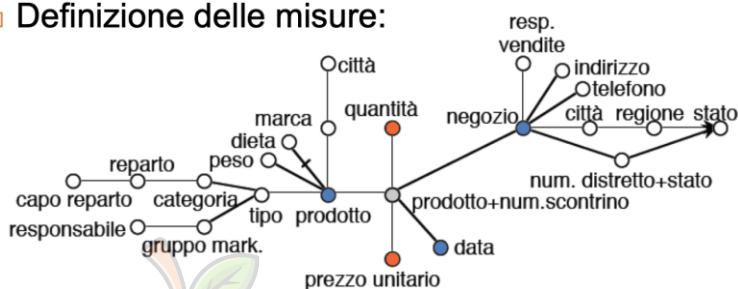
incasso = $\text{SUM}(\text{VENDITA.quantità} * \text{VENDITA.prezzoUnitario})$

num.clienti = $\text{COUNT} (*)$

Se la granularità del fatto è differente da quella dello schema sorgente, può essere utile definire più misure che aggregano lo stesso attributo tramite operatori diversi.

L'esempio delle vendite

Definizione delle misure:



GLOSSARIO

quantità venduta = $\text{SUM}(\text{VENDITA.quantità})$

incasso = $\text{SUM}(\text{VENDITA.quantità} * \text{VENDITA.prezzoUnitario})$ = prezzo unitario = $\text{AVG}(\text{VENDITA.prezzoUnitario})$
 num.clienti = $\text{COUNT} (*)$

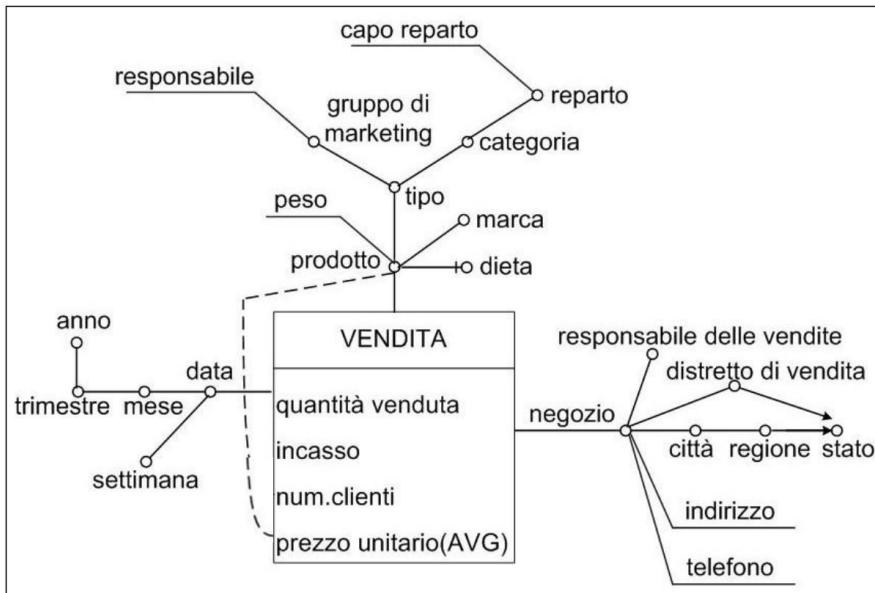
Creazione dello schema di fatto

L'albero degli attributi può ora essere tradotto in uno schema di fatto che include le dimensioni e le misure definite:

- Le gerarchie corrispondono ai sottoalberi dell'albero degli attributi con radice nelle diverse dimensioni.
- Il nome del fatto corrisponde al nome dell'entità scelta come fatto.
- È possibile potare e innestare l'albero per eliminare dettagli inutili.
- È possibile aggiungere attributi dimensionali definendo opportuni intervalli per attributi numerici (es. sulla dimensione tempo)
- Gli attributi che non verranno usati per l'aggregazione possono essere contrassegnati come descrittivi;
 - o Tra questi compariranno in genere anche gli attributi determinati da associazioni 1-a-1 e privi di discendenti.
- Per quanto riguarda eventuali **attributi alfanumerici** figli della radice ma non prescelti né come dimensioni né come misure:
 - o Se la granularità degli eventi primari coincide con quella entità F, essi possono essere rappresentati come attributi descrittivi associati direttamente al fatto, di cui descriveranno ciascuna occorrenza;
 - o Se invece le due granularità sono differenti, essi devono necessariamente essere potati.

Esempi di generazione finale dello schema di fatto

Si ottiene il seguente schema di fatto:



Modellazione Logica

Esistono due distinti modelli logici per rappresentare la struttura multidimensionale dei dati:

- Quello relazionale, che dà luogo ai sistemi **ROLAP** (Relation On-Line Analytical Processing);
- Quello multidimensionale, che dà luogo ai sistemi **MOLAP** (Multidimensional On-Line Analytical Processing)

La maggior parte del mercato è orientata ai sistemi ROLAP, a causa di un insieme di problemi relativi al sistema MOLAP.

Sistemi MOLAP: vantaggi

I sistemi MOLAP memorizzano i dati usando strutture dati multidimensionali, ad esempio vettori multidimensionali in cui ogni elemento è associato ad un insieme di coordinate nello spazio dei valori.

Vantaggi:

- Il tipo di struttura dati utilizzata è la rappresentazione più naturale per i dati di un DW;
- Fornisce ottime prestazioni poiché si presta bene alle esecuzioni delle operazioni OLAP, che sono esprimibili direttamente sulla struttura dati e non hanno bisogno di essere simulate attraverso interrogazioni SQL.

Svantaggi:

- Sparsità dei dati;
- Mancanza di standard, i diversi sistemi hanno in comune solo principi di base (ES. strutture dati), ma non ci conoscono i dettagli implementativi;
- Non esistono standard di interrogazione che svolgono un ruolo simile a quello di SQL nei sistemi relazionali.

Sistemi MOLAP: Problema della sparsità

- **Causa:** solo una piccola porzione delle celle di un cubo contiene effettivamente informazioni, le rimanenti corrispondono ad eventi non accaduti. Questa comporta un forte spreco in termini di spazio su disco e di tempo per caricare i dati nel cubo.
- Questo problema non incide sui sistemi ROLAP, poiché essi consentono di memorizzare solo le celle di interesse.

I sistemi ROLAP

Utilizzano il modello relazionale per la rappresentazione dei dati multidimensionali;

I motivi che spingono all'adozione di un modello bidimensionale per modellare concetti multidimensionali sono i seguenti:

- Il modello relazionale è lo standard “*de facto*” dei database, ed è conosciuto dai professionisti del settore.
- L’evoluzione subita dai DBMS relazionali, da trent’anni sul mercato, li rende strumenti raffinati ed ottimizzati.
- L’assenza di sparsità dei dati garantisce maggiore scalabilità, fondamentale per database in continua crescita quali i DW.

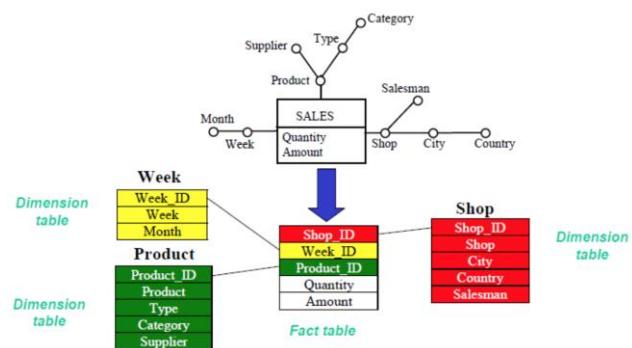
I sistemi ROLAP: Schema a stella

Si usa per la modellazione multidimensionale su sistemi ROLAP.

È composto da:

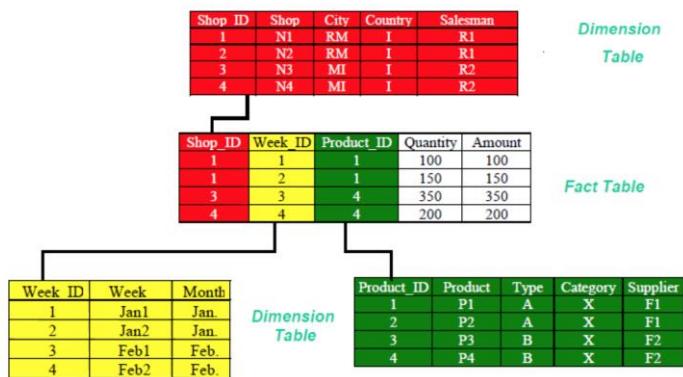
- Un insieme di relazioni DT1... DTn, chiamate **dimension table**, ciascuna associata ad una dimensione e caratterizzata da una chiave primaria di ed un insieme di attributi che descrivono le dimensioni a vari livelli di aggregazione.
- Una relazione FT, chiamata **fact table**, la cui chiave primaria è data dall’insieme delle chiavi primarie delle dimension table.
- Inoltre, FT contiene un attributo per ogni misura.

Esempio Schema a Stella



Schema a stella per il fatto delle vendite. La chiave della fact table SALES è costituita dalla combinazione delle chiavi sterne sulle tre dimension table

Esempio 2



Schema a Stella: Osservazioni

La visione multidimensionale dei dati si ottiene eseguendo il **join** tra le *fact table* e le diverse dimension table.

Per il fatto delle vendite riportato in precedenza, l'interrogazione SQL che ricostruisce le celle associando i valori delle misure ai corrispondenti valori degli attributi presenti nelle gerarchie è:

SELECT * FROM VENDITE AS FT, PRODOTTO AS DT1,

 NEGOZIO AS DT2, DATA AS DT3

WHERE FT.chiaveP=DT1.chiaveP AND

FT.chiaveN=DT2.chiaveN AND

FT.chiaveD=DT3.chiaveD;

CoScienze
Associazione

Schema a Stella: Proprietà

Alla chiave di un dimension table si possono riferire più fact table, se le gerarchie sono conformi.

Le dimension table **non** sono in 3NF, a causa della presenza di dipendenze funzionali transitive generate dalla presenza contemporanea di tutti gli attributi della gerarchia.

La sparsità non rappresenta un problema, poiché nella fact table vengono memorizzate solo le combinazioni di chiavi per le quali effettivamente esiste l'informazione.

I sistemi ROLAP: lo schema snowflake

Uno schema snowflake è ottenibile da uno schema a stella decomponendo una o più **dimension table** DT_i in più tabelle DT_{i,1}, ..., DT_{i,n}, al fine di eliminare alcune delle dipendenze funzionali transitive presenti.

Ogni dimension table è caratterizzata da:

- Una chiave primaria **d(i,j)** (di solito surrogata);
- Un sottoinsieme degli attributi di DT_i che dipendono funzionalmente da **d(i,j)**;
- Zero o più chiavi esterne riferite ad altre DT_{t,k} necessarie a garantire la ricostruibilità del contenuto informativo di DT_i.

Le dimension table primarie sono quelle in cui le chiavi sono importate nella fact table, secondarie le altre.

Esempio di schema snowflake

Un possibile snowflake per lo schema a stella delle vendite prevede l'inserimento delle tabelle CITTA' (City) e CATEGORIA (Type), ottenendo una parziale normalizzazione dei dati contenuti nelle dimension table.

Vengono spezzate le dipendenze transitive tra negozio (Shop) e città (City), e tra *prodotto (Product)* e *categoria (Type)*:

- Lo spazio richiesto per la memorizzazione dei dati si riduce:
 - o ES: le corrispondenze tra valori degli attributi città e regione vengono memorizzate una sola volta;
- Il tempo di esecuzione delle interrogazioni che coinvolgono attributi delle dimension table secondarie aumenta poiché è necessario un maggior numero di join.

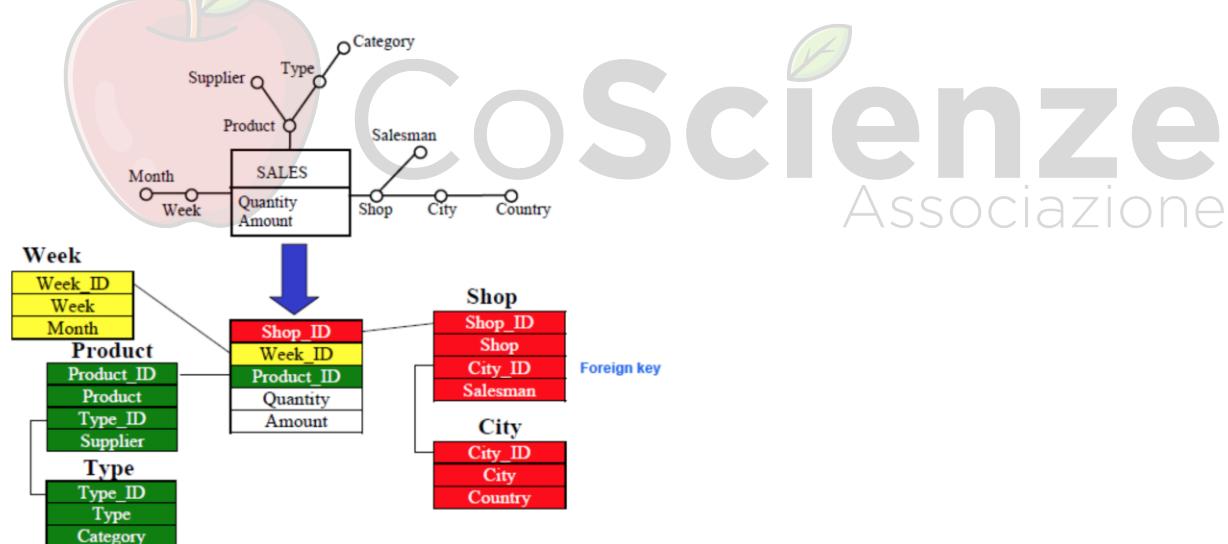
I sistemi ROLAP: vantaggi schema snowflake

È necessario inserire nuove chiavi surrogate per determinare le corrispondenze tra dimension table primarie e secondarie.

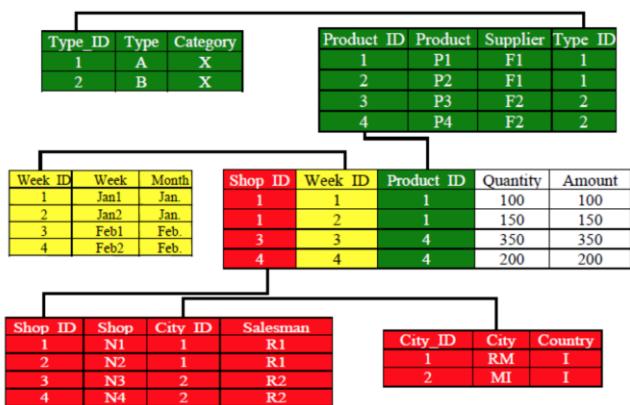
- ES: l'importanza di Type_ID nella tabella PRODOTTO permette di associare ad ogni prodotto il relativo tipo;

L'esecuzione di interrogazioni che coinvolgono solo gli attributi contenuti nella fact table e nelle dimension table primarie è avvantaggiata poiché i join coinvolgono tabelle di dimensioni inferiori.

Esempio schema Snowflake



Esempio (2)



Star o Snowflake?

Lo schema snowflake normalmente non è raccomandato:

- La diminuzione dello spazio di memorizzazione raramente è benefico:
 - o Maggiore spazio è consumato dalla fact table;
- Il costo della esecuzione del join potrebbe essere significativo;

Lo schema snowflake è utile:

- Quando parte di una gerarchia è condivisa tra le dimensioni (ES: gerarchie geografiche)
- Per le viste materializzate che richiedono una rappresentazione aggregata delle dimensioni corrispondenti.

Le viste

Problema: Analisi utente rese difficili dalla quantità di dati memorizzati nel DW.

Soluzione: Ridurre la porzione da esaminare attraverso operazioni di:

- Selezione: restringono la porzione di dati di interesse individuano quelli effettivamente interessanti per la specifica analisi.
- Aggregazione: riducono i dati collassando più elementi non aggregati in un unico elemento aggregato.

Le viste (2)

L'aumento delle prestazioni è ottenuto precalcolando i dati aggregati di uso comune.

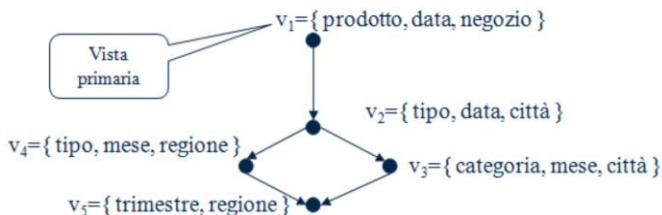
Le fact table contenenti dati aggregati sono dette **viste**:

- **Viste primarie:** corrispondono a pattern primari (definito dall'insieme delle dimensioni);
- **Viste secondarie:** corrispondono a pattern secondari o sono individuate verificando se possono essere alimentate a partire da viste nel DW (aggregati)

Le viste (3)

Sono rappresentate alcune viste materializzabili per lo schema a stella delle vendite:

- Una freccia da v_i a v_j indica che $P_j \leq P_i$ essendo P_i e P_j rispettivamente i pattern di v_i e v_j . Quindi i dati contenuti in v_j possono essere calcolati aggregando quelli di v_i .
- Un'interrogazione relativa alle vendite che richieda i dati aggregati per il tipo del prodotto, data di vendita e città (i.e., {tipo, data, città}) in cui la vendetta è stata effettuata risulterà meno costosa se eseguita su v_2 poiché essa insisterà su una fact table piccola e non richiederà ulteriori operazioni di aggregazione.



Le viste: Problemi

Problema: calcolando l'incasso delle vendite a partire dalla tabella aggregata si ottiene un dato diverso da quello ottenuto a partire dalla tabella non aggregata.

Causa: Applicando l'operatore di media si perde l'informazione relativa ai singoli prezzi praticati.

Soluzione: Memorizzare nella tabella aggregata anche i valori degli incassi.

tipo	prodotto	quantità	prezzo	incasso
latticino	Latte Slurp	5	1,0	5,0
latticino	Latte Gnam	7	1,5	10,5
bibita	Colissima	9	0,8	7,2
		totale:	22,7	

Tabella delle vendite di diversi prodotti

tipo	quantità	prezzo	Quantità x prezzo
latticino	12	1,25	15,0
bibita	9	0,8	7,2
		totale:	22,2

Tabella dei dati aggregati in base ai tipi di prodotti

Progettazione Logica

La progettazione logica definisce l'insieme dei passi per trasformare lo schema concettuale in uno schema logico.

La progettazione logica dei Data mart è profondamente diversa dai sistemi operazionali:

- Nei DW l'obiettivo è quello di massimizzare la velocità del reperimento dei dati mentre nei sistemi operazionali si mira a minimizzare la quantità di informazione da memorizzare.

I principali passi di questo processo sono:

- **Traduzione degli schemi di fatto in schemi logici;**
- **Materializzazione delle viste;**
- **Frammentazione verticale ed orizzontale delle fact table.**

Traduzione degli schemi di fatto in schemi logici

Uno schema di fatto può essere modellato in ambito relazionale mediante uno schema a stella in cui la fact table contiene tutte le misure e gli attributi descrittivi e, per ogni gerarchia, viene creata una dimension table che ne contiene tutti gli attributi.

La traduzione dal DFM al modello logico non è del tutto automatica, richiedendo in alcuni momento l'intervento del progettista

- Una corretta traduzione di uno schema di fatto richiede una trattazione più approfondita per i costrutti avanzati del DFM.

Costrutti avanzati: Attributi descrittivi

Sappiamo che un attributo descrittivo contiene informazioni non utilizzabili per effettuare aggregazioni che si ritiene utile mantenere.

Durante la modellazione un attributo descrittivo:

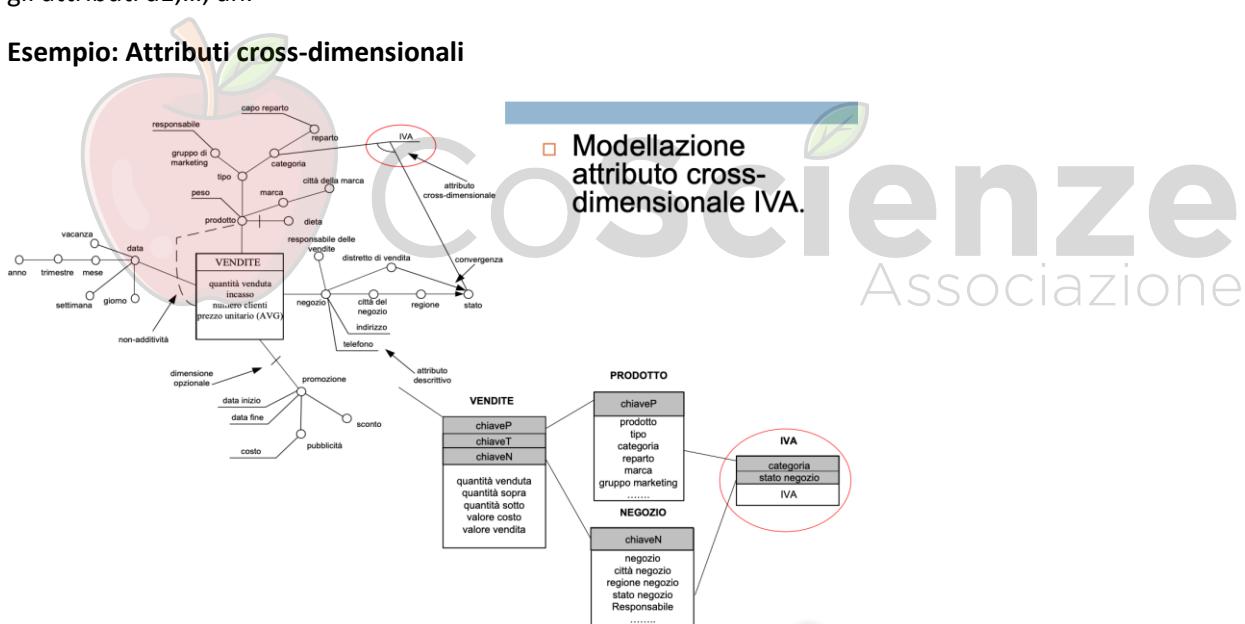
- Viene incluso nella dimension table relativa alla gerarchia che lo contiene se collegato ad un attributo dimensionale da cui dipende funzionalmente.
- Viene incluso nella fact table assieme alle misure collegate direttamente al fatto.

Costrutti avanzati: Attributi cross-dimensional

Sappiamo che un attributo cross-dimensional è tale se il suo valore è determinato dalla combinazione di due o più attributi dimensionali eventualmente appartenenti a gerarchie diverse.

Se un attributo cross-dimensional b definisce un'associazione multi-a-molti tra due o più attributi dimensionali a1,..., an, esso richiede l'inserimento di una nuova tabella che includa b ed abbia come chiave gli attributi a1,..., an.

Esempio: Attributi cross-dimensional



Costrutti avanzati: Gerarchie condivise

Sappiamo che una gerarchia condivisa è una porzione di gerarchia che viene ripetuta due o più volte;

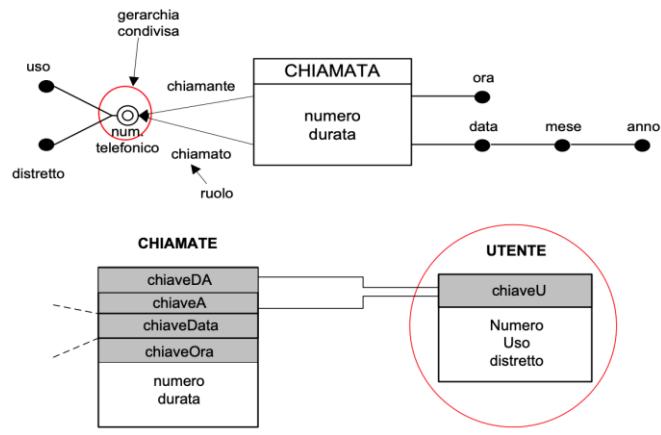
A livello logico non è consigliabile introdurre più dimension table che contengano gli stessi dati.

A livello progettuale esistono due soluzioni per due situazioni distinte:

- Se due gerarchie contengono esattamente gli stessi attributi è sufficiente importare due valori diversi dell'unica chiave della dimension table nella fact table.
- Se due gerarchie condividono una parte degli attributi si può scegliere di replicare le informazioni comuni o di introdurre una nuova dimension table comune ad entrambe le gerarchie.

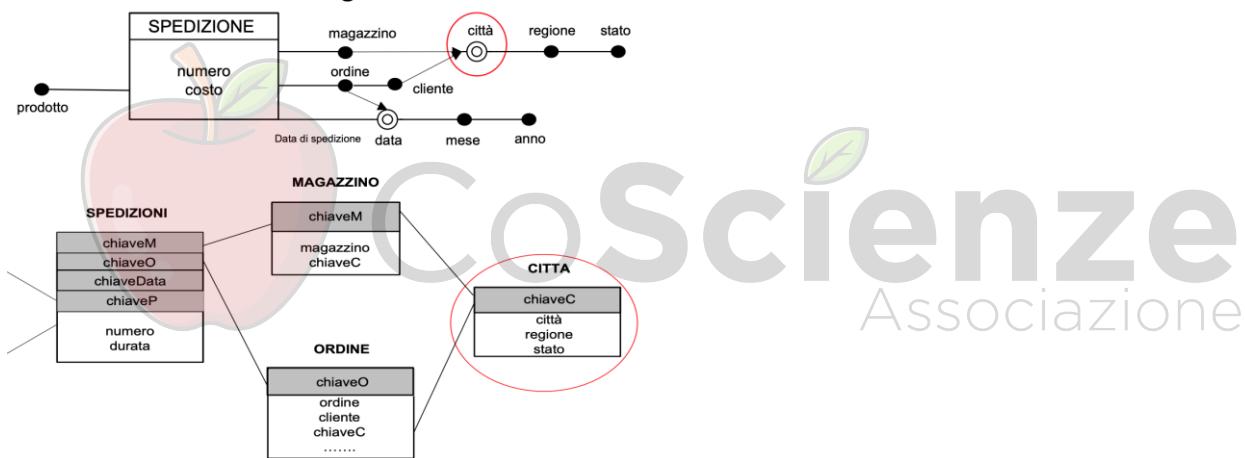
Esempio: Condivisione totale

Modellazione numero telefonico chiamante e chiamato.



Esempio: Condivisione parziale

Modellazione città del magazzino e del cliente



Costrutti avanzati: Archi multipli

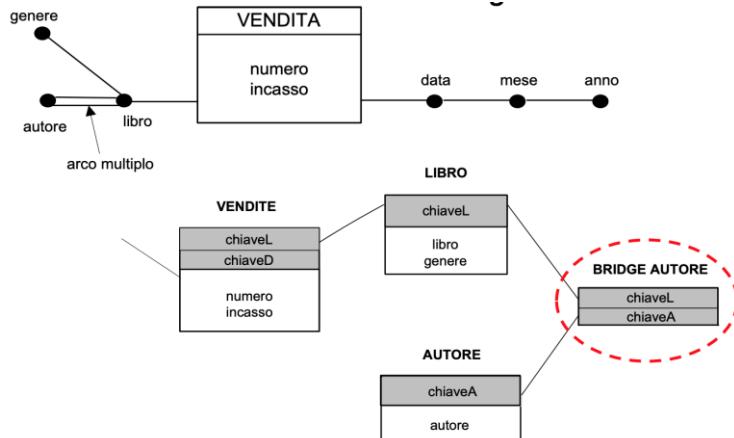
Ricordiamo che una gerarchia che codifica associazioni molti-a-molti viene modellata con archi multipli.

A livello logico esistono diverse soluzioni:

- Soluzione bridge table: utilizzo di una nuova tabella la cui chiave è composta dalla combinazione degli attributi collegati dall'arco multiplo (schema snowflake)
- Soluzione push-down: l'associazione molti-a-molti viene modellata direttamente all'interno della fact table. Viene poi aggiunta una nuova dimensione corrispondente all'attributo terminale a dell'arco multiplo, ed eventuali figli di a verranno memorizzati nella nuova dimension table (schema a stella)

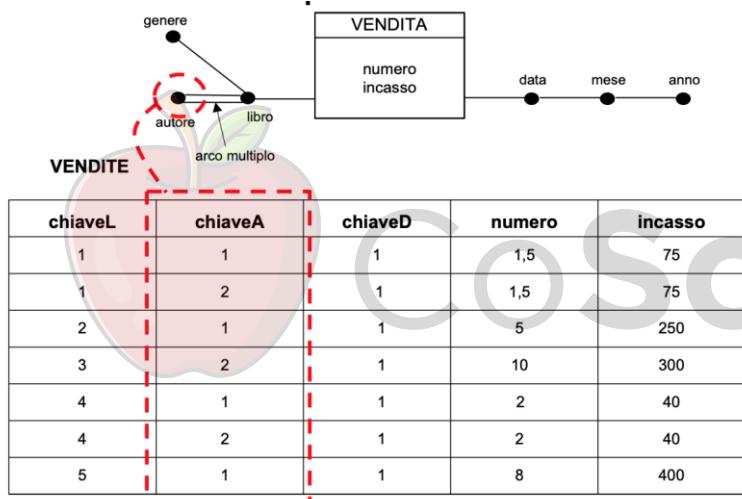
Esempio Bridge table

Modellazioni di autore tramite bridge table.



Esempio: Push-down

Una possibile istanza della fact table Vendite avendo effettuato il push-down di autore.



Bridge table vs. push-down

La soluzione push-down introduce una forte ridondanza nella fact table le cui righe devono essere replicate tante volte quante sono le corrispondenze dell'arco multiplo.

- La forte ridondanza causa operazioni di aggiornamento molto costose che non si verificano nella soluzione bridge table.

Operazioni di interrogazione nelle soluzioni push-down prevedono un singolo join mentre con bridge table risultano più complesse.

Il calcolo degli eventi primari avviene durante l'alimentazione nella soluzione push-down mentre nella soluzione con bridge table avviene durante l'interrogazione.

Costrutti avanzati: Archi opzionali

Ricordiamo che un arco opzionale si riferisce ad una gerarchia opzionale, in cui un'associazione dello schema di fatto non è definita per un sottoinsieme di eventi.

La presenza di archi opzionali non incide sulla struttura della corrispondente dimension table:

- L'attributo continua a comparire anche se per alcune istanze non risulterà valorizzato;
- Per le istanze in cui non è definito tale valore si introduce un valore fittizio.

L'opzionalità non può essere gestita direttamente nella fact table introducendo un valore fittizio per la chiave.

- Bisognerà introdurre un'intera tupla fittizia all'interno della dimension table.

Costrutti avanzati: Gerarchie ricorsive

Ricordiamo: una gerarchia ricorsiva è una gerarchia in cui le relazioni padre-figlio tra i livelli sono consistenti ma possono avere istanze di lunghezza differenti.

La modellazione delle gerarchie ricorsive può essere effettuata in due modi:

- Nelle dimension table la ricorsione è modellata con un auto anello che rappresenta un numero variabile, e potenzialmente illimitato, di livelli. Tale modellazione non è supportata dalla maggior parte dei DBMS commerciali.
- Con una **tabella di navigazione** che modella un'associazione molti a molti tra le fact table e la dimension table. La dimensione di tale tabella cresce in maniera esponenziale rispetto alla profondità della gerarchia ma tale modellazione ha un maggiore potere espressivo in fase di interrogazione.

Lo schema logico relazionale

Schema logico relazionale di vendita:

PRODOTTO (<u>prodotto</u> , peso, dieta, marca: MARCA, tipo: TIPO)
MARCA (<u>marca</u> , prodottoIn:CITTÀ)
CITTÀ (città, regione:REGIONE)
REGIONE (<u>regione</u> , stato:STATO)
STATO (<u>stato</u>)
TIPO (ting, gruppoMarketing:GRUPPOMARK, categoria:CATEGORIA)
GRUPPOMARK (<u>gruppoMarketing</u> , responsabile)
CATEGORIA (<u>categoria</u> , reparto:REPARTO)
REPARTO (<u>reparto</u> , capoReparto)
IVA (<u>categoria</u> :CATEGORIA, <u>stato</u> :STATO, iva)
NEGOZIO (<u>negozio</u> , indirizzo, telefono, respVendite, (numDistr, stato):DISTRETTO, inCittà:CITTÀ)
DISTRETTO (<u>numDistr</u> , <u>stato</u> :STATO)
DATA (<u>data</u> , giorno, vacanza, settimana, mese:MESE)
MESE (<u>mese</u> , trimestre:TRIMESTRE)
TRIMESTRE (<u>trimestre</u> , anno:ANNO)
ANNO (<u>anno</u>)
PROMOZIONE (<u>promozione</u> , dataInizio, dataFine, sconto, pubblicità:PUBBLICITA)
PUBBLICITA (<u>pubblicità</u> , costo)
VENDITA (<u>prodotto</u> :PRODOTTO, <u>negozio</u> :NEGOZIO, <u>data</u> :DATA, <u>promozione</u> :PROMOZIONE, <u>quantità</u> , <u>prezzoUnitario</u>)

Gli attributi facenti parte di chiavi esterne composte sono indicati tra parentesi

Materializzazione delle viste

Con il termine materializzazione delle viste si intende il processo di selezione di un insieme di viste secondarie ottenute a partire dai dati contenuti nelle viste primarie.

La scelta delle viste da materializzare deve essere fatta sulla base di un insieme di obiettivi di progetto.

Due sono gli elementi principali nel processo di materializzazione:

- La definizione degli obiettivi della materializzazione, che possono essere funzioni di minimizzazione di costo o vincoli.
- La tecnica di selezione da utilizzare.

Funzioni di costo da minimizzare

Tipicamente le funzioni di costo che possono essere minimizzate sono:

- **Costo del carico di lavoro:** rappresenta il costo totale del carico di lavoro che può essere calcolato come somma pesata del costo delle diverse interrogazioni, dove il peso di ogni singola interrogazione può essere la frequenza e/o la sua importanza per l'utente.
- **Costo di manutenzione delle viste:** rappresenta il costo delle interrogazioni necessarie a propagare gli aggiornamenti delle sorgenti operazionali alle viste.

Vincoli

Vincoli di sistema: sono dettati dalla limitatezza delle risorse disponibili e riguardano:

- **Spazio di memorizzazione:** per calcolare questo vincolo è necessaria una funzione per la stima della dimensione delle viste.
- **Tempo di aggiornamento:** il tempo di aggiornamento delle viste

Vincoli utente: sono legati a particolari requisiti espressi dagli utilizzatori del sistema:

- **Tempo di risposta alle interrogazioni:** tempo di massimo di risposta richiesto dall'utente per le diverse interrogazioni.
- **Data di aggiornamento delle risposte:** limite massimo imposto dall'utente per il tempo intercorso dall'ultimo aggiornamento di una vista impiegata per l'esecuzione di un'interrogazione.

Tecniche di selezione

Le tecniche di selezione operano su due diverse fasi:

- Tra tutte le possibili viste materializzabili, vengono individuate quelle effettivamente utili per il carico di lavoro.
- Successivamente, tramite tecniche euristiche, se ne determina un sottoinsieme che minimizza la funzione di costo nel rispetto dei vincoli di sistema.

Reticolo multidimensionale

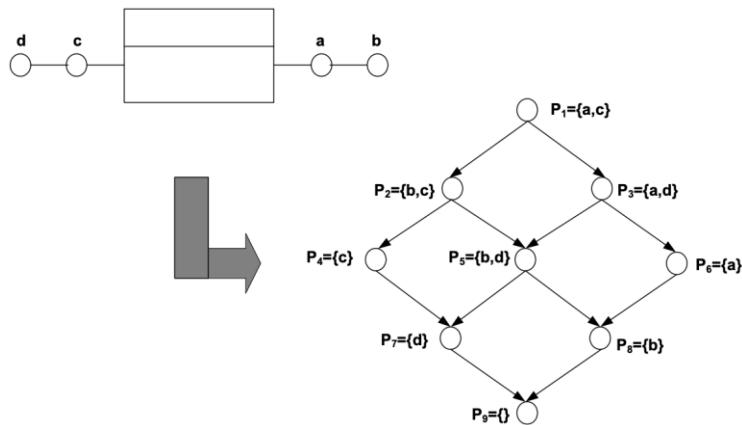
Il **reticolo multidimensionale** o MD-lattice viene utilizzato per individuare tutte le possibili viste materializzabili a partire da uno schema di fatto, definendo tutti i possibili pattern di aggregazione validi.

Un pattern è valido se non esistono dipendenze funzionali tra i suoi elementi.

Nota. Un arco del reticolo da un pattern P ad un pattern Pj indica che Pj è meno fine di Pi ($Pj \leq Pi$)

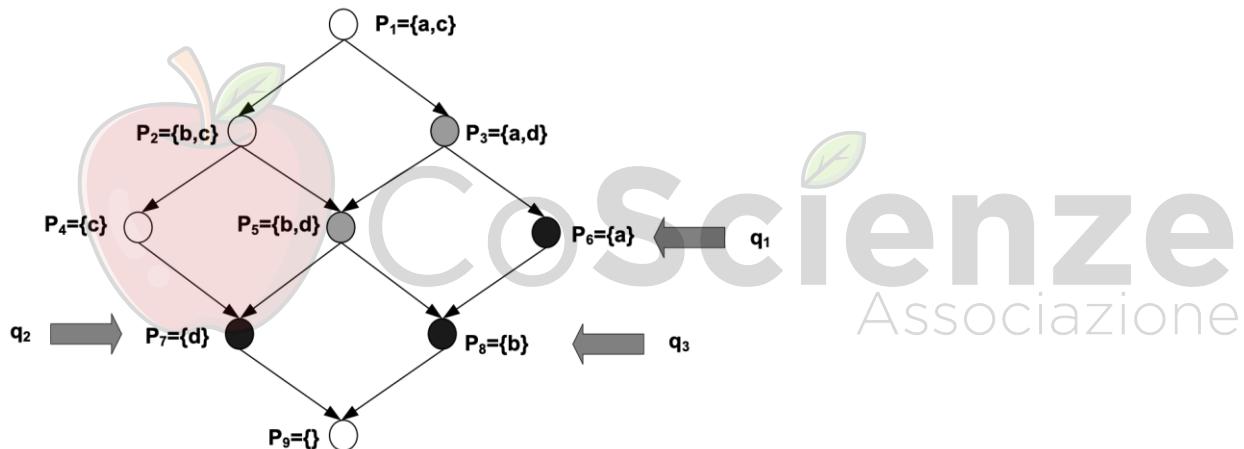
Esempio: Reticolo multidimensionale

Reticolo corrispondente al cubo multidimensionale con dimensioni a, c e gerarchie a->b e c->d



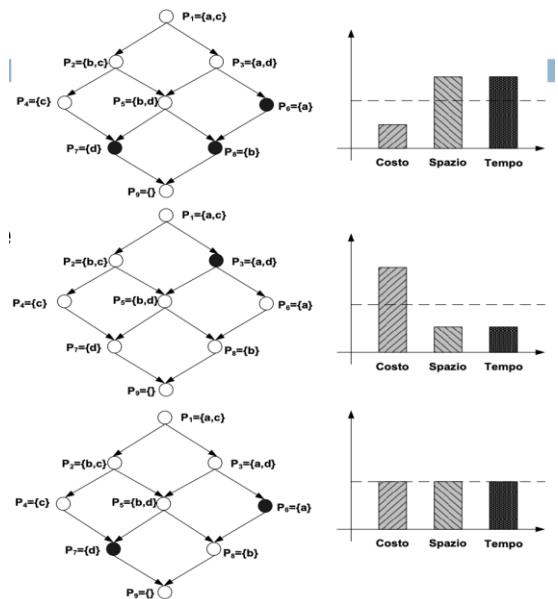
Esempio: Viste candidate

In grigio e nero sono evidenziate le viste per il carico di lavoro q1, q2, q3.



Esempio: Scelta delle viste da materializzare

Tre possibili soluzioni al problema della materializzazione delle viste



Linee guida per selezionare le viste

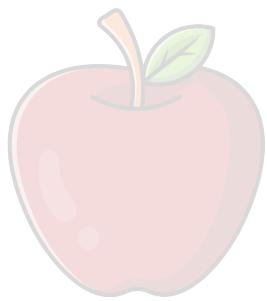
È consigliabile materializzare una vista quando:

- Risolve direttamente un'interrogazione molto frequente;
- Permette di risolvere molte interrogazioni;

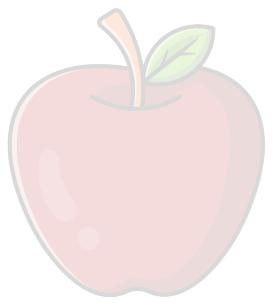
Non è consigliabile materializzare una vista quando:

- Il suo pattern è molto simile ad una vista già materializzata;
- Il suo pattern è molto fine;
- La materializzazione non riduce di almeno un ordine di grandezza il carico di lavoro.





CoScienze
Associazione



CoScienze
Associazione