

# Basi di dati

Francesco Bonistalli

March 2022

# Indice

<b>I Vaglini</b>	<b>13</b>
<b>1 Introduction</b>	<b>14</b>
<b>2 Modello relazionale</b>	<b>16</b>
2.1 modelli logici nei sistemi di basi di dati . . . . .	16
2.2 Tabelle e relazioni . . . . .	16
2.3 Relazioni e basi di dati . . . . .	17
2.4 Informazione incompleta e valori nulli . . . . .	17
2.5 Vincoli di integrità . . . . .	18
2.6 Chiavi . . . . .	19
2.6.1 Tipi di Chiave . . . . .	19
2.7 Vincoli di integrità referenziale . . . . .	19
<b>3 Algebra relazionale</b>	<b>21</b>
3.1 Introduzione . . . . .	21
3.2 Operatori dell'algebra relazionale . . . . .	23
3.2.1 Operatori su insiemi . . . . .	23
3.2.2 Ridenominazione . . . . .	24
3.2.3 Selezione . . . . .	24
3.2.4 Proiezione . . . . .	24
3.2.5 Join . . . . .	25
3.2.6 Equivalenza di espressioni . . . . .	26
3.3 Algebra relazionale estesa . . . . .	27
3.4 Relazioni derivate . . . . .	28
3.5 Calcolo relazionale . . . . .	29
<b>4 Progettazione di basi di dati</b>	<b>32</b>
4.1 introduzione . . . . .	32
4.2 Modello concettuale . . . . .	33
4.2.1 Modello E-R . . . . .	34
4.3 Documentazione . . . . .	41
4.4 Progettazione concettuale . . . . .	42
4.5 Progettazione logica . . . . .	44

4.5.1	Dal modello concettuale al modello logico . . . . .	44
4.5.2	Analisi delle prestazioni su schema ER . . . . .	46
4.5.3	Attività di ristrutturazione . . . . .	47
4.5.4	Algoritmo di traduzione nel modello logico . . . . .	51
4.6	Normalizzazione . . . . .	54
4.6.1	Ridondanze e anomalie . . . . .	54
4.6.2	Dipendenze funzionali . . . . .	55
4.6.3	Forma normale di Boyce Codd . . . . .	56
4.6.4	Proprietà delle decomposizioni . . . . .	57
4.6.5	Terza forma normale . . . . .	59
4.6.6	Teoria delle dipendenze e normalizzazione . . . . .	61
4.6.7	progettazione e normalizzazione . . . . .	67
<b>5</b>	<b>Tecnologie della base di dati</b>	<b>68</b>
5.1	Organizzazione fisica e gestione delle interrogazioni . . . . .	68
5.1.1	Memoria principale, secondaria e gestione del buffer .	69
5.2	Gestione delle transazioni . . . . .	71
5.2.1	Transazioni . . . . .	71
5.2.2	Controllo dell'affidabilità . . . . .	74
5.2.3	Controllo della concorrenza . . . . .	85
5.3	Organizzazione fisica . . . . .	96
5.3.1	Intro . . . . .	96
5.3.2	Strutture primarie per l'organizzazione di file . . . . .	100
5.3.3	Strutture ad albero (non sequenziali) . . . . .	103
5.3.4	Strutture fisiche e indici nei DBMS relazionali . . . . .	109
5.3.5	Gestione delle interrogazioni: ottimizzazione . . . . .	109
<b>6</b>	<b>Transazioni in SQL</b>	<b>125</b>
6.1	SQL nei linguaggi di programmazione . . . . .	125
6.1.1	Differenze tra SQL e altri linguaggi: . . . . .	126
6.1.2	Tecniche principali . . . . .	126
<b>II</b>	<b>Pistolesi</b>	<b>130</b>
<b>7</b>	<b>DB e la gestione dei dati</b>	<b>131</b>
7.1	L'interrogazione . . . . .	131
7.2	MySQL . . . . .	131
7.3	Gestione delle date . . . . .	133
7.4	Operatori di aggregazione . . . . .	134
7.5	La gestione delle tabelle (manipolazione del FROM) . . . . .	135
7.6	Derived table . . . . .	138
7.7	lezione 3 . . . . .	139
7.7.1	Join multiplo . . . . .	139

7.7.2	Self join . . . . .	139
7.7.3	Subquery . . . . .	139
<b>8</b>	<b>lezione 4</b>	<b>142</b>
8.1	Raggruppamento . . . . .	142
8.1.1	Condizioni sui gruppi . . . . .	143
8.1.2	Condizioni sui gruppi vs. condizioni sui record . . . . .	143
8.1.3	Raggruppamento su più attributi . . . . .	144
8.2	CTE . . . . .	144
<b>9</b>	<b>Lezione 5</b>	<b>147</b>
9.1	Correlated subquery . . . . .	147
9.2	Divisione . . . . .	148
9.3	Differenza . . . . .	149
9.4	Modificatori . . . . .	149
9.5	Query complesse . . . . .	150
9.6	Data manipulation . . . . .	150
9.6.1	Inserimento . . . . .	151
9.6.2	Aggiornamento . . . . .	152
9.6.3	Cancellazione . . . . .	152
<b>10</b>	<b>Lezione 6</b>	<b>155</b>
10.1	Stored procedure . . . . .	155
10.1.1	Prestazioni . . . . .	156
10.1.2	Sicurezza . . . . .	156
10.1.3	Riuso del codice . . . . .	156
10.1.4	chiamata . . . . .	157
10.2	Variabili . . . . .	157
10.2.1	Variabili locali . . . . .	157
10.2.2	Variabili user-defined . . . . .	159
10.3	Parametri . . . . .	159
10.3.1	Ingresso (IN) . . . . .	159
10.3.2	Uscita . . . . .	160
10.3.3	Ingresso-Uscita . . . . .	160
10.4	Istruzioni condizionali . . . . .	161
10.4.1	Istruzione IF . . . . .	161
10.5	Più variabili di uscita . . . . .	163
10.6	Istruzioni iterative . . . . .	163
10.6.1	WHILE . . . . .	163
10.6.2	REPEAT . . . . .	164
10.6.3	LOOP . . . . .	165
10.7	Istruzioni di salto . . . . .	165
10.8	Cursori . . . . .	166
10.9	Handler . . . . .	167

10.10	Stored function . . . . .	170
10.11	Temporary table . . . . .	172
<b>11</b>	<b>Lezione 7</b>	<b>174</b>
11.1	Databese attivi . . . . .	174
11.2	Trigger . . . . .	174
11.2.1	Tipi di trigger . . . . .	174
11.2.2	Business rule . . . . .	175
11.2.3	Trigger di tipo before . . . . .	175
11.2.4	Trigger multi-statement . . . . .	177
11.3	Gestione di una ridondanza . . . . .	178
11.3.1	Trigger di tipo after . . . . .	179
11.4	Event . . . . .	181
11.4.1	Event a singolo scatto . . . . .	183
11.4.2	Scheduler degli event . . . . .	183
<b>12</b>	<b>Lezione 9</b>	<b>185</b>
12.1	Materialized view . . . . .	185
12.1.1	Utilizzi . . . . .	186
12.1.2	Refresh . . . . .	187
<b>13</b>	<b>lezione 10</b>	<b>200</b>

# List of Theorems

1	Definizione (Sistema informativo) . . . . .	14
2	Definizione (Basi di Dati) . . . . .	14
3	Definizione (Base di dati (accezione tecnologica)) . . . . .	14
4	Definizione (DBMS) . . . . .	14
5	Definizione (Transazione) . . . . .	15
1	Nota . . . . .	15
6	Definizione (database NoSQL) . . . . .	15
7	Definizione (Prodotto cartesiano) . . . . .	16
8	Definizione (Relazione) . . . . .	16
9	Definizione (Struttura non posizionale) . . . . .	16
10	Definizione (Tabelle e relazioni) . . . . .	16
11	Definizione (Schema di relazione) . . . . .	17
12	Definizione (Schema di base di dati) . . . . .	17
13	Definizione (Tupla) . . . . .	17
14	Definizione (Istanza di relazione su uno schema R(X)) . . . . .	17
15	Definizione (Istanza di base di dati su uno schema R) . . . . .	17
16	Definizione (Rappresentazione dell'informazione incompleta) .	17
17	Definizione (Valore nullo) . . . . .	17
18	Definizione (Vincoli di integrità) . . . . .	18
19	Definizione (Vincoli intrarelazionali) . . . . .	18
20	Definizione (Sintassi del vincolo di dominio) . . . . .	18
21	Definizione (Identificatore di tupla) . . . . .	19
22	Definizione (Chiave) . . . . .	19
23	Definizione (Superchiave) . . . . .	19
24	Definizione (Chiave (Superchiave Minimale)) . . . . .	19
25	Definizione (Chiave Primaria) . . . . .	19
26	Definizione (Dipendenza funzionale) . . . . .	19
27	Definizione (Vincolo di integritá referenziale (Chiave esterna))	19
28	Definizione (vincoli su più attributi) . . . . .	19
29	Definizione (integrità referenziale e valori nulli) . . . . .	20
30	Definizione (interrogazione) . . . . .	21
31	Definizione (Query processor) . . . . .	21

32	Definizione (algebra relazionale) . . . . .	21
33	Definizione (Prodotto cartesiano) . . . . .	23
34	Definizione (Unione) . . . . .	23
35	Definizione (intersezione) . . . . .	23
36	Definizione (Differenza) . . . . .	23
37	Definizione (Ridenominazione) . . . . .	24
38	Definizione (Selezione) . . . . .	24
39	Definizione (Proiezione) . . . . .	24
40	Definizione (join naturale) . . . . .	25
41	Definizione (Theta join) . . . . .	26
42	Definizione (Equi join) . . . . .	26
43	Definizione . . . . .	26
44	Definizione . . . . .	27
45	Definizione (Join esterno (outer)) . . . . .	27
46	Definizione (Proiezione generalizzata) . . . . .	27
47	Definizione (Funzioni aggregate) . . . . .	27
48	Definizione (Raggruppamento) . . . . .	28
49	Definizione (Divisione (def da internet)) . . . . .	28
50	Definizione (Relazioni di base) . . . . .	28
51	Definizione (Relazioni derivate) . . . . .	28
52	Definizione (Viste materializzate) . . . . .	29
53	Definizione (Viste virtuali) . . . . .	29
54	Definizione . . . . .	29
55	Definizione (Ciclo di vita) . . . . .	32
56	Definizione (Modello concettuale) . . . . .	33
57	Definizione (Entità (Entity)) . . . . .	34
58	Definizione (Occorrenza (o istanza) di entità) . . . . .	34
59	Definizione (Relazione (Relationship)) . . . . .	35
60	Definizione (occorrenza di relationship) . . . . .	35
61	Definizione (Attributo) . . . . .	36
62	Definizione (Attributi composti) . . . . .	36
63	Definizione (Cardinalità di relationship) . . . . .	37
64	Definizione (Tipi di relationship) . . . . .	38
65	Definizione (Cardinalità di attributi (attributi multivалore)) . .	38
66	Definizione (identificatore di una entità) . . . . .	38
67	Definizione (Generalizzazione) . . . . .	39
68	Definizione (Ereditarietà) . . . . .	39
69	Definizione (Regola aziendale) . . . . .	41
70	Definizione (Strategia top-down) . . . . .	43
71	Definizione (Strategia bottom-up) . . . . .	43
72	Definizione (schema scheletro) . . . . .	43
73	Definizione (Attività di ristrutturazione) . . . . .	47
74	Definizione (Le gerarchie nel modello relazionale (punto 1)) .	47

75	Definizione (Attributo derivabile) . . . . .	49
76	Definizione (Ridondanza) . . . . .	49
77	Definizione (Forme di ridondanza in uno schema ER) . . . . .	49
2	Nota . . . . .	51
3	Nota . . . . .	54
78	Definizione (Dipendenze funzionali (FD)) . . . . .	55
79	Definizione (FD non banale) . . . . .	55
80	Definizione (FD e il vincolo di chiave) . . . . .	55
81	Definizione (Forma normale di Boyce-Codd (BCNF)) . . . . .	56
82	Definizione (Normalizzazione) . . . . .	57
83	Definizione (Decomposizione senza perdita) . . . . .	58
84	Definizione (conservazione delle dipendenze) . . . . .	59
85	Definizione (Terza forma normale) . . . . .	60
4	Nota (importante!!!) . . . . .	60
5	Nota . . . . .	61
86	Definizione (prima forma normale) . . . . .	61
87	Definizione (seconda forma normale) . . . . .	61
88	Definizione (Implicazione) . . . . .	62
89	Definizione (Chiusura) . . . . .	62
6	Nota . . . . .	62
90	Definizione (Algoritmo per il calcolo di $X_F^+$ ) . . . . .	62
7	Nota . . . . .	62
91	Definizione (FD semplici) . . . . .	64
92	Definizione (Attributi estranei) . . . . .	64
93	Definizione (Ridondanza di attributi) . . . . .	64
94	Definizione (Ridondanza di FD) . . . . .	64
95	Definizione (Algoritmo per FD ridondanti) . . . . .	64
96	Definizione (FD e chiave) . . . . .	64
97	Definizione (Chiusura) . . . . .	64
98	Definizione (Regole di inferenza di Armstrong) . . . . .	64
99	Definizione (Regole derivate di Armstrong) . . . . .	65
100	Definizione (Equivalenza) . . . . .	65
101	Definizione (Copertura minimale) . . . . .	66
8	Nota . . . . .	66
102	Definizione (Algoritmo copertura minimale) . . . . .	66
103	Definizione (Buffer) . . . . .	69
104	Definizione (gestore del buffer) . . . . .	70
105	Definizione (Transazione) . . . . .	71
106	Definizione (Sistema transazionale) . . . . .	72
107	Definizione (Transazione (sintatticamente)) . . . . .	72
108	Definizione (Atomicità) . . . . .	73
109	Definizione (Consistenza) . . . . .	73
110	Definizione (Isolamento) . . . . .	73

111	Definizione (Durabilità) . . . . .	73
112	Definizione (log) . . . . .	76
113	Definizione (Checkpoint) . . . . .	79
114	Definizione (Dump) . . . . .	80
115	Definizione (Modalità immediata (caso a)) . . . . .	81
116	Definizione (Modalità differita (caso b)) . . . . .	81
117	Definizione (Modalità mista (caso c)) . . . . .	81
118	Definizione (rollback di una transazione) . . . . .	82
119	Definizione (lettura sporca) . . . . .	86
120	Definizione (Transazione (def. formale)) . . . . .	88
121	Definizione (schedule) . . . . .	88
122	Definizione (Schedule seriale) . . . . .	88
123	Definizione (Schedule serializzabile (view-serializzabile)) . . . . .	89
124	Definizione (view-equivalenza) . . . . .	89
125	Definizione (Conflict-equivalenza) . . . . .	89
126	Definizione (Conflict-seriabilizzabilità) . . . . .	89
127	Definizione (t. ben formata rispetto al locking) . . . . .	91
128	Definizione (lock upgrade) . . . . .	91
129	Definizione (Risorsa acquisita o rilasciata) . . . . .	91
130	Definizione (transazione in stato di attesa) . . . . .	91
131	Definizione (Locking a due fasi (2PL=2 phase locking)) . . . . .	92
9	Nota (upgrading e downgrading dei lock) . . . . .	93
10	Nota (2PL e CSR) . . . . .	93
132	Definizione (Locking a 2 fasi stretto (strict 2PL)) . . . . .	94
11	Nota . . . . .	94
133	Definizione (Timestamp (TS)) . . . . .	94
12	Nota . . . . .	95
13	Nota (2PL e TS) . . . . .	95
134	Definizione (stallo (libro=blocco critico)) . . . . .	95
135	Definizione (DBMS) . . . . .	96
136	Definizione (Tavola hash) . . . . .	101
14	Nota . . . . .	102
137	Definizione (indice) . . . . .	103
15	Nota (commenti) . . . . .	105
16	Nota (Osservazioni su indici secondari) . . . . .	106
138	Definizione (Query processor (o ottimizzatore)) . . . . .	109
139	Definizione (Ottimizzazione basata sui costi) . . . . .	111
140	Definizione (Accesso diretto) . . . . .	111
141	Definizione (Accesso diretto basato su indice) . . . . .	112
142	Definizione (Accesso diretto basato su hash) . . . . .	112
143	Definizione (Indice hash su più campi) . . . . .	112
144	Definizione (Ricerca (scansione)) . . . . .	112
145	Definizione (Ordinamento) . . . . .	112
146	Definizione (Nested loop) . . . . .	113

1	Esempio (Costo per nested loop join) . . . . .	113
147	Definizione (Merge scan) . . . . .	113
148	Definizione (hash join) . . . . .	113
149	Definizione (SQL immerso) . . . . .	126
150	Definizione (Conflitto di impedenza (impedance mismatch)) .	127
151	Definizione (Cursore) . . . . .	128
152	Definizione (SQL dinamico) . . . . .	128
153	Definizione (Call Level Interface) . . . . .	129
154	Definizione (Linguaggio dichiarativo) . . . . .	131
1	Elemento di codifica (Asterisco '*) . . . . .	132
2	Elemento di codifica (Operatori logici) . . . . .	132
3	Elemento di codifica (costrutto DISTINCT) . . . . .	132
4	Elemento di codifica (IS NOT NULL/IS NULL) . . . . .	132
5	Elemento di codifica (DATE, TIMESTAMP e DATE_FORMAT) .	133
6	Elemento di codifica (Estrazione di giorno, mese, anno) . . . . .	133
7	Elemento di codifica (La data odierna) . . . . .	133
8	Elemento di codifica . . . . .	133
9	Elemento di codifica (Sommare intervalli) . . . . .	133
10	Elemento di codifica (Funzioni utility sulle date) . . . . .	134
11	Elemento di codifica (Conteggio) . . . . .	134
12	Elemento di codifica (Somma) . . . . .	134
13	Elemento di codifica (Media aritmetica) . . . . .	134
14	Elemento di codifica (Massimo e Minimo) . . . . .	134
15	Elemento di codifica (combinare più tabelle, il JOIN) . . . . .	135
17	Nota (Ambiguità) . . . . .	139
18	Nota . . . . .	139
155	Definizione (Noncorrelated subquery) . . . . .	140
19	Nota . . . . .	140
156	Definizione (Risultato di una N. subquery) . . . . .	140
16	Elemento di codifica (IN) . . . . .	140
157	Definizione (Equivalenza join-subquery) . . . . .	140
20	Nota . . . . .	140
158	Definizione (annidamento multiplo) . . . . .	140
159	Definizione (Visibilità) . . . . .	141
160	Definizione . . . . .	141
21	Nota . . . . .	141
161	Definizione (Raggruppamento) . . . . .	142
22	Nota . . . . .	142
23	Nota (regola fondamentale raggruppamento) . . . . .	142
24	Nota . . . . .	142
162	Definizione (Condizioni sui gruppi) . . . . .	143

25	Nota . . . . .	143
26	Nota (regola sul raggruppamento su più attributi) . . . . .	144
163	Definizione (Common Table Expression) . . . . .	144
164	Definizione (correlated subquery) . . . . .	147
27	Nota . . . . .	147
28	Nota . . . . .	147
165	Definizione (Costrutto EXISTS) . . . . .	147
166	Definizione (divisione) . . . . .	148
167	Definizione (differenza) . . . . .	149
17	Elemento di codifica (ANY) . . . . .	150
18	Elemento di codifica (ALL) . . . . .	150
29	Nota . . . . .	151
30	Nota (IMPORTANTE) . . . . .	153
168	Definizione (Stored procedure) . . . . .	155
31	Nota . . . . .	155
32	Nota (IMPORTANTE) . . . . .	157
33	Nota . . . . .	157
34	Nota . . . . .	159
35	Nota . . . . .	159
36	Nota . . . . .	160
37	Nota . . . . .	165
38	Nota (warning) . . . . .	166
39	Nota . . . . .	167
40	Nota . . . . .	171
169	Definizione (deterministico) . . . . .	171
41	Nota . . . . .	171
2	Esempio . . . . .	172
42	Nota . . . . .	173
43	Nota . . . . .	174
170	Definizione (Trigger) . . . . .	174
44	Nota (Preprocessing) . . . . .	175
45	Nota (a posteriori) . . . . .	175
46	Nota (NEW) . . . . .	175
47	Nota . . . . .	175
48	Nota . . . . .	175
49	Nota . . . . .	177
50	Nota . . . . .	177
171	Definizione (ridondanza) . . . . .	178
51	Nota . . . . .	178
52	Nota (IMPORTANTE) . . . . .	179
53	Nota . . . . .	179

54	Nota (importante) . . . . .	179
55	Nota . . . . .	181
172	Definizione (Event) . . . . .	181
56	Nota . . . . .	181
57	Nota . . . . .	182
58	Nota . . . . .	182
59	Nota . . . . .	182
60	Nota (ON COMPLETION PRESERVE) . . . . .	183
61	Nota . . . . .	183
173	Definizione (Materialized view) . . . . .	185
62	Nota (fun fact) . . . . .	186
174	Definizione (Refresh) . . . . .	187
63	Nota . . . . .	187
175	Definizione (Incremental refresh) . . . . .	195
176	Definizione (log table) . . . . .	195
177	Definizione (push) . . . . .	195
64	Nota . . . . .	196
65	Nota . . . . .	197
178	Definizione (Data analitycts) . . . . .	200
179	Definizione (Pattern) . . . . .	200
180	Definizione (Windows function) . . . . .	200
66	Nota . . . . .	200
3	Esempio (aggregazioni spinte) . . . . .	201
67	Nota . . . . .	201
4	Esempio (windows functions es 1) . . . . .	201
68	Nota . . . . .	201
181	Definizione (Clausola OVER) . . . . .	202
69	Nota . . . . .	202
182	Definizione (Partition) . . . . .	203
5	Esempio (con def. della partition) . . . . .	203
183	Definizione (windows functions non-aggregate) . . . . .	205
6	Esempio (di rank) . . . . .	206
7	Esempio (su rank n2) . . . . .	207
184	Definizione (DENSE_RANK) . . . . .	208
8	Esempio . . . . .	209
185	Definizione (Lead & Lag) . . . . .	210
186	Definizione (LAG) . . . . .	210
9	Esempio . . . . .	210
187	Definizione (LEAD) . . . . .	211
10	Esempio . . . . .	211
188	Definizione (Frame) . . . . .	212
189	Definizione (windows function su frame) . . . . .	212

190	Definizione (Aggregate functions su frame) . . . . .	212
191	Definizione (Non-aggregate functions che lavorano SOLO su frame) . . . . .	212
192	Definizione (funzione FIRST_VALUE) . . . . .	213
11	Esempio . . . . .	213
193	Definizione (funzione LAST_VALUE) . . . . .	214
12	Esempio . . . . .	214
13	Esempio (moving average (media mobile)) . . . . .	214
14	Esempio . . . . .	215
15	Esempio (variante con frame temporale) . . . . .	216
70	Nota (IMPORTANTE) . . . . .	217
71	Nota . . . . .	217

# Parte I

## Vaglini

# Capitolo 1

## Introduction

**Definizione 1 Sistema informativo:** :

- Il sistema organizzativo è costituito da risorse e regole per lo svolgimento coordinato di attività (processi) per perseguire gli scopi propri di un'organizzazione (azienda o ente) (le risorse possono essere: persone, denaro, materiali, informazioni)
- Il sistema informativo è la componente del sistema organizzativo che acquisisce, elabora, conserva, produce le informazioni di interesse (cioè utili al perseguimento degli scopi); inoltre esegue/gestisce i processi informativi (cioè i processi che coinvolgono informazioni)

□

**Definizione 2 Basi di Dati:** è un insieme organizzato di dati utilizzati per rappresentare le informazioni di interesse e sono caratterizzate:

- da dimensioni molto maggiori della memoria centrale dei sistemi di calcolo utilizzati
- tempo di vita indipendente dalle singole esecuzioni dei programmi che le utilizzano (persistenza dei dati)

□

**Definizione 3 Base di dati (accezione tecnologica):** è un insieme organizzato di dati grandi, persistenti e condivisi gestito da un DataBase Management System (DBMS)

□

**Definizione 4 DBMS:** nei DBMS esiste una porzione della base di dati (il catalogo o dizionario) che contiene la descrizione centralizzata (unica) dei dati, e che può essere utilizzata dai vari programmi. I DBMS usano comunque i file per la memorizzazione dei dati, ma estendono le funzionalità dei file system, fornendo più servizi ed in maniera integrata.

- pro

- gestione centralizzata con possibilità di “economia di scala”
- disponibilità di servizi integrati
- indipendenza dei dati (favorisce lo sviluppo e la manutenzione delle applicazioni)
- contro
  - costo dei prodotti e della transizione verso di essi
  - non scorporabilità (spesso) delle funzionalità (con riduzione di efficienza)

Il DBMS garantisce quindi, nella gestione dei dati, privatezza, efficienza, efficacia, affidabilità.  $\square$

**Definizione 5 Transazione:** è l’unità di lavoro elementare: insieme di operazioni sulla base di dati da considerare indivisibile (atomicità), corretto anche in presenza di concorrenza (controllo della concorrenza) e con effetti definitivi (permanenza)  $\square$

**Nota 1:** *L’uso delle transazioni permette di garantire stati consistenti del database*

**Definizione 6 database NoSQL:** .

- sacrificano l’affidabilità a favore della scalabilità (orizzontale ovvero la distribuzione dei dati su più server, a differenza dei DB transazionali che possono scalare solo verticalmente, ovvero aumentare la potenza del server(cosa difficile da realizzare per via dei costi))
- l’assenza di uno schema consente l’uso di record con un numero di attributi variabile
- "prima o poi sono in stato consistente"

$\square$

#### CAP (Bewer) THEOREM

Asserisce che un sistema distribuito (*dunque non si applica ai DBMS relazionali*) è in grado di supportare solamente due tra le seguenti caratteristiche:

- Consistency      *tutti i nodi vedono gli stessi dati in ogni momento*
- Availability      *ogni operazione deve sempre ricevere risposta*
- Partition Tolerance      *sistema tollerante ad aggiunta/rimozione di un nodo*

I database NoSQL consentono strutture complesse, cosa impossibile in un database relazionale (*non si possono avere relazioni annidate, ossia relazioni come valori di un campo di un record*), e hanno come modello logico l’aggregato, ossia il raggruppamento di dati secondo un particolare criterio, i **modelli logici NoSQL** più utilizzati sono:

## Capitolo 2

# Modello relazionale

### 2.1 modelli logici nei sistemi di basi di dati

Modello logico proposto da Codd nel 1970, ma usato in DBMS reali solo dal 1981, si basa sul concetto matematico di relazione (con una variante)

**Definizione 7 Prodotto cartesiano:** insieme di **tutte** le n-uple  $(d_1, \dots, d_n)$  tali che  $d_i \in D_i \quad \forall 1 \leq i \leq n$  □

**Definizione 8 Relazione:** sottoinsieme del prodotto cartesiano □

**Definizione 9 Struttura non posizionale:** .

- L'ordinamento tra righe (tuple) è irrilevante
- anche tra colonne è irrilevante

□

### 2.2 Tabelle e relazioni

**Definizione 10 Tabelle e relazioni:** le relazioni hanno naturale rappresentazione per mezzo di tabelle. Una tabella rappresenta una relazione se:

- i valori di ogni colonna sono fra loro omogenei
- le righe sono diverse fra loro
- le intestazioni delle colonne sono diverse tra loro

**oss.** il modello relazionale è basato su valori, quindi i riferimenti fra dati di relazioni diverse sono ottenuti tramite valori uguali in tuple diverse □

## 2.3 Relazioni e basi di dati

**Definizione 11 Schema di relazione:** costituito da un nome R (della relazione) e da un insieme di attributi  $A_1, \dots, A_n$

$$R(A_1, \dots, A_n)$$

oppure

$$R(X)$$

con  $X = \{A_1, \dots, A_n\}$

□

**Definizione 12 Schema di base di dati:** insieme di schemi di relazione

$$R = \{R_1(X_1), \dots, R_n(X_n)\}$$

□

**Definizione 13 Tupla:** una tupla su un insieme di attributi X è una funzione che associa a ciascun attributo di A in X un valore del dominio di A ( $t[A]$  denota il valore della tupla sull'attributo A)

□

**Definizione 14 Istanza di relazione su uno schema R(X):** insieme r di tuple su X

□

**Definizione 15 Istanza di base di dati su uno schema R:** insieme di relazioni  $r = \{r_1, \dots, r_n\}$  (con  $r_i$  relazione su  $R_i$ )

□

## 2.4 Informazione incompleta e valori nulli

**Definizione 16 Rappresentazione dell'informazione incompleta:** si usa un valore distinto aggiunto a tutti i domini per indicare l'assenza di informazione detto valore nullo:

- valore nullo: denota l'assenza di un valore del dominio
- se i valori dell'attributo A appartengono al dominio  $\text{dom}(A)$ ,  $t[A]$  è un valore del dominio oppure il valore nullo (NULL)

(Si possono e devono imporre restrizioni sulla presenza di valori nulli in una relazione)

□

**Definizione 17 Valore nullo:** può avere 3 tipi diversi di significato:

- valore sconosciuto
- valore insesistente
- valore non interessante

**Attenzione!**: i DBMS non distinguono tra tipi diversi di valore nullo

□

## 2.5 Vincoli di integrità

**Definizione 18 Vincoli di integrità:** si devono associare alla base di dati delle proprietà che, se soddisfatte, esprimono la sua correttezza rispetto all'applicazione. I vincoli di integrità:

- permettono una descrizione più accurata della realtà
- danno un contributo alla "qualità dei dati"
- sono utili alla progettazione
- sono usati dai DBMS nella esecuzione delle interrogazioni

Essi corrispondono a proprietà del mondo reale modellato dalla base di dati e interessano tutte le istanze; sono inoltre associati allo schema e si considerano corrette le sue istanze che soddisfano tutti i vincoli.

**def.** Un vincolo è un predicato che associa ad ogni istanza della base di dati il valore vero o falso (se il predicato vale vero la proprietà è soddisfatta).

Ne esistono 2 tipi:

- intrarelazionali
- interrelazionali

□

**Definizione 19 Vincoli intrarelazionali:** esprimono condizioni rispetto a singole relazioni della base di dati:

- vincoli di tupla: esprimono condizioni sui valori di ciascuna tupla, indipendentemente dalle altre
- (caso particolare) vincoli di dominio: coinvolgono un solo attributo

□

**Definizione 20 Sintassi del vincolo di dominio:** espressione booleana di atomi che confrontano valori di attributo o espressioni aritmetiche su di essi

$$(voto \geq 18) AND (voto \leq 30)$$

□

### Vincoli su più domini

$$(Voto = 30) OR NOT (Lode = "e lode")$$

Stipendi	Impiegato	Lordo	Ritenute	Netto
	Rossi	55.000	12.500	42.500
	Neri	45.000	10.000	35.000
	Bruni	47.000	11.000	36.000

$$\text{Lordo} = (\text{Ritenute} + \text{Netto})$$

**Definizione 21 Identificatore di tupla:** .

Matricola	Cognome	Nome	Corso	Nascita
27655	Rossi	Mario	Ing Inf	5/12/78
78763	Rossi	Mario	Ing Inf	3/11/76
65432	Neri	Piero	Ing Mecc	10/7/79
87654	Neri	Mario	Ing Inf	5/11/76
67653	Rossi	Piero	Ing Mecc	5/12/78

- non ci sono due tuple con lo stesso valore dell'attributo Matricola
- non ci sono due tuple con lo stesso valore di tutti e tre gli attributi Cognome, Nome e Nascita

□

## 2.6 Chiavi

**Definizione 22 Chiave:** insieme di attributi che identificano univocamente le tuple di una relazione

□

### 2.6.1 Tipi di Chiave

**Definizione 23 Superchiave:** insieme di attributi che non contiene tuple duplicate al suo interno

□

**Definizione 24 Chiave (Superchiave Minimale):** insieme di attributi che al suo interno non contiene altre superchiavi

□

**Lemma:** un insieme di attributi è una chiave se è una superchiave

**Definizione 25 Chiave Primaria:** Chiave su cui non sono ammessi valori nulli (notazione = sottolineatura)

□

**Definizione 26 Dipendenza funzionale:** dati due insiemi di attributi X e Y si dice che X determina Y o che Y dipende funzionalmente da X, e si scrive  $X \rightarrow Y$  se date due tuple distinte t1 e t2, se  $t1[X] = t2[X]$  allora  $t1[Y] = t2[Y]$

□

## 2.7 Vincoli di integrità referenziale

**Definizione 27 Vincolo di integrità referenziale (Chiave esterna):** Un vincolo di integrità referenziale fra gli attributi X (anche più di uno) di una relazione  $R_1$  e un'altra relazione  $R_2$  impone ai valori su X di ciascuna tupla in  $R_1$  di comparire come valori della chiave primaria di  $R_2$ .

Permettono di collegare i valori di un attributo di una tabella con quelli di un'altra o della stessa (es. relationship ricorsiva)

□

**Definizione 28 vincoli su più attributi:** l'ordine degli attributi tra cui è stabilito il vincolo è significativo

□

**Definizione 29 integrità referenziale e valori nulli:** in presenza di valori nulli i vincoli possono essere resi meno restrittivi. il vincolo non è fra ogni valore degli attributi X di una relazione  $R_1$  e la chiave primaria della relazione  $R_2$ , ma tra i valori di X diversi da NULL e la chiave primaria di  $R_2$   $\square$

### Integrità referenziale e valori nulli

Impiegati	Matricola	Cognome	Progetto
	34321	Rossi	IDEA
	53524	Neri	XYZ
	64521	Verdi	NULL
	73032	Bianchi	IDEA

Progetti	Codice	Inizio	Durata	Costo
	IDEA	01/2000	36	200
	XYZ	07/2001	24	120
	BOH	09/2001	24	150

# Capitolo 3

## Algebra relazionale

### 3.1 Introduzione

È un linguaggio procedurale basato su concetti di tipo algebrico costituito da un insieme di operatori definiti su relazioni e che producono ancora relazioni come risultati. In questo modo è possibile costruire espressioni che coinvolgono più operatori allo scopo di formulare interrogazioni anche complesse.

**Definizione 30 interrogazione:** operazione di lettura sul DB che può richiedere l'accesso a più di una tabella.

**Semantica di una interrogazione**

- modo operazionale: si specificano le modalità di generazione del risultato
- modo dichiarativo: si specificano le proprietà del risultato

Si definisce il comportamento delle interrogazioni in modo **operazionale** utilizzando espressioni dell'algebra relazionale.

In modo **dichiarativo** si definisce qual è il risultato di un'interrogazione utilizzando espressioni del calcolo relazionale  $\square$

Il metodo dichiarativo è l'effettiva semantica del linguaggio, infatti le interrogazioni sono espresse ad alto livello (indipendenza dei dati).

Tramite l'algebra operazionale si definisce il modo in cui il DBMS esegue una istruzione SQL

**Definizione 31 Query processor:** modulo specifico contenuto nel DBMS all'interno del quale è definito il processo di esecuzione delle interrogazioni (una parte del QP si occupa di ottimizzare la query prima dell'esecuzione)  $\square$

**Definizione 32 algebra relazionale:** • Algebra = dati + operatori

- Algebra relazionale:
  - su relazioni che producono relazioni e possono essere composti

□

## 3.2 Operatori dell'algebra relazionale

**Definizione 33 Prodotto cartesiano:**

- operatore binario
- date due relazioni  $r_1$  e  $r_2$ , crea una relazione composta da tutte le combinazioni possibili delle tuple della prima con le tuple della seconda
- simbolo =  $\times$

□

### 3.2.1 Operatori su insiemi

Le relazioni sono insiemi quindi si possono applicare gli operatori su insiemi (i risultati devono essere ancora relazioni)

**Definizione 34 Unione:** L'unione di due relazioni sullo stesso insieme di attributi  $X$  è una relazione su  $X$  che contiene le tuple sia dell'una che dell'altra relazione originaria (i doppiioni non vengono aggiunti) □

Laureati triennali			Laureati magistrali		
Matricola	Nome	Età	Matricola	Nome	Età
7274	Rossi	32	9297	Neri	33
7432	Neri	24	7432	Neri	24
9824	Verdi	25	9824	Verdi	25

Laureati triennali $\cup$ Laureati magistrali		
Matricola	Nome	Età
7274	Rossi	32
7432	Neri	24
9824	Verdi	25
9297	Neri	33

**Definizione 35 intersezione:** L'intersezione di due relazioni sullo stesso insieme di attributi  $X$  è una relazione su  $X$  che contiene le tuple appartenenti ad entrambe le relazioni (quelle in comune) □

Laureati triennali			Laureati magistrali		
Matricola	Nome	Età	Matricola	Nome	Età
7274	Rossi	32	9297	Neri	33
7432	Neri	24	7432	Neri	24
9824	Verdi	25	9824	Verdi	25

Laureati triennali $\cap$ Laureati magistrali		
Matricola	Nome	Età
7432	Neri	24
9824	Verdi	25

**Definizione 36 Differenza:** è una relazione che contiene le tuple di  $R_1$  che non appartengono anche a  $R_2$  □

### 3.2.2 Ridenominazione

**Definizione 37 Ridenominazione:** "modifica lo schema" dell'argomento lasciando inalterata l'istanza (operatore monadico = con un argomento)

$\rho_{\text{Genitore} \leftarrow \text{Padre}} (\text{Paternità})$	
Genitore	Figlio
Adamo	Abele
Adamo	Caino
Abramo	Isacco

$\rho_{\text{Genitore} \leftarrow \text{Padre}} (\text{Paternità})$	
Genitore	Figlio
Adamo	Abele
Adamo	Caino
Abramo	Isacco

$\rho_{\text{Genitore} \leftarrow \text{Madre}} (\text{Maternità})$	
Genitore	Figlio
Eva	Abele
Eva	Set
Sara	Isacco

$\rho_{\text{Genitore} \leftarrow \text{Madre}} (\text{Maternità})$	
Genitore	Figlio
Adamo	Abele
Adamo	Caino
Abramo	Isacco
Eva	Abele
Eva	Set
Sara	Isacco

□

### 3.2.3 Selezione

**Definizione 38 Selezione:** operatore che produce un risultato che:

- ha lo stesso schema dell'argomento
- contiene un sottoinsieme delle sue tuple che soddisfano una condizione fissata
- operatore monadico
- simbolo =  $\sigma$

#### Sintassi e semantica

- data una relazione  $r(X)$   
 $\sigma_F(r) = r'$ 
  - $F$ : espressione booleana ottenuta componendo con and, or e not condizioni atomiche del tipo  $A \neq B$ , oppure  $A \neq c$ , con  $A$  e  $B$  attributi in  $X$  con domini compatibili, o operatore di confronto ( $<$ ,  $>$ ,  $=$ ) e  $c$  costante compatibile con il dominio di  $A$ .
  - $r'$  contiene il sottoinsieme delle tuple di  $r$  per cui  $F$  è vera

□

### 3.2.4 Proiezione

**Definizione 39 Proiezione:** .

- operatore che produce un risultato che ha parte degli attributi dell'argomento e su tali attributi contiene tutte le possibili tuple di valori esistenti nella relazione argomento

- operatore monadico
- simbolo=  $\pi$

Sintassi dato  $y \subseteq X$

$$\pi_y(r(X)) = r'$$

□

### 3.2.5 Join

è l'operatore piú interessante dell'algebra relazionale perché permette di correlare dati in relazioni diverse

**Definizione 40 join naturale:**

- operatore binario(generalizzabile)
- produce come risultato una relazione tale che:
  - il suo schema ha l'unione degli attributi degli argomenti
  - l'insieme delle tuple è ottenuto componendo una tupla di ognuno degli operandi per valori uguali degli attributi comuni (stesso nome)
  - simbolo=  $\bowtie$

Impiegato	Reparto	Reparto	Capo
Rossi	A	B	Mori
Neri	B	C	Bruni
Bianchi	B		

Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Mori

Figura 3.1: Join Naturale

**Osservazione:** Funziona anche senza attributi comuni e ha come risultato una relazione che contiene sempre un numero di tuple pari al prodotto delle cardinalità degli operandi (equivale al prodotto cartesiano su tuple [33](#))

R		Q	
Impiegato	Reparto	Codice	Capo
Rossi	A	A	Mori
Neri	B	B	Bruni
Bianchi	B		

R $\bowtie Q$			
Impiegato	Reparto	Codice	Capo
Rossi	A	A	Mori
Rossi	A	B	Bruni
Neri	B	A	Mori
Neri	B	B	Bruni
Bianchi	B	A	Mori
Bianchi	B	B	Bruni

Figura 3.2: Join senza attributi comuni

□

**Definizione 41 Theta join:** .

- operatore derivato
- è un prodotto cartesiano ridotto eseguendo una selezione su una condizione F (F è spesso una congiunzione di atomi di confronto tra 2 attributi di relazioni diverse mediante uno degli operatori di confronto)
- simbolo =  $R_1 \bowtie_F R_2$

□

**Definizione 42 Equi join:** è un theta join (41) nel caso in cui l'operatore è sempre l'uguaglianza(=)

□

### 3.2.6 Equivalenza di espressioni

**Definizione 43:** Due espressioni sono equivalenti se producono lo stesso risultato qualunque sia l'istanza attuale della basi di dati (è un concetto importante perché i DBMS cercano di eseguire espressioni equivalenti a quelle date ma meno "costose")

□

- **push selections down** con A attributo di  $R_2$

$$\sigma_{A=10}(R_1 \bowtie R_2) = R_1 \bowtie \sigma_{A=10}(R_2)$$

riduce in modo isgnificativo la dimensione del risultato intermedio e quindi il costo dell'operazione

- **push projections down** con  $Y_2 \subseteq X_2$

$$\pi_{x_1 y_2}(R_1 \bowtie R_2) = R_1 \bowtie \pi_{y_2}(R_2)$$

riduce in modo significativo la dimensione del risultato intermedio

Il processo di **ottimizzazione algebrica** si basa sulla nozione di equivalenza (due espressioni sono equivalenti se producono lo stesso risultato qualunque sia l'istanza attuale della base di dati) da cui l'**euristica fondamentale** secondo cui selezioni e proiezioni il più presto possibile per ridurre le dimensioni dei risultati intermedi

**Procedura euristica dell'ottimizzatore per la prima fase dell'ottimizzazione algebrica**

1. Decomporre le selezioni congiuntive in successive selezioni atomiche
2. Anticipare il più possibile le selezioni
3. In una sequenza di selezioni, anticipare le più selettive
4. Combinare prodotti cartesiani e selezioni per formare join
5. Anticipare il più possibile le proiezioni (anche introducendone di nuove)

### 3.3 Algebra relazionale estesa

**Definizione 44:** Il modello relazionale può essere facilmente esteso a comprendere gli operatori SQL non direttamente riconducibili agli operatori algebrici introdotti (questa estensione non modifica il funzionamento del modello)  $\square$

**Definizione 45 Join esterno (outer):** permette di generare valori null per mezzo delle espressioni dell'algebra relazionale per modellare le informazioni mancanti

- **Left** ( $=\bowtie$ ): tutte le tuple dell'operando sinistro sono mantenute, quelle che non fanno join (dell'operando destro) sono riempite con NULL
- **Right** ( $\bowtie=$ ): idem per l'operando destro
- **Full** ( $=\bowtie=$ ): per entrambi gli operandi

$\square$

**Definizione 46 Proiezione generalizzata:**

$$\pi_{f_1, f_2, f_3}(E)$$

con  $f_1$ ,  $f_2$  e  $f_3$  espressioni aritmetiche (esempio fare una differenza o una somma tra vari attributi ecc) su attributi di  $E$  e costanti  $\square$

**Definizione 47 Funzioni aggregate:** Si possono usare nelle espressioni alcuni nomi di funzioni (operatori) che si applicano a (multi)insiemi e producono un valore come risultato, essi sono:

- $SUM_y(R)$

- COUNT
- MAX
- COUNT DISTINCT

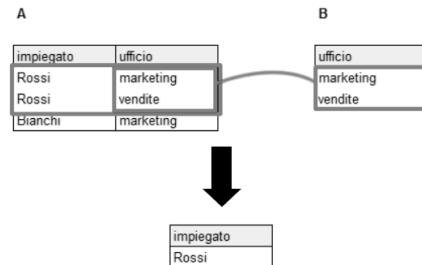
□

**Definizione 48 Raggruppamento:** si possono raggruppare gli elementi di una relazione usando un'operatore apposito

$$\text{cliente} G_{\text{sum}}(\text{credito})(\text{conto})$$

cliente è l'attributo su cui si fa il raggruppamento, sum è la funzione aggregata che si applica all'attributo credito, conto è la relazione su cui si applica il tutto (ci possono essere più attributi a sx di G e più funzioni a dx) □

**Definizione 49 Divisione (def da internet):** La divisione è un operatore binario dell'algebra relazionale che contiene le tuple  $\langle x \rangle$  della prima relazione A tali che per ogni tupla  $\langle y \rangle$  della seconda relazione B ci sia una tupla  $\langle x, y \rangle$  nella prima relazione A. (è utile per interrogazioni di tipo "universale")



Voli	Codice	Data
AZ427	21/07/2001	
AZ427	23/07/2001	
AZ427	24/07/2001	
TW056	21/07/2001	
TW056	24/07/2001	
TW056	25/07/2001	

Linee	Cedice
AZ427	
	TW056

Voli + Linee	Codice	Data
(Voli + Linee) $\bowtie_d$ Linee	AZ427	21/07/2001
	AZ427	24/07/2001
	TW056	21/07/2001
	TW056	24/07/2001

La divisione trova le date con voli per tutte le linee

□

### 3.4 Relazioni derivate

**Definizione 50 Relazioni di base:** contenuto autonomo □

**Definizione 51 Relazioni derivate:** relazioni il cui contenuto è funzione del contenuto di altre relazioni (ed è definito per mezzo di interrogazioni). Due tipi di relazioni derivate:

- viste materializzate
- viste virtuali (viste)

□

**Definizione 52 Viste materializzate:** .  
relazioni derivate memorizzate nella base di dati.

- vantaggio: immediatamente disponibili per le interrogazioni
- svantaggi:
  - ridondanti
  - appesantiscono gli aggiornamenti
  - non sempre supportate dai DBMS

□

**Definizione 53 Viste virtuali:** .

- sono supportate da tutti i DBMS
- una interrogazione su una vista viene eseguita "ricalcolando" la vista (o quasi)

**Esempio**

Afferenza	Impiegato	Reparto	Direzione	
	Rossi	A	Reparto	Capo
Neri	B		A	Mori
Bianchi	B		B	Bruni
Verdi	C		C	Leoni

• una vista  
Supervisione =  
 $\pi_{\text{Impiegato}, \text{Capo}} (\text{Afferenza} \triangleright \triangleleft \text{Direzione})$

- **Interrogazioni sulle viste:** sono eseguite sostituendo alla vista la sua definizione

□

### 3.5 Calcolo relazionale

**Definizione 54:** Una famiglia di linguaggi dichiarativi (vedi:30) basati sul calcolo dei predicati del primo ordine

□

Ne esistono diverse versioni

- calcolo relazionale su domini
- calcolo su ennuple con dichiarazioni di range

## 4. Calcolo relazionale

Il calcolo relazionale è una famiglia di linguaggi **dichiarativi**, basati sul calcolo dei predicati del prim'ordine: *un linguaggio formale che serve per gestire meccanicamente enunciati e ragionamenti che coinvolgono i connettivi logici, le relazioni e i quantificatori.* Algebra e calcolo relazionali sono equivalenti (*puoi tradurre un'interrogazione dall'algebra al calcolo e viceversa*) ma tuttavia hanno dei limiti: le interrogazioni ricorsive come la **chiusura transitiva** non sono rappresentabili (*ad esempio trovare la scala gerarchica di una azienda*)

### Calcolo relazionale sui domini

Le espressioni hanno forma  $\{ A_1: x_1, \dots, A_n: x_n \mid f \}$  dove  $A_i$  sono i nomi degli attributi di una relazione ed esprimono la struttura del risultato,  $x_i$  sono gli effettivi valori che questi assumono nelle varie tuple ed  $f$  è una formula dove compaiono in congiunzione varie condizioni ed eventualmente quantificatori esistenziali ( $\exists$ ) ed universali ( $\forall$ ).

Difetti: è possibile scrivere espressioni insensate e i nomi delle variabili vengono ripetuti spesso

### Calcolo relazionale su tuple con dichiarazioni di range (il più vicino all'SQL)

A differenza del calcolo sui domini ogni variabile qui rappresenta una tupla, le espressioni hanno forma  $\{ x_1.Z_1, \dots, x_n.Z_n \mid x_1(R_1), \dots, x_n(R_n) \mid f \}$  dove  $x_i$  sono tuple prese sugli insiemi di attributi  $Z_i$  (*per indicare che voglio tutti gli attributi di un certo  $Z_i$  uso \**) delle una relazioni  $R_i$  (*range list*) ed  $f$  è una formula dove compaiono in congiunzione varie condizioni ed eventualmente quantificatori esistenziali ( $\exists$ ) ed universali ( $\forall$ ).

Difetti: non si possono esprimere interrogazioni i quali risultati possono provenire da due o più relazioni (*quelle che in algebra sfruttano le unioni*) perché ogni variabile (*tupla*) ha un solo range (*elenco di possibili attributi*)

Supponiamo di voler formulare un'interrogazione che restituisca l'unione di  $R_1(A)$  e  $R_2(A)$

- $\{ x_1.* \mid x_1(R_1) \mid true \} \rightarrow$  Il risultato è composto dalle **tuple di una sola** relazione
- $\{ x_1.*, x_2.* \mid x_1(R_1), x_2(R_2) \mid true \} \rightarrow$  Il risultato è composto solo dalle **tuple di entrambe** le relazioni, mentre l'unione vuol dire prendere le tuple di una o dell'altra relazione

*Nota: è possibile esprimere l'unione di due relazioni identiche (il range è lo stesso).*

## Esempi di interrogazioni nei vari linguaggi

Data la base di dati : **Impiegati**(*Matricola*, *Nome*, *Eta*, *Stipendio*) , **Supervisione**(*Capo*, ***Impiegato***)

- 1) Trova nome e stipendio dei capi degli impiegati che guadagnano più di 40'000
- 2) Trova matricola e nome dei capi degli impiegati che guadagnano tutti più di 40'000

In algebra relazionale

$$\begin{aligned} \pi_{\text{NomeC}, \text{StipendioC}} & \left( \rho_{\text{MatricolaC}, \text{NomeC}, \text{StipendioC}, \text{EtaC} \leftarrow \text{Matricola}, \text{Nome}, \text{Stipendio}, \text{Eta}}(\mathbf{Impiegati}) \right. \\ & \left. \bowtie_{\text{MatricolaC} = \text{Capo}} \mathbf{Supervisione} \bowtie_{\text{Impiegato} = \text{Matricola}} \sigma_{\text{Stipendio} > 40}(\mathbf{Impiegati}) \right) \\ \pi_{\text{Matricola}, \text{Nome}} & \left( \mathbf{Impiegati} \bowtie_{\text{Matricola} = \text{Capo}} [\pi_{\text{Capo}}(\mathbf{Supervisione}) - \right. \\ & \left. \pi_{\text{Capo}}(\mathbf{Supervisione}) \bowtie_{\text{Impiegato} = \text{Matricola}} \sigma_{\text{Stipendio} \leq 40}(\mathbf{Impiegati})] \right) \end{aligned}$$

Nel **calcolo sui domini** (andrebbero indicati tutti gli attributi anche con variabili inutilizzate)

$$\begin{aligned} & \{ \text{NomeC: nc, StipendioC: sc} \mid \\ & \mathbf{Impiegati}(\text{Matricola: m, Nome: n, Età: e, Stipendio: s}) \wedge s > 40 \\ & \quad \wedge \mathbf{Supervisione}(\text{Impiegato: m, Capo: c}) \\ & \quad \wedge \mathbf{Impiegato}(\text{Matricola: c, Nome: nc, Età: ec, Stipendio: sc}) \} \\ \\ & \{ \text{Matricola: m, Nome: n} \mid \\ & \mathbf{Impiegato}(\text{Matricola: c, Nome: n, Età: e, Stipendio: s}) \\ & \quad \wedge \mathbf{Supervisione}(\text{Impiegato: m, Capo: c}) \\ & \quad \wedge \neg (\exists m' (\exists n' (\exists e' (\exists s' \mathbf{Impiegato}(\text{Matricola: m', Nome: n', Età: e', Stipendio: s') \\ & \quad \wedge \mathbf{Supervisione}(\text{Impiegato: m', Capo: c}) \wedge s' \leq 40)))) \} \end{aligned}$$

Nel **calcolo sulle tuple**

$$\begin{aligned} & \{ \text{NomeC, StipC: i'. (Matricola, Nome)} \mid \\ & i'(\mathbf{Impiegato}), s(\mathbf{Supervisione}), i(\mathbf{Impiegato}) \mid \\ & i'. \text{Matr} = s. \text{Capo} \wedge s. \text{Impiegato} = i. \text{Matricola} \wedge i. \text{Stipendio} > 40 \} \\ \\ & \{ i. (\text{Matricola, Nome}) \mid i(\mathbf{Impiegato}), s(\mathbf{Supervisione}) \mid \\ & i. \text{Matr} = s. \text{Capo} \wedge \neg [\exists i'(\mathbf{Impiegato})(\exists s'(\mathbf{Supervisione}) \\ & (s. \text{Capo} = s'. \text{Capo} \wedge s'. \text{Impiegato} = i'. \text{Matricola} \wedge i'. \text{stipendio} \leq 40))] \} \end{aligned}$$

## Capitolo 4

# Progettazione di basi di dati

### 4.1 introduzione

Lo sviluppo di sistemi software in generale, e di sistemi informativi in particolare, è un'attività che comprende diverse fasi.

**Definizione 55 Ciclo di vita:** sequenza di attività nello sviluppo e nell'uso dei sistemi software

#### Fasi del ciclo di vita

- Raccolta e analisi dei requisiti: studio delle proprietà del sistema
- Progettazione: individuazione dei dati e delle funzioni
- Realizzazione
- Validazione e collaudo: sperimentazione
- Funzionamento: il sistema diventa operativo

□

La progettazione è una fase del ciclo di vita ed è composta fondamentalmente di due aspetti:

1. progettazione dei dati (che nel caso dei sistemi informativi ha un ruolo centrale)
2. progettazione delle applicazioni

La progettazione dei dati può essere effettuata su 3 diversi livelli di astrazione (3 fasi):

- **Livello concettuale:** esprime i requisiti di un sistema in una descrizione adatta all'analisi da punti di vista esterno (si crea uno schema che tramite un algoritmo va a creare le tabelle vere e proprie ovvero si ha una traduzione automatica nel livello logico): si rappresentano le

specifiche informali delle realtà di 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 si chiama *schema concettuale* e fa riferimento a un *modello concettuale* dei dati.

- **Livello logico:** evidenzia l'organizzazione dei dati dal punto di vista del loro contenuto informativo, descrivendo la struttura di ciascun record e i collegamenti tra record diversi (andiamo noi in prima persona a creare le tabelle): consiste nella traduzione dello schema concettuale in termini del modello di rappresentazione dei dati adottato dal sistema di gestione di base di dati a disposizione. Il prodotto di questa fase viene denominato *schema logico* della base di dati e fa riferimento a un *modello logico* dei dati. Il modello logico consente di descrivere i dati secondo una rappresentazione ancora indipendente da dettagli fisici, ma concreta perché disponibile nei sistemi di gestione di base di dati. Inoltre in questa fase le scelte progettuali si basano anche su criteri di ottimizzazione delle operazioni da effettuare sui dati. Infine si fa comunemente uso anche di tecniche formali di verifica della qualità dello shcema logico (nel modello relazionale la tecnica comunemente utilizzata è quella della *normalizzazione*.)
- **Livello fisico:** a questo livello la base di dati è vista come un insieme di blocchi fisici su disco. Qui viene decisa l'allocazione dei dati e le modalità di memorizzazione dei dati sul disco (non ci interessa)



## 4.2 Modello concettuale

**Definizione 56 Modello concettuale:** I dati sono rappresentati in modo più vicino al modo di pensare umano (cerca di replicare i concetti del mondo reale). Prevede efficaci rappresentazioni grafiche. □

A questo livello di progettazione i requisiti di un qualsiasi sistema informatico sono specificati in modo:

- **formale**: cioè in modo non ambiguo, ma adeguato a catturare le caratteristiche fondamentali del mondo da descrivere
- **integrato**: la descrizione si riferisce alla totalità dell'ambiente

#### 4.2.1 Modello E-R

Il modello E-R (entity-relationship) inventato da Chen del 1976 è ormai affermato nelle metodologie di progetto. Essendo un modello concettuale fornisce una serie di strutture, atte a descrivere la realtà di interesse in una maniera facile e che prescinde dai criteri di organizzazione dei dati nei calcolatori, dette *costrutti* che vengono poi utilizzati per definire *schemi* che descrivono l'organizzazione e la struttura delle *occorrenze*<sup>1</sup> dei dati, ovvero dei valori assunti dai dati al variare del tempo. È basato su 3 costrutti di base:

- Entità
- Relazioni
- attributi

e altri costrutti come Identificatore, generalizzazione ecc. Con cui si può rappresentare e tradurre qualsiasi base di dati.

#### Costrutti base

**Definizione 57 Entità (Entity):** Classe di oggetti (fatti, persone, cose...) della applicazione di interesse con proprietà comuni e con esistenza "autonoma" □

**Definizione 58 Occorrenza (o istanza di entità):** elemento della classe □

#### Rappresentazione grafica di entità



**caratteristiche dell'entità**= ogni entità ha un nome che la identifica univocamente nello schema:

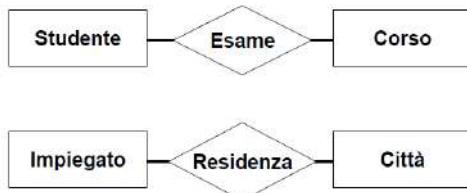
---

<sup>1</sup>si usa occorrenza invece di istanza per non confonderlo con l'istanza (= insieme di tuple) utilizzato nel modello relazionale

- nomi espressivi
- opportune convenzioni (singolare)

**Definizione 59 Relazione (Relationship):** legame logico fra due o più entità, rilevante nell'applicazione di interesse.  
(chiamata anche correlazione o associazione) □

### Rappresentazione grafica di relationship

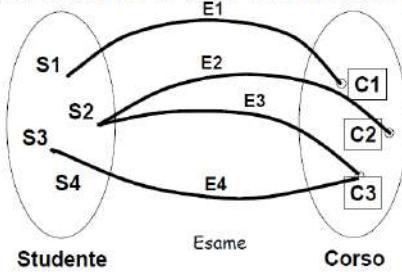


**caratteristiche delle relationship**= ogni relationship ha un nome che la identifica univocamente nello schema:

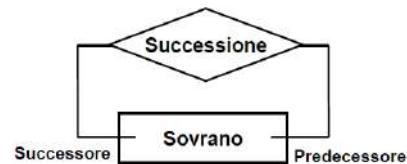
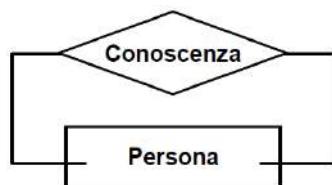
- nomi espressivi
- opportune convenzioni
  - signolare
  - sostantivi invece che verbi (se possibile) per non dare un verso alla relationship

**Definizione 60 occorrenza di relationship:** un'occorrenza di relationship è una t-upla di occorrenze di entità, una per ciascuna delle t entità coinvolte (non ci possono essere occorrenze ripetute sulle stesse occorrenze di entità) □

### Esempi di occorrenze (di entità e di relationship)



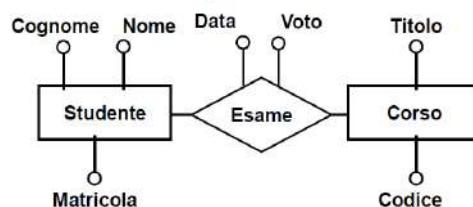
esempi base modello E-R



**Definizione 61 Attributo:** descrive le proprietà elementari di un'entità o di una relationship e associa ad ogni occorrenza di entità o relationship un valore appartenente a un insieme detto dominio dell'attributo

□

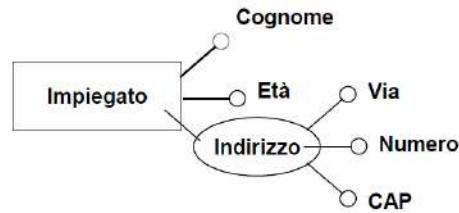
### Attributi: rappresentazione grafica



**Definizione 62 Attributi composti:** raggruppano attributi di una medesima entità nel loro significato o uso

□

## Rappresentazione grafica



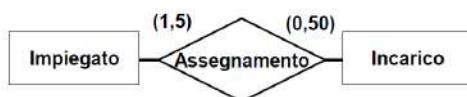
### Altri costrutti

Gli altri costrutti del modello E-R sono:

- cardinalità
  - di relationship
  - di attributo
- identificatore
  - interno
  - esterno
- generalizzazione

**Definizione 63 Cardinalità di relationship:** coppia di valori associati a ogni entità che partecipa a una relationship (specificano il numero minimo e massimo di occorrenze della relationship cui ciascuna occorrenza di entità può partecipare) □

### Esempio di cardinalità



### Simboli della cardinalità:

- 0 e 1 per la cardinalità minima:
  - 0 = "partecipazione opzionale"
  - 1 = "partecipazione obbligatoria"
- 1 e N per la massima:

- 1 associa a una occorrenza dell'entità una sola (o nessuna) dell'altra entità che partecipa alla relazione
- N non pone alcun limite

**Definizione 64 Tipi di relationship:** con riferimento alle cardinalità massime, possiamo caratterizzare una relationship come:

- uno a uno
- uno a molti
- molti a molti

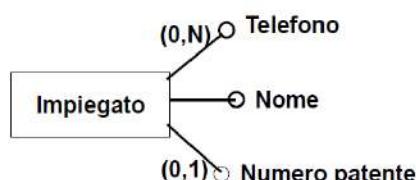
□

**Definizione 65 Cardinalità di attributi (attributi multivaleore):** è possibile associare una cardinalità anche agli attributi, con due scopi:

- indicare opzionalità ("informazione incompleta")
- indicare attributi **multivalore**

□

## Rappresentazione grafica

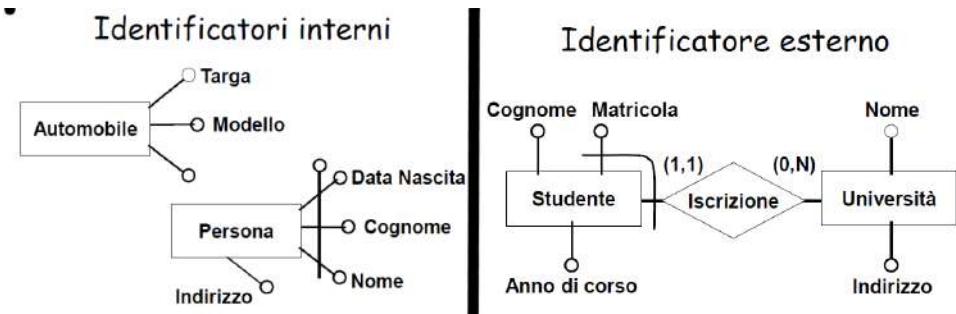


(continua ristrutturazione [75](#))

**Definizione 66 identificatore di una entità:** "strumento" per l'identificazione univoca delle occorrenze di un'entità costituito da:

- attributi dell'entità (identificatore interno o chiave)
- (attributi +) la chiave di entità esterne raggiunta attraverso una relationship (identificatore eserno)

□



### Caratteristiche degli identificatori:

- ogni entità deve possedere almeno un identificatore, ma può averne in generale più di uno
- una identificazione esterna è possibile solo attraverso una relationship a cui l'entità da identificare partecipa con una cardinalità  $(1,1)$

**Definizione 67 Generalizzazione:** mette in relazione una o più entità  $E_1, E_2, \dots, E_n$  con una entità  $E$ , che le comprende come casi particolari

- $E$  è generalizzazione di  $E_1, E_2, \dots, E_n$
- $E_1, E_2, \dots, E_n$  sono specializzazioni (o sottotipi) di  $E$

□

### Generalizzazione: rappresentazione grafica



**Proprietà delle generalizzazioni:** se  $E$  (genitore) è generalizzazione di  $E_1, E_2, \dots, E_n$  (figlie):

- ogni proprietà di  $E$  è significativa per  $E_1, E_2, \dots, E_n$
- ogni occorrenza di  $E_1, E_2, \dots, E_n$  è occorrenza anche di  $E$

**Definizione 68 Ereditarietà:** tutte le proprietà (attributi, relationship, altre generalizzazioni) dell'entità genitore vengono ereditate dalle entità figlie e non rappresentate esplicitamente

□

### Tipi di generalizzazioni

- **totale** se ogni occorrenza dell'entità genitore è occorrenza di almeno una delle entità figlie, altrimenti è parziale
- **esclusiva** se ogni occorrenza dell'entità genitore è occorrenza di al più una delle entità figlie entità, altrimenti è sovrapposta

Consideriamo negli schemi solo generalizzazioni esclusive (si può sempre trasformare una generalizzazione sovrapposta in una esclusiva) e distinguiamo fra generalizzazioni parziali e totali.

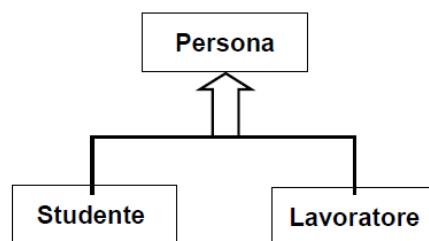


Figura 4.1: freccia vuota = generalizzazione parziale ( ci sono alcune persone che non sono né studenti né lavoratori)

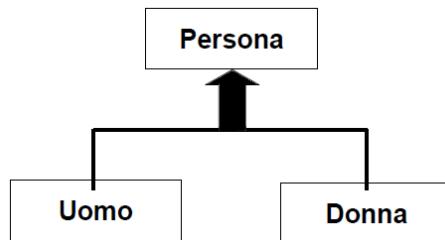


Figura 4.2: generalizzazione totale ed esclusiva (freccia piena) in quanto gli uomini e le donne costituiscono tutte le persone e una persona o è uomo o è donna

Altre proprietà:

- possono esistere gerarchie a più livelli e multiple generalizzazioni allo stesso livello
- un'entità può essere inclusa in più gerarchie, come genitore e/o come figlia
- se una generalizzazione ha solo un'entità figlia si parla di sottoinsieme, di che tipo è questa generalizzazione?

- il genitore di una generalizzazione parziale deve avere un identificatore tra i suoi attributi, quello di una generalizzazione totale no, però? ...

## 4.3 Documentazione

Contiene il **dizionario dei dati** (entità, relazioni) e le **regole aziendali** (vincoli di integrità, possibili derivazioni)

**Definizione 69 Regola aziendale:** una qualunque informazione che definisce o vincola qualche aspetto di un'applicazione. Una regola aziendale può essere:

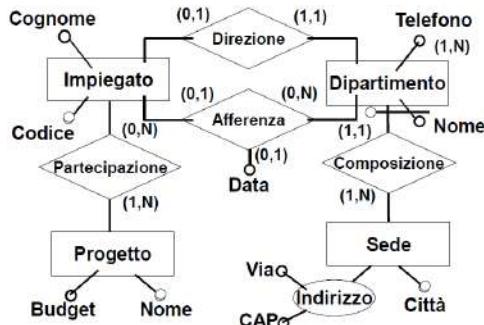
- la descrizione di un concetto (definizione precisa di un costrutto)
- un vincolo di integrità (documentazione di un vincolo sia esprimibile nel modello ER che non)
- una derivazione (un concetto che può essere ottenuto tramite un calcolo aritmetico)

esempio di struttura epr enunciare una regola aziendale

*< concetto > deve/non deve < espressione >*

□

Esempio dcumentazione:



Dizionario dei dati (entità)

Entità	Descrizione	Attributi	Identificatore
Impiegato	Dipendente dell'azienda	Codice, Cognome,	Codice
Progetto	Progetti aziendali	Nome, Budget	Nome
Dipartimento	Struttura aziendale	Nome, Telefono	Nome, Sede
Sede	Sede dell'azienda	Città, Indirizzo	Città

## Dizionario dei dati (relationship)

Relazioni	Descrizione	Componenti	Attributi
Direzione	Direzione di un dipartimento	Impiegato, Dipartimento	
Afferenza	Afferenza a un dipartimento	Impiegato, Dipartimento	Data
Partecipazione	Partecipazione a un progetto	Impiegato, Progetto	
Composizione	Composizione dell'azienda	Dipartimento, Sede	

### Regole di vincolo

(1) Il direttore di un dipartimento deve afferire a tale dipartimento
(2) Un impiegato non deve avere uno stipendio maggiore del direttore del dipartimento al quale afferisce
(3) Un dipartimento con sede a Roma deve essere diretto da un impiegato con più di dieci anni di anzianità
(4) Un impiegato che non afferisce a nessun dipartimento non deve partecipare a nessun progetto

### Regole di derivazione

(1) Il numero di impiegati di un dipartimento si ottiene contando gli impiegati che afferiscono a tale dipartimento
(2) Il budget di un progetto si ottiene moltiplicando per 3 la somma degli stipendi degli impiegati che vi partecipano

I vincoli espressi nelle immagini precedenti e le regole di derivazione riguardano i valori che vengono immessi nelle entità e associazioni in esecuzione, essi stanno solo nella documentazione associata allo schema ER. Alcuni vengono associati alla definizione delle tabelle, altri vengono espressi in altro modo.

## 4.4 Progettazione concettuale

Per determinare quale costrutto ER va utilizzato per rappresentare un concetto presente nelle specifiche dei requisiti bisogna basarsi sulle definizioni dei costrutti del modello ER.:

- se ha proprietà significative e descrive oggetti con esistenza autonoma = entità
- se è semplice e non ha proprietà = attributo
- se correla due o più concetti = relazione

- se è caso particolare di una altro = generalizzazione

Qualità di uno schema concettuale: correttezza, completezza, leggibilità, minimalità

Strategie di progetto: top-down, bottom-up, inside-out.

**Definizione 70 Strategia top-down:** si parte da uno schema iniziale che viene successivamente raffinato e integrato per mezzo di primitive che lo trasformano in una serie di schemi intermedi per arrivare allo schema ER finale.

□

#### Primitive di raffinamento top-down:

- da entità a associazione tra entità
- da entità a generalizzazione
- da associazione a insiemi di associazioni
- da associazione a entità con associazioni
- introduzione di attributi su entità e associazioni

**Definizione 71 Strategia bottom-up:** si parte dalle specifiche iniziali e si suddividono fino a dare specifica ad una componente minima di cui si dà lo schema ER, gli schemi prodotti vengono fusi e integrati fino ad ottenere lo schema finale.

□

#### Primitive di trasformazione bottom-up

- generazione di entità
- generazione di associazione
- generazione di generalizzazione

**Nella pratica effettiva:** si procede di solito con una strategia ibrida:

- si individuano i concetti principali e si realizza uno schema scheletro
- sulla base di questo si può decomporre
- poi si raffina, si espande, si integra

**Definizione 72 schema scheletro:** si individuano i concetti più importanti, ad esempio perché più citati o perché indicati esplicitamente come cruciali e li si organizza in un semplice schema concettuale

□

**esempio: Archivio fotografico**

Si vuole rappresentare la base di dati di un archivio fotografico distribuito in varie sedi. Le fotografie sono catalogate in base ad un catalogo di soggetti possibili, ciascun soggetto ha una propria chiave. Le foto hanno una dimensione ed uno stato di conservazione; per le foto a colori, è noto il tipo di stampa (chiaro o opaco). Le foto sono reperibili in archivi, di cui è noto il responsabile, l'indirizzo, il numero telefonico e l'orario di apertura. Le foto possono descrivere personaggi, luoghi o oggetti. I personaggi hanno un nome ed un sesso; alcuni sono deceduti. Per i personaggi politici, si indica il partito di appartenenza e l'eventuale carica governativa ricoperta. Per gli artisti, si indica la loro attività prevalente (pittura, scultura, ...). Quando le foto descrivono opere artistiche, è noto il nome dell'opera d'arte, l'artista che l'ha realizzata, il luogo dove l'opera risiede e l'anno di realizzazione. Quando le foto descrivono luoghi o oggetti, è noto nome e descrizione.

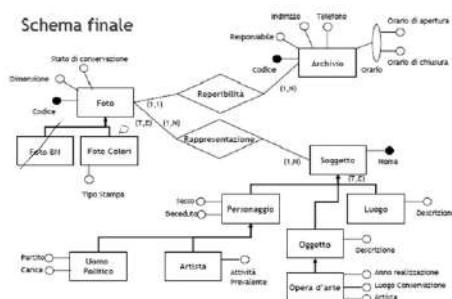
## Definizione dello scheletro

- Da una prima lettura del testo individuiamo che le entità fondamentali sono
    - ▶ Le fotografie
    - ▶ Gli archivi
    - ▶ I soggetti
  - Queste entità sono in relazione fra loro ed è facilmente individuabile il seguente scheletro di base



- Passiamo ora ad esaminare i singoli elementi dello scheletro...

## Schema finale



## 4.5 Progettazione logica

#### 4.5.1 Dal modello concettuale al modello logico

Il modello logico ha come obiettivo "tradurre" in modo automatico lo schema concettuale in uno schema logico che rappresenti gli stessi dati in maniera

corretta ed efficiente, passando prima per una riorganizzazione dello schema concettuale. Il tool di traduzione automatica prende quindi in ingresso:

- schema concettuale
- modello logico scelto
- informazioni sul carico applicativo (dimensione dei dati)

e da in uscita:

- schema logico

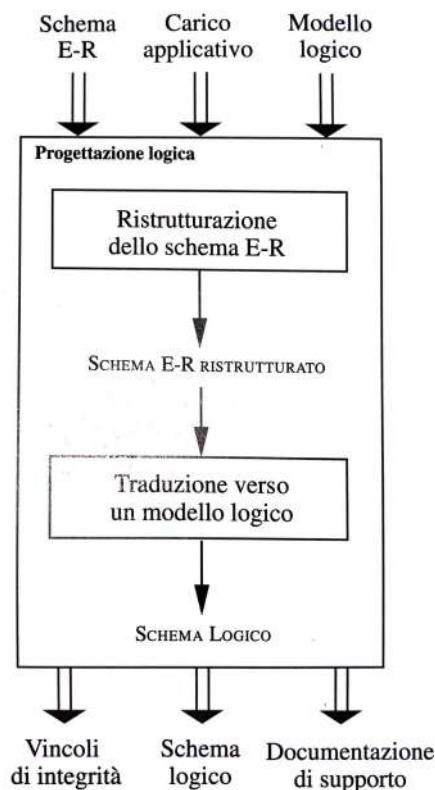


Figura 4.3: dati in ingresso schema ER e carico applicativo si ottiene lo schema ER ristrutturato che dato in ingresso con il modello logico scelto genera lo schema logico (durante questa seconda fase è possibile effettuare verifiche della qualità e ottimizzazioni mediante una tecnica (o più) basata sulle caratteristiche dello schema logico che nel caso del modello relazionale è detta normalizzazione)

-subsection della creazione di schemi con vincoli in sql-

#### 4.5.2 Analisi delle prestazioni su schema ER

Lo schema ER può essere modificato per ottimizzare alcuni indici di prestazione ( $\neq$  da prestazioni: queste dipendono anche da parametri fisici):

- costo di una operazione: viene valutato in termini di numero di occorrenze (di entità e associazioni) che mediamente vanno visitato per rispondere a una operazione sulla base di dati
- occupazione di memoria: viene valutato in termini dello spazio di memoria necessario per memorizzare i dati descritti dallo schema

Per studiare questi parametri è necessario conoscere:

- volume dei dati:
  - numero di occorrenze dello schema
  - dimensioni di ciascun attributo
- caratteristiche delle operazioni:
  - tipo dell'operazione (interattiva (richieste dagli utenti della base di dati) o batch (pianificate da altri programmi in particolari momenti della giornata))
  - frequenza (numero medio di esecuzioni in un certo intervallo di tempo)
  - dati coinvolti (entità o associazioni)

Il volume dei dati e le caratteristiche generali delle operazioni possono essere descritti facendo uso di tabelle:

Tavola dei volumi		
Concetto	Tipo	Volume
Sede	E	10
Dipartimento	E	80
Impiegato	E	2000
Progetto	E	500
Composizione	R	80
Afferenza	R	1900
Direzione	R	80
Partecipazione	R	6000

Tavola delle operazioni		
Operazione	Tipo	Frequenza
Op. 1	I	50 al giorno
Op. 2	I	100 al giorno
Op. 3	I	10 al giorno
Op. 4	B	2 a settimana

Figura 4.4: esempio

Avendo queste informazioni si può stimare il costo di una operazione costruendone la relativa tabella degli accessi basata a sua volta su uno schema di operazione (o navigazione).

**esempio dalle slide di valutazione costo per la seguente operazione:**

trovare tutti i dati di un impiegato X, del dipartimento nel quale lavora e dei progetti ai quali partecipa.

Si costruisce una tavola degli accessi basata su uno schema di navigazione:



Figura 4.5: nota: partecipazione e progetto hanno 3 accessi perché si è supposto che in media un impiegato lavora a 3 progetti

#### 4.5.3 Attività di ristrutturazione

**Definizione 73 Attività di ristrutturazione:**

1. eliminazione delle generalizzazioni
2. eliminazione degli attributi multivalore
3. analisi ed eventuale eliminazione delle ridondanze
4. partizionamento/accorpamenti entità e relationship

□

#### Eliminazione delle generalizzazioni

**Definizione 74 Le gerarchie nel modello relazionale (punto 1):** Il modello relazionale non può rappresentare direttamente le generalizzazioni. Le gerarchie vanno sostituite con entità e relazioni (semplificare la traduzione).

#### 3 possibilità

1. accorpamento delle figlie della generalizzazione nel genitore
2. accorpamento del genitore della generalizzazione nelle figlie
3. sostituzione della generalizzazione con relazioni

□

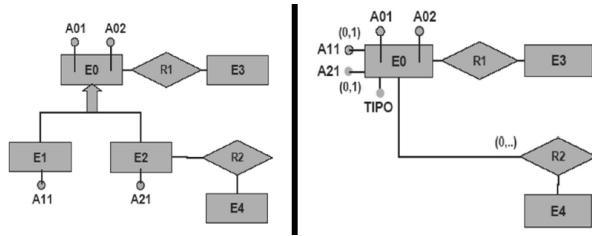


Figura 4.6: caso 1

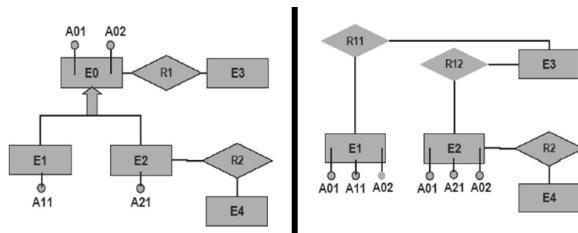


Figura 4.7: caso 2

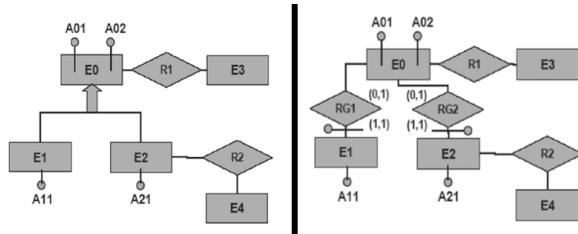


Figura 4.8: caso 3

La scelta fra le alternative si può fare basandosi sul numero e il tipo degli accessi fatti alle singole entità per eseguire le operazioni, si possono quindi seguire delle regole generali:

1. conviene se gli accessi al padre e alle figlie sono contestuali (fun fact: può essere sempre attuato se non ci sono associazioni tra entità di livelli diversi)
2. conviene se gli accessi sono solo alle figlie e sono distinti dall'una all'altra
3. conviene se si effettuano accessi separati alle entità figlie e al padre

## Eliminazione degli attributi multivalore

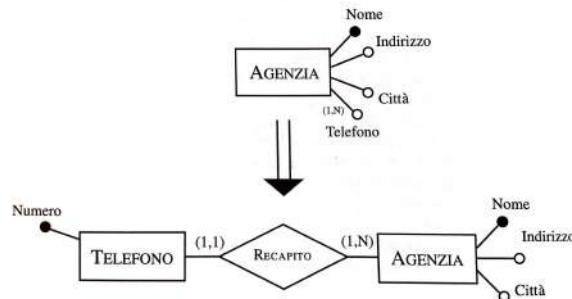


Figura 4.9: vanno ristrutturati solo quelli NON opzionali

**Definizione 75 Attributo derivabile:** (vedi definizione65) □

### Gestione delle ridondanze

**Definizione 76 Ridondanza:** Una ridondanza in uno schema E-R è una delle informazioni significativa, ma derivabile da altre. □

**Definizione 77 Forme di ridondanza in uno schema ER:** .

- attributi derivabili (da attributi della stessa entità o associazione, da attributi di altre entità o associazioni)
- associazioni derivabili dalla composizione di altre associazioni (presenza di cicli)

□

La presenza di un dato derivato presenta il vantaggio della riduzione degli accessi necessari per calcolare il dato derivato ma ha lo svantaggio di occupare memoria (spesso trascurabile) e la necessità di effettuare operazioni aggiuntive per mantenere il dato aggiornato. Si deve decidere se eliminare le ridondanze eventualmente presenti o mantenerle/inserirle in base ad una valutazione del costo delle operazioni.

Esempio analisi ridondanza:

Analisi di una ridondanza: schema E-R

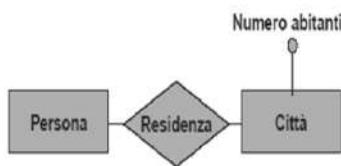


Tavola dei volumi e operazioni

Concetto	Tipo	Volume
Città	E	200
Persona	E	1000000
Residenza	R	1000000

- Operazione 1: memorizza una nuova persona con la relativa residenza, supponendo che la città sia già presente (500 volte al giorno)
- Operazione 2: stampa tutti i dati di una città (incluso il numero di abitanti) (2 volte al giorno)

Accessi in presenza di ridondanza

Operazione 1			
Concetto	Costrutto	Accessi	Tipo
Persona	Entità	1	S
Residenza	Relazione	1	S
Città	Entità	1	L
Città	Entità	1	S

Operazione 2			
Concetto	Costrutto	Accessi	Tipo
Città	Entità	1	L

Accessi in assenza di ridondanza

Operazione 1			
Concetto	Costrutto	Accessi	Tipo
Persona	Entità	1	S
Residenza	Relazione	1	S

Operazione 2			
Concetto	Costrutto	Accessi	Tipo
Città	Entità	1	L
Residenza	Relazione	5000	L

### Costi in presenza di ridondanza:

- operazione 1: 1500 accessi in scrittura e 500 accessi in lettura al giorno
- operazione 2: trascurabile
- contiamo doppi gli accessi in scrittura → totale di 3500 accessi al giorno

### Costi in assenza di ridondanza:

- operazione 1: 1000 accessi in scrittura al giorno
- operazione 2: 10 000 accessi in lettura al giorno
- contiamo doppi gli accessi in scrittura → totale di 12 000 accessi al giorno

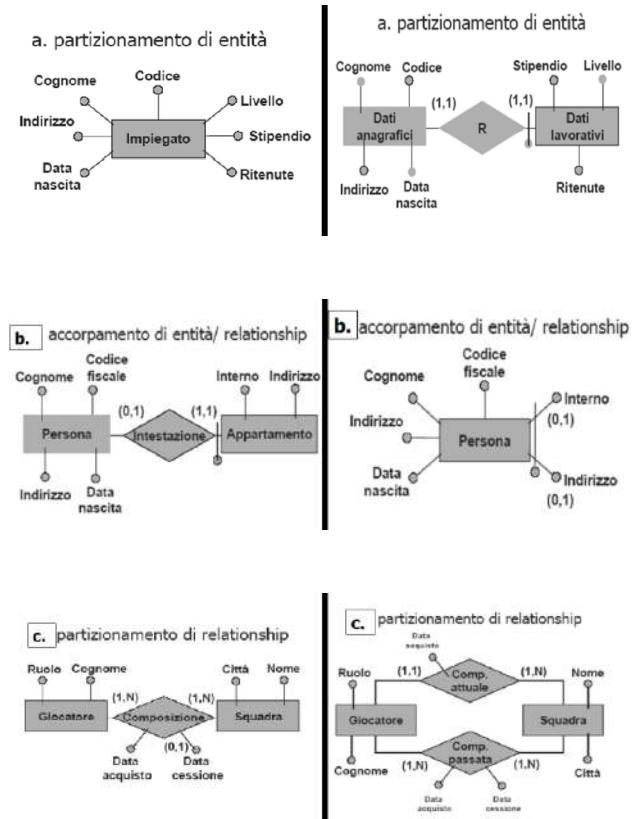
### Partizionamento o accorpamento di entità e associazioni

Ristrutturazioni effettuate per rendere più efficienti le operazioni in base al principio che:

- gli accessi si riducono (separando attributi di un concetto che vengono acceduti separatamente e raggruppando attributi di concetti diversi acceduti insieme)
- si considera sempre che ad ogni accesso si legge l'intera tupla

### Casi principali:

1. partizionamento di entità
2. accorpamento di entità/relationship
3. partizionamento di relationship



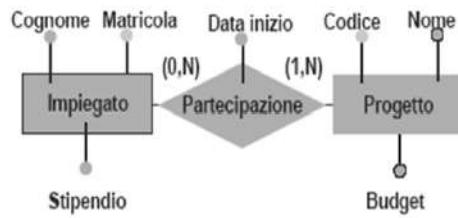
#### 4.5.4 Algoritmo di traduzione nel modello logico

La seconda fase della progettazione logica corrisponde a una traduzione tra modelli di dati diversi: a partire dallo schema ER ristrutturato si costruisce uno schema logico equivalente. L'idea di base della traduzione verso il modello relazionale è:

- le entità diventano relazioni sugli stessi attributi
- le associazioni diventano relazioni sugli identifieri delle entità coinvolte (più gli attributi propri)

#### Entità e associazioni molti a molti

**Nota 2:** si ottiene lo stesso schema di risultato in caso di relationship n-arie (sempre se le cardinalità max sono N)



La figura 4.5.4 diventa:

- Impiegato (Matricola, Cognome, Stipendio)
- Progetto (Codice, Nome, Budget)
- Partecipazione (Matricola, Codice, DataInizio)

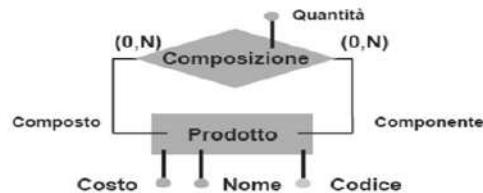
con vincoli di integrità referenziale tra

- Matricola in Partecipazione e (la chiave di) Impiegato
- Codice in Partecipazione e (la chiave di) Progetto

Utilizzando nomi più espressivi per gli attributi della chiave della relazione che rappresenta la relationship diventa:

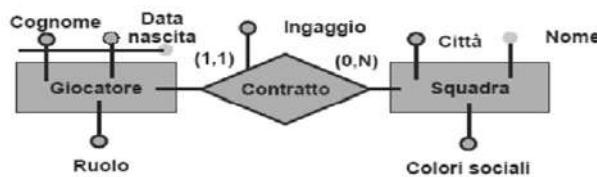
- Partecipazione (Impiegato, Progetto, DataInizio)

### Associazioni ricorsive



**Prodotto(Codice, Nome, Costo)**  
**Composizione (Composto, Componente, Quantità)**

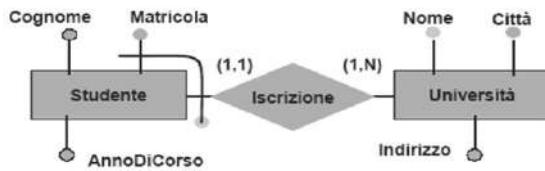
### Associazioni uno a molti



Poiché la chiave nella relazione Contratto è costituita solo dall'identificatore di Giocatore, data la cardinalità (1,1), allora Giocatore e Contratto hanno la stessa chiave ed è quindi possibile fonderli in un'unica relazione (perché esiste corrispondenza biunivoca tra le rispettive occorrenze). Si ha quindi:

Giocatore(Cognome, DataNascita, Ruolo, NomeSquadra, Ingaggio)  
 Squadra(Nome, Città, ColoriSociali)

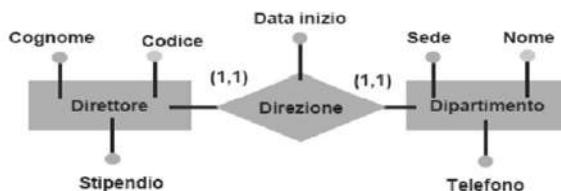
### Entità con identificatore esterno



Danno luogo a relazioni con chiavi che includono gli identificatori delle entità "identificanti" e hanno come schema relazionale:

Studente(Mtricola, NomeUniversità, Cognome, AnnoIscrizione)  
 Università(Nome, Città, Indirizzo)

### Associazioni uno a uno



Nel caso in cui entrambe le entità hanno partecipazione obbligatoria si hanno 2 possibilità simmetriche:

- 1)   
 Direttore(Codice, Cognome, Stipendio, DipartimentoDiretto, InizioDirezione)  
 Dipartimento(Nome, Telefono, Sede)
  - 2)   
 Direttore(Codice, Cognome, Stipendio)  
 Dipartimento(Nome, Telefono, Sede, Direttore, InizioDirezione)
- Nel caso in cui una o entrambe le partecipazioni siano opzionali:



In caso di una partecipazione opzionale (per esempio di Impiegato):

Impiegato(Codice, Cognome, Stipendio)  
 Dipartimento(Nome, Telefono, Sede, Direttore, InizioDirezione)

Nel caso in cui entrambe le partecipazioni siano opzionali si ha:

Impiegato(Codice, Cognome, Stipendio)  
 Dipartimento(Nome, Telefono, Sede)  
 Direzione(Direttore, Dipartimento, DataInizioDirezione)

**Nota 3:** *nell'ultimo caso, in modo alternativo ma equivalente, la chiave può essere, invece che "Direttore", "Dipartimento"*

## 4.6 Normalizzazione

### 4.6.1 Ridondanze e anomalie

Impiegato	Stipendio	Progetto	Bilancio	Funzione
Rossi	20 000	Marte	2000	tecnico
Verdi	35 000	Giove	15 000	progettista
Verdi	35 000	Venere	15 000	progettista
Neri	55 000	Venere	15 000	direttore
Neri	55 000	Giove	15 000	consulente
Neri	55 000	Marte	2000	consulente
Mori	48 000	Marte	2000	direttore
Mori	48 000	Venere	15 000	progettista
Bianchi	48 000	Venere	15 000	progettista
Bianchi	48 000	Giove	15 000	direttore

Figura 4.10:

Considerando la figura 4.6.1 tale relazione ha come chiave Impiegato e Progetto. Si possono inoltre osservare le seguenti proprietà sulle tuple:

1. lo stipendio di ogni impiegato è unico ed è funzione del solo impiegato, indipendentemente dal progetto a cui partecipa
2. il bilancio di ogni progetto è unico e dipende solo da esso, indipendentemente dagli impiegati che vi partecipano

Questi fatti hanno conseguenze sul contenuto della relazione e sulle operazioni che si possono effettuare su di essa. Riguardo la prima proprietà si osserva che:

- il valore dello stipendio di ogni impiegato è ripetuto in tutte le tuple riferite ad esso → *ridondanza*

- se lo stipendio di un impiegato varia tale valore va modificato in tutte le tuple corrispondenti (solo così la dipendenza continua a valere), questo comporta la necessità di effettuare più modifiche contemporaneamente → *anomalia di aggiornamento*
- se un impiegato interrompe la partecipazione ad ogni progetto a cui partecipava ma rimane nell'azienda, non è possibile conservare traccia del suo nome e del suo stipendio → *anomalia di cancellazione*
- se si hanno informazioni su un nuovo impiegato non è possibile inserirle finché non viene assegnato ad un progetto → *anomalia di inserimento*

Il motivo per cui si sono presentati questi inconvenienti è che è stata utilizzata una sola relazione per rappresentare informazioni eterogenee.

#### 4.6.2 Dipendenze funzionali

Sono particolari vincoli di integrità per il modello relazionale che descrivono legami di tipo funzionale tra gli attributi di una relazione. Facendo ancora riferimento all'esempio precedente si può allora dire che il valore dell'attributo Impiegato determina il valore dell'attributo stipendio, o meglio che esiste una funzione che associa a ogni elemento del dominio dell'attributo Impiegato un solo elemento del dominio dell'attributo Stipendio (stesso discorso per Progetto e Bilancio). Quindi:

**Definizione 78 Dipendenze funzionali (FD):** esprime un legame semantico tra due gruppi di attributi di uno schema di relazione R.

Data quindi la relazione r su R(X) e due sottoinsiemi non vuoti Y e Z di X, esiste in r una dipendenza funzionale da Y a Z se, per ogni coppia di tuple  $t_1$  e  $t_2$  di r aventi gli stessi valori sugli attributi Y, risulta che  $t_1$  e  $t_2$  hanno gli stessi valori sugli attributi Z. Notazione:

$$Y \rightarrow Z$$

(N.B. non è detto che esista  $Z \rightarrow Y$ )

Note:

Una FD è quindi una proprietà di R, non di un particolare stato valido r di R. Una FD non può essere dedotta a partire da uno stato valido r, ma deve essere definita esplicitamente da qualcuno che conosce la semantica degli attributi di R. □

**Definizione 79 FD non banale:** una FD  $Y \rightarrow A$  si dice non banale se, dato un attributo A, A non compare tra gli attributi di Y □

**Definizione 80 FD e il vincolo di chiave:** se prendiamo una chiave K di una relazione r si può facilmente verificare che esiste una dipendenza funzionale tra K e ogni altro attributo dello schema di r (questo per definizione

di vincolo di chiave). Possiamo quindi dire che il vincolo di dipendenza funzionale generalizza il vincolo di chiave. Più precisamente una FD  $Y \rightarrow Z$  su uno schema  $R(X)$  degenera nel vincolo di chiave se l'unione di  $Y$  e  $Z$  è pari a  $X$ . In questo caso  $Y$  è superchiave per lo schema  $R(X)$   $\square$

#### 4.6.3 Forma normale di Boyce Codd

##### Definizione di forma normale di Boyce Codd

L'idea di base è che si possono introdurre delle proprietà dette forme normali definite con riferimento alle FD e che sono soddisfatte quando non ci sono anomalie.

Nel nostro esempio (fig. 4.6.1) le due proprietà causa di anomalie corrispondono esattamente ad attributi coinvolti in FD.

- La proprietà 1 implica il soddisfacimento della dipendenza funzionale  $Impiegato \rightarrow Stipendio$
- La proprietà 2 corrisponde alla FD  $Progetto \rightarrow Bilancio$

Inoltre l'attributo Funzione indica per ciascuna tupla il ruolo svolto dell'impiegato nel progetto (ruolo che è unico per ciascuna coppia impiegato-progetto). Quindi si ha una FD:

- La proprietà "in ciascun progetto, ciascuno degli impiegati coinvolti può svolgere una e una sola funzione" implica il soddisfacimento della FD  $Impiegato \rightarrow Funzione$  (conseguenza del fatto che Impiegato e Progetto sono chiave)

Quindi la prima proprietà genera ridondanze e anomalie come già visto, in modo analogo lo fa anche la seconda mentre la terza non genera ridondanze (essendo Impiegato-Progetto la chiave) e non genera (dal punto di vista concettuale) anomalie: ogni impiegato ha un solo stipendio, ogni progetto ha un solo bilancio.

Da questo possiamo quindi dedurre che le ridondanze e le anomalie sono causate da FD  $X \rightarrow A$  che permettono la presenza di più tuple tra loro uguali sugli attributi  $X$  (ovvero se  $X$  non contiene una chiave). Precisiamo queste idee per mezzo della forma normale più importante:

**Definizione 81 Forma normale di Boyce-Codd (BCNF):** una relazione  $r$  è in forma normale di Boyce-Codd se, per ogni dipendenza funzionale (non banale)  $X \rightarrow Y$  definita su di essa,  $X$  è superchiave di  $r$ .

(Questa forma normale richiede che i concetti in una relazione siano omogenei, cioè che tutte le proprietà siano direttamente associate alla chiave).

Se un insieme di dipendenze per  $R$  non è in BCNF allora in  $F$  c'è almeno una dipendenza  $X \rightarrow Y$  non banale con  $X$  non superchiave di  $R$ .

**teorema:** dato uno schema R e un insieme F di FD, se F non contiene alcuna  $X \rightarrow Y$  non banale con X non superchiave di R, allora neanche  $F^+$  la contiene.  $\square$

### Decomposizione in forma normale di Boyce e Codd

Data una relazione che non soddisfa la BCNF è possibile sostituirla con due o più relazioni normalizzate attraverso un processo detto normalizzazione.

**Definizione 82 Normalizzazione:** è la procedura che permette di portare uno schema relazionale in una determinata forma normale. La normalizzazione può essere utilizzata come tecnica di verifica dei risultati della progettazione, non costituisce una metodologia di progetto.  $\square$

Questo processo si fonda su un semplice criterio: se una relazione rappresenta più concetti indipendenti allora va decomposta in relazioni più piccole, una per ogni concetto.

Impiegato	Stipendio	Progetto	Bilancio
Rossi	20 000	Marte	2000
Verdi	35 000	Giove	15 000
Neri	55 000	Venere	15 000
Mori	48 000		
Bianchi	48 000		

Impiegato	Progetto	Funzione
Rossi	Marte	tecnico
Verdi	Giove	progettista
Verdi	Venere	progettista
Neri	Venere	direttore
Neri	Giove	consulente
Neri	Marte	consulente
Mori	Marte	direttore
Mori	Venere	progettista
Bianchi	Venere	progettista
Bianchi	Giove	direttore

Figura 4.11: relazione della figura 4.6.1 sostituita con 3 relazioni ottenute tramite proiezioni sugli attributi corrispettivi delle 3 proprietà descritte in precedenza (sono state eliminate anomalie e ridondanze: le 3 relazioni sono in BCNF)

#### 4.6.4 Proprietà delle decomposizioni

Esamina del concetto di decomposizione.

### Decomposizione senza perdita

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venre	Milano

La tabella soddisfa le FD  $Impiegato \rightarrow Sede$  e  $Progetto \rightarrow Sede$  (ciascun impiegato una sede e ciascun progetto una sede). Si osserva inoltre che un impiegato può partecipare a più progetti anche se, sulla base delle FD, questi devono essere tutti progetti assegnati alla sede cui afferisce tale impiegato. Operando quindi come fatto nell'esempio precedente si andrebbe a separare la relazione in due sulla base delle rispettive FD e ottendendo quindi:

Impiegato	Sede	progetto	Sede
Rossi	Roma	Marte	Roma
Verdi	Milano	Giove	Milano
Neri	Milano	Saturno	Milano
		Venere	Milano

Provando però a ricostruire la relazione iniziale tramite un join si ottiene:

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venre	Milano
Verdi	Saturno	Milano
Neri	Giove	Milano

Il risultato del join ha quindi le due tuple finali in più rispetto alla relazione originale le quali possiamo quindi chiamare "spurie".

**Definizione 83 Decomposizione denza perdita:** r si decomponete senza perdita su  $X_1$  e  $X_2$  se il join sulle due proiezioni è uguale a r stessa. (è irrinunciabile che una decomposizione a fini di normalizzazione sia senza perdita)  $\square$

## Condizione sufficiente per assicurare il join senza perdita

- **Teorema.** Data la decomposizione di  $R(X)$  in due relazioni  $R1(X1)$  e  $R2(X2)$ , con  $X1 \cup X2 = X$  e  $X0 = X1 \cap X2$ , se
  - $X0 \rightarrow X1$  è in  $F^+$  oppure  $X0 \rightarrow X2$  è in  $F^-$
  - la decomposizione è senza perdita nel join
- Ovvero: gli attributi comuni alle due relazioni devono essere chiave in una delle due tabelle.
- N.B. Se non è verificata la condizione, la decomposizione può essere comunque senza perdita sul join.

### Conservazione delle dipendenze

Operando una decomposizione senza perdite sulla relazione 4.6.4 si potrebbe pensare di sfruttare solo la dipendenza  $Impiegato \rightarrow Sede$  ottenendo:

Impiegato	Sede	Impiegato	Progetto
Rossi	Roma	Rossi	Marte
Verdi	Milano	Verdi	Giove
Neri	Milano	Verdi	Venere
		Neri	Saturno
		Neri	Venere

Questo risultato risulta però inefficace in caso di aggiunta, per esempio, della partecipazione di Neri al progetto Marte; aggiunta che nella relazione originale verrebbe impedita in quanto Neri opera a Milano. Per evitare queste violazioni delle dipendenze è necessario allora stabilire che:

**Definizione 84 conservazione delle dipendenze:** in ogni decomposizione ciascuna delle FD dello schema originario dovrebbe coinvolgere attributi che compaiono tutti insieme in uno degli schemi decomposti.  $\square$

### Qualità delle decomposizioni

Le decomposizioni dovrebbero sempre soddisfare le proprietà di decomposizione senza perdita e conservazione delle dipendenze e contenere relazioni in forma normale (BCNF). Valutiamo quindi come accettabili da ora in poi solo quelle che soddisfano tali proprietà.

#### 4.6.5 Terza forma normale

Non sempre la BCNF è raggiungibile.

Dirigente	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Marte	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

Sulla relazione in foto sono definite le seguenti FD:

- $\text{Dirigente} \rightarrow \text{Sede}$  :ogni dirigente opera presso una sede
- $\text{Progetto Sede} \rightarrow \text{Dirigente}$  : ogni progetto ha più dirigenti che ne sono responsabili ma in diverse sedi, ogni dirigente può essere responsabile di più progetti però per ogni sede un progetto ha un solo responsabile

Tale relazione non è in BCNF ( $\text{Dirigente} \rightarrow \text{Sede}$  non è superchiave, e la seconda FD in quanto coinvolge tutti gli attributi non è in grado di essere conservata da alcuna decomposizione).

### Definizione di Terza forma normale

**Definizione 85 Terza forma normale:** una relazione è in 3NF se per ogni FD (non banale)  $X \rightarrow A$  definita su di essa è verificata almeno una delle seguenti condizioni:

- X contiene una chiave K di r
- A appartiene ad almeno una chiave di r

□

La relazione in figura 4.6.5 è quindi in 3NF in quanto

$\text{Progetto Sede} \rightarrow \text{Dirigente}$  ha come primo membro una chiave della r, e in quanto  $\text{Dirigente} \rightarrow \text{Sede}$  ha un unico attributo al secondo membro parte della chiave ProgettoSede. (La ridondanza che questa relazione presenta, essendo infatti in 3NF e non in BCNF, è che ogni volta che un dirigente compare in una tupla viene ripetuta la sede in cui opera, tuttavia conserva tutte le dipendenze).

**Nota 4 :** importante!!!: ricapitolando si può capire che la 3NF è meno forte della BCNF (e quindi non offre le solite garanzie di qualità) ma è sempre ottenibile: è dimostrabile che una qualunque relazione che non soddisfa la 3NF è certamente decomponibile senza perdita e con conservazione delle dipendenze in relazioni in 3NF

### Decomposizione in terza forma normale

Una relazione che non soddisfa la terza forma normale si decomponete in relazioni ottenute per proiezione su gli attributi corrispondenti alle FD, con l'unica accortezza di mantenere sempre una relazione che contiene una chiave della relazione originaria.

Nota: una decomposizione tesa a ottenere la 3NF produce nella maggior parte dei casi schemi in BCNF, in particolare se una r ha solo una chiave allora le due forme normali coincidono.

**Nota 5:** Per la 3FN il miglior algoritmo deterministico conosciuto, nel caso peggiore ha complessità esponenziale. Per ogni dipendenza di R si guarda se a SX c'è una chiave o superchiave, e per ognuna di esse si vanno a guardare a DX tutti i possibili sottoinsiemi delle chiavi.

### Altre forme normali

**Definizione 86 prima forma normale:** gli attributi delle relazioni sono definiti su valori atomici e non su valori complessi (quali insiemi o relazioni).

□

**Definizione 87 seconda forma normale:** una relazione è in seconda forma normale se su di essa non sono definite dipendenze parziali, cioè dipendenze fra un sottoinsieme proprio della chiave e altri attributi.

□

Impiegato	Categoria	Stipendio
Neri	3	30000
Verdi	3	30000
Rossi	4	50000
Mori	4	50000
Bianchi	5	72000

Ad esempio la tabella, che soddisfa le FD  $Impiegato \rightarrow Categoria$  e  $Categoria \rightarrow Stipendio$  e quindi viola la 3NF in quanto Categoria non è chiave, è in seconda forma normale perché Stipendio dipende comunque (attraverso Categoria) dall'intera chiave Impiegato.

### Normalizzazione e scelta degli attributi

La non raggiungibilità del BCNF è spesso dovuta a un'analisi non sufficientemente accurata dell'applicazione (controllare sempre gli attributi).

#### 4.6.6 Teoria delle dipendenze e normalizzazione

Problema posto: data una r e un insieme di FD generare una decomposizione della r che contenga solo f in forma normale e soddisfi la decomposizione senza

perdite e con conservazione delle dipendenze (utilizziamo quindi la 3NF che ci da la certezza di essere raggiungibile piuttosto che la BCNF (non è raggiungibile per certo)).

procedimento: definire una relazione per chiascun gruppo di dipendenze fra loro strettamente correlate.

### Implicazione di dipendenze funzionali

(consideriamo per semplicità anche le dipendenze banali)

**Definizione 88 Implicazione:** sia  $F$  un insieme di dipendenze funzionali definite su  $R(Z)$  e sia  $X \rightarrow Y$ :

- si dice che  $F$  implica  $X \rightarrow Y$  ( $F \Rightarrow X \rightarrow Y$ ) se, per ogni istanza  $r$  di  $R$  che verifica (soddisfa) tutte le dipendenze in  $F$ , risulta verificata (soddisfa) anche  $X \rightarrow Y$
- si dice anche che  $X \rightarrow Y$  è implicata da  $F$

□

Osservando la tabella 4.6.5 si osserva come le dipendenze funzionali *Impiegato*  $\rightarrow$  *Categoria* e *Categoria*  $\rightarrow$  *Stipendio* implicano *Impiegato*  $\rightarrow$  *Stipendio*: ogni relazione che soddisfa le prime due soddisfa anche la terza. Per verificare l'implicazione bisogna introdurre il concetto di:

**Definizione 89 Chiusura:** dato un insieme di dipendenze funzionali  $F$  definite su  $R(Z)$  e dato  $X$  insieme di attributi contenuti in  $Z$ , la chiusura di  $X$  rispetto a  $F$  è l'insieme degli attributi che dipendono funzionalmente da  $X$ :

$$X_F^+ = \{A \mid A \in Z : F \Rightarrow X \rightarrow A\}$$

L'insieme  $X_F^+$  risulta molto utile se vogliamo, ad esempio, vedere se  $X \rightarrow A$  è implicata da  $F$ : basta vedere se  $A$  appartiene a  $X_F^+$  (a patto di saperlo calcolare).

**Nota 6:** sulle slide la notazione è  $(X)^+F$  e si legge "chiusura di  $X$  rispetto a  $F$ : è definita come l'insieme di attributi della relazione che dipendono da  $X$  secondo  $F$

□

**Definizione 90 Algoritmo per il calcolo di  $X_F^+$ :**

**Nota 7:** ha complessità lineare e quindi si usa per il calcolo della chiusura

Input:  $X$  insieme di attributi e  $F$

Output: insieme  $X_p$  di attributi

1. inizializziamo  $X_p$  con l'insieme  $X$
2. esaminiamo le dipendenze in  $F$ , se esiste una dip.  $Y \rightarrow A$  con  $Y \subseteq X_p$  e  $A \notin X_p$  allora aggiungiamo  $A$  a  $X_p$

3. ripetiamo il passo 2 fino al momento in cui non vi sono ulteriori attributi che possono essere aggiunti a  $X^p$

□

## Esempio 6

- Supponiamo di avere  
 $F = \{A \rightarrow B, BC \rightarrow D, B \rightarrow E, E \rightarrow C\}$   
e calcoliamo  $A^+$ , ovvero l'insieme di attributi  
che dipendono da  $A$ 
  - $A^+ = A$
  - $A^+ = AB$  poiché  $A \rightarrow B$  e  $A \subseteq A^+$
  - $A^+ = ABE$  poiché  $B \rightarrow E$  e  $B \subseteq A^+$
  - $A^+ = ABEC$  poiché  $E \rightarrow C$  e  $E \subseteq A^+$
  - $A^+ = ABECD$  poiché  $BC \rightarrow D$  e  $BC \subseteq A^+$
- Quindi da  $A$  dipendono tutti gli attributi dello schema,  
ovvero  $A$  è superchiave (e anche chiave)!

Figura 4.12: per la chiave: 1) parto dagli attributi che sono solo a sinistra e  
calcolo la chiusura 2) se tutti gli attributi di  $F$  sono nella chiusura allora ho  
la chiave

## Esempio 5 (cont)

- $F = \{A \rightarrow C, AC \rightarrow D, E \rightarrow AD, E \rightarrow H\}$
  - $G = \{A \rightarrow CD, E \rightarrow AH\}$
- Verificare se  $F$  e  $G$  sono equivalenti
- Invece di verificare se  $X \rightarrow Y$  in  $F$  è anche in  $G^+$ ,  
verifico se  $Y \subseteq (X)^{+G}$  (chiusura di  $X$  rispetto a  $G$ ), e  
viceversa per ogni FD in  $G$
  - per  $A \rightarrow C$  risulta  $(A)^{+G} = ACD$ ; o.k.  $C \subseteq (A)^{+G}$
  - per  $AC \rightarrow D$  risulta  $(AC)^{+G} = ACD$ ; o.k.  $D \subseteq (AC)^{+G}$
  - per  $E \rightarrow AD$  risulta  $(E)^{+G} = EADCH$ ; o.k.  $AD \subseteq (E)^{+G}$
  - per  $E \rightarrow H$  risulta  $(E)^{+G} = EHADC$ ; o.k.  $H \subseteq (E)^{+G}$

E viceversa per ogni FD in  $G$

$F$  e  $G$  sono equivalenti se:

- per ogni  $X \rightarrow Y \in F, Y \in X^+$  secondo  $G$ , e,
- per ogni  $Z \rightarrow W \in G, W \in Z^+$  secondo  $F$

**Definizione 91 FD semplici:** a destra hanno un solo attributo □

**Definizione 92 Attributi estranei:** es.  $F = \{AB \rightarrow C, A \rightarrow B\}$  calcolato  $A+$  e  $B+$  si nota che C dipende solo da A (in quanto B dipende da A): B è allora estraneo:  $F = \{A \rightarrow C, A \rightarrow B\}$

In generale: data  $AX \rightarrow B$  A è estraneo se  $X+$  include B □

**Definizione 93 Ridondanza di attributi:** .

- a destra:  $X \rightarrow YZ \Rightarrow X \rightarrow Y, X \rightarrow W$
- a sinistra:  $X \rightarrow Y, Y \rightarrow Z, XZ \rightarrow W \Rightarrow \dots, X \rightarrow W$

□

**Definizione 94 Ridondanza di FD:**  $X \rightarrow Y$  in F è ridondante se, dopo averla rimossa,  $Y \subseteq X^{+(F)}$  = se le dipendenze rimanenti erano già sufficienti per determinare Y da X: es.  $X \rightarrow Y, Y \rightarrow Z, X \rightarrow Z$  l'ultima è ridondante

□

**Definizione 95 Algoritmo per FD ridondanti:** per stabilire se una Fd è ridondante (es  $X \rightarrow A$ ):

1. la eliminiamo da F
2. calcoliamo  $X+$  e verifichiamo se include A (nel caso A è inclusa allora  $X \rightarrow A$  è ridondante)

□

**Definizione 96 FD e chiave:** un insieme di attributi K è chiave per uno schema di relazione R(U) su cui è definito F se F implica  $K \rightarrow U$  □

(dalle slide:)

**Definizione 97 Chiusura:** dato un insieme di dipendenze funzionali F definite su R(Z), la chiusura di F è l'insieme di tutte le dipendenze funzionali implicate da F

$$F^+ = \{X \rightarrow Y | F \Rightarrow X \rightarrow Y\}$$

Dato un insieme di dipendenze funzionali F definite su R(Z), un'istanza r di R che soddisfa F soddisfa anche  $F^+$  □

la definizione di implicazione non è direttamente utilizzabile nella pratica, essa prevede, infatti, una quantificazione universale sulle istanze della base di dati ("per ogni istanza r.."). Le regole che permettono di derivare costruttivamente tutte le dipendenze funzionali che sono implicate da un dato insieme iniziale sono state fornite da Armstrong (1974); tali regole sono corrette e complete, cioè permettono di ottenere tutte e sole le dipendenze in  $F^+$

**Definizione 98 Regole di inferenza di Armstrong:** .

1. Riflessività: se  $Y \subseteq X$ , allora  $X \rightarrow Y$

2. Additività (o arricchimento): se  $X \rightarrow Y$ , allora  $XZ \rightarrow YZ$  per qualunque Z
3. Transitività: se  $X \rightarrow Y$  e  $Y \rightarrow Z$ , allora  $X \rightarrow Z$

□

### Proprietà delle regole di Armstrong

- Correttezza: le regole di inferenza di Armstrong sono corrette, cioè applicandole ad un insieme F di dipendenze funzionali, si ottengono solo dipendenze logicamente implicate da F.
- Completezza: le regole di inferenza di Armstrong sono complete, cioè applicandole ad un insieme F di dipendenze funzionali, si ottengono tutte le dipendenze logicamente implicate da F.
- Minimalità: le regole di inferenza di Armstrong sono minimali, cioè ignorando anche una sola di esse, l'insieme di regole che rimangono non è più completo.

### Definizione 99 Regole derivate di Armstrong: .

1. regola di unione

$$\{X \rightarrow Y, X \rightarrow Z\} \Rightarrow X \rightarrow YZ$$

2. regola di pseudo transitività (o aggiunta sinistra)

$$\{X \rightarrow Y, WY \rightarrow Z\} \Rightarrow XW \rightarrow Z$$

3. regola di decomposizione: se

$$Z \subseteq Y, X \rightarrow Y \Rightarrow XW \rightarrow Z$$

□

### Coperture di insiemi di FD

**Definizione 100 Equivalenza:** dato un insieme di FD F, è molto utile poter determinare se un insieme di FD G sia equivalente ad F: F e G sono equivalenti se F implica ciascuna dipendenza in G e viceversa. Se F e G sono equivalenti allora ognuno è **copertura** dell'altro.

def slide: F e G sono equivalenti se  $F^+ = G^+$ , ovvero, per ogni  $X \rightarrow Y \in F$ , deve essere  $X \rightarrow Y \in G^+$  e, viceversa, per ogni  $X \rightarrow Y \in G$ , deve essere  $X \rightarrow Y \in F^+$ .

(Anche la verifica di questa proprietà usa la chiusura transitiva dell'insieme delle dipendenze) □

Questo permette di utilizzare dato un insieme di dipendenze un suo equivalente ma più semplice (meno dip. o meno attributi). Tra gli insiemi semplici consideriamo:

- non ridondante: se non esiste dipendenza  $f \in F$  tale che  $F - \{f\} \Rightarrow f$
- ridotto: se è non ridondante e non esiste un insieme  $F'$  equivalente a  $F$  ottenuto eliminando attributi dai primi membri di una o più dipendenze di  $F$ .

esempio: dati

$$F_1 = \{A \rightarrow B, AB \rightarrow C, A \rightarrow C\}$$

$$F_2 = \{A \rightarrow B, AB \rightarrow C\}$$

$$F_3 = \{A \rightarrow B, A \rightarrow C\}$$

osserviamo che:

- $F_1$  è ridondante perché  $\{A \rightarrow B, AB \rightarrow C\}$  implica  $A \rightarrow C$ ;  $F_1$  è equivalente a  $F_2$
- $F_2$  non è ridondante ma non è ridotto perché  $B$  può essere eliminato dal primo membro della seconda dipendenza:  $F_2$  è equivalente a  $F_3$
- $F_3$  è ridotto

Per trovare una copertura non ridondante è sufficiente eliminare dall'insieme le dipendenze implicate da altre.

Per trovare una copertura ridotta bisogna sostituire l'insieme con quello equivalente che ha tutti i secondi membri costituiti da singoli attributi, eliminare poi le dipendenze ridondanti e infine verificare per ogni dipendenza se esistono attributi eliminabili dal primo membro.

**Definizione 101 Copertura minimale: definizione preliminare:** un insieme  $F$  di FD è minimale se nella parte destra di ogni FD c'è un solo attributo e non si possono togliere attributi nella parte sinistra di qualche FD né alcuna FD senza perdere l'equivalenza dell'insieme ottenuto  $F$ .

**definizione** Una copertura minimale di un insieme  $F$  è quindi un insieme equivalente a  $F$  ma di complessità minore

**Nota 8:** una copertura minimale in genere non è unica

□

**Definizione 102 Algoritmo copertura minimale:** .

1. portare ogni FD in FD semplice
2. rimuovere gli attributi estranei
3. rimuovere le FD ridondanti

□

## Sintesi schemi in terza forma normale

### Metodologia di decomposizione (1)

1. Data  $R$  ed  $F$  minimale, si usa
  - Decomponi  $(R, F)$  ottenendo gli schemi  $R_1(X_1), R_2(X_2), \dots, R_n(X_n)$  ciascuno con dipendenze  $F_{X_i}$
2. Sia  $N$  l'insieme di dipendenze non preservate in  $R_1, R_2, \dots, R_n$ , cioè non incluse nella chiusura dell'unione dei vari  $F_{X_i}$ 
  - Per ogni dipendenza  $X \rightarrow A$  in  $N$ , aggiungiamo lo schema relazionale  $X \rightarrow A$  con le dipendenze funzionali relative a  $XA$

La prima metodologia garantisce come primo passo l'assenza di perdita sul join e poi conserva le dipendenze

### Altra metodologia (2)

- Si deriva la copertura minima  $G$  di  $F$ .
- Si raggruppano le dipendenze in  $G$  in sottoinsiemi tali che ad ogni sottoinsieme  $G_i$  appartengono le dipendenze i cui membri sinistri hanno la stessa chiusura: i.e.  $X \rightarrow A$  e  $Y \rightarrow B$  appartengono a  $G_i$ , se  $X=Y$  secondo  $G$ .
- Si partizionano gli attributi  $U$  nei sottoinsiemi  $U_i$  individuati dai sottoinsiemi  $G_i$  del passo precedente. Se un sottoinsieme è contenuto in un altro si elimina.
- Si crea una relazione  $R_i(U_i)$  per ciascun sottoinsieme  $U_i$ , con associate le dipendenze  $G_i$ .
- Si aggiunge una relazione per gli attributi che non sono coinvolti in alcuna FD
- Se non c'è già una relazione che contenga una chiave della relazione originaria, si aggiunge

La seconda conserva le dipendenze e poi risolve l'eventuale perdita sul join

Figura 4.13: Decomposizione in 3NF

### 4.6.7 progettazione e normalizzazione

La verifica della correttezza delle relazioni è utile a trovare imprecisioni o errori avvenuti in fase di progettazione.

#### Verifiche di normalizzazione su entità

È sufficiente considerare le FD che sussistono fra gli attributi dell'entità e verificare che ciascuna di esse abbia come primo membro l'identificatore o lo contenga.

La decomposizione può avvenire con diretto riferimento alle dipendenze oppure, più semplicemente, ragionando qualitativamente sui concetti rappresentati dall'entità insieme a quelli che drivano dalle dipendenze funzionali.

#### Verifiche di normalizzazione su associazioni

Considerando che:

- l'insieme delle occorrenze di ciascuna associazione è una relazione: è quindi possibile applicare direttamente i concetti connessi con le forme normali
- i domini su cui tale relazione è definita sono gli insiemi delle occorrenze delle entità coinvolte

Di conseguenza per verificare il soddisfacimento della terza forma normale è necessario individuare le FD che sussistono fra le entità coinvolte.

**nota:** ogni relazione binaria è in forma normale quindi la verifica va effettuata solo sulle associazioni non binarie

## Capitolo 5

# Tecnologie della base di dati

### 5.1 Organizzazione fisica e gestione delle interrogazioni

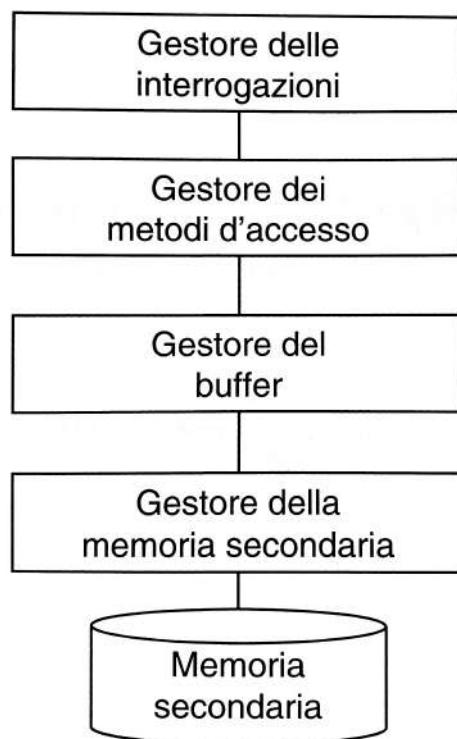


Figura 5.1: componenti di un DBMS coinvolti nella gestione di interrogazioni e nell'accesso alla memoria secondaria

### 5.1.1 Memoria principale, secondaria e gestione del buffer

Necessità di gestire dati in memoria secondaria:

1. spazio non sufficiente in memoria principale per una intera base di dati.
2. persistenza: le memorie centrali sono volatili mentre una base di dati deve essere conservata anche quando il sistema è spento o si guasta.

#### Caratteristiche memoria secondaria

- "secondaria" perché non è direttamente utilizzabile dai programmi (i dati per essere utilizzati devono passare in memoria principale).
- I dati sono organizzati in blocchi di dimensione (di solito) fissa (da alcuni kb a decine di kb).

Le uniche operazioni possibili su un disco sono lettura o scrittura di un intero blocco (quindi anche accedere a un solo bit ha lo stesso costo di accedere a un blocco). Per questo motivo nelle applicazioni che coinvolgono basi di dati si considerano solo i costi di accesso in memoria secondaria (essendo quelli in centrale trascurabili a confronto). Componenti del tempo di accesso a un blocco:

1. posizionamento della testina (nei vecchi dischi fisici)
2. latenza (il tempo tra la richiesta e il passaggio del blocco sotto la testina)
3. trasferimento (tempo necessario a effettuare scrittura e/o lettura)

(con le memorie allo stato solido (quelle moderne) i tempi di posizionamento e latenza non sono rilevanti)

#### Gestione dei buffer

**Definizione 103 Buffer:** è una grande zona di memoria centrale gestita dal DBMS in modo condiviso per tutte le applicazioni che realizza l'interazione tra memoria centrale e secondaria. (buffer sempre più vasti permettono di evitare accessi alla memoria secondaria quando un dato viene utilizzato spesso in tempi ravvicinati, questo rende la gestione del buffer essenziale nel funzionamento della base di dati in particolare delle sue prestazioni) □

Il buffer è organizzato in pagine di dimensioni pari o multiple (=un numero intero) di quelle di blocchi di memoria secondaria (consideriamo una pagina corrisponde a un blocco in modo che un caricamento di una pagina corrisponde a una lettura in mem. sec. mentre un salvataggio di pagina corrisponde a una scrittura).

**Definizione 104 gestore del buffer:** si occupa del caricamento e dello scaricamento delle pagine dalla mem. centrale alla memoria di massa (=memoria secondaria)  $\square$

Intuitivamente lo possiamo pensare come un modulo che riceve (dai programmi) richieste per la lettura o la scrittura di blocchi e le esegue interagendo col Gestore della memoria secondaria quando è indispensabile, utilizzando invece il buffer quando possibile (Le pagine vengono mantenute nel buffer finché non è pieno e non possono essere inserite altre pagine). Le politiche di gestione del buffer sono simili a quelle che il sistema operativo usa per gestire la memoria con le condizioni:

- **principio della località dei dati:** i dati a cui si accede correntemente hanno maggiore probabilità di esser usati anche in futuro
- **legge empirica:** l'80% delle transazioni accede solo al 20% dei dati

L'interfaccia offerta dal gestore del buffer (non potendosi limitare alle richieste di W e R) ha però la necessità di informazioni sul prevedibile riutilizzo delle pagine e sulla eventuale necessità di effettuare immediatamente gli aggiornamenti (in modo da non perdere nulla in caso di guasto), risultano quindi necessari:

- un direttorio che indica per ciascuna pagina i corrispondenti file del disco e numero di blocco (per la descrizione del contenuto corrente del buffer)
- per ogni pagina alcune variabili di stato (contatore (C) per quanti programmi la utilizzano, bit di stato per indicare eventuali modifiche (D) poi da riportare in mem sec ecc...)

Per l'interazione con le transazioni il gestore fornisce delle operazioni fondamentali (primitive):

- **fix:** richiesta (dell'accesso a) di una pagina (richiede una lettura in memoria secondaria solo se la pagina non è nel buffer). Funzionamento:
  1. controlla se una pagina P sta nel buffer
  2. Se c'è incrementa il C relativo e restituisce alla transazione l'indirizzo della pagina (per il principio di località dei dati avviene spesso)
  3. Se non c'è viene scelto un elemento F del buffer (cercando tra le pagine libere ovvero con  $C = 0$ ) secondo una certa politica di rimpiazzamento: la più usata è LRU (least recently used). Una volta trovata la pagina libera se ne incrementa C e si restituisce F come risultato della fix (se però il bit D di F è true il contenuto di F deve essere prima riportato in memoria secondaria e devono essere aggiornati gli indirizzi prima di effettuare la lettura).

4. nel caso in cui non esistano pagine libere la transazione viene messa in attesa o abortita

- setDirty: comunica che la pagina è stata modificata
- unifix: indica che la transazione ha concluso l'utilizzo della pagina
- force: trasferisce in modo sincrono una pagina in memoria secondaria (su richiesta del gestore dell'affidabilità, non del gestore degli accessi)
- flush: trasferimento asincrono nella memoria di massa dei risultati di transazioni in commit

Altre strategie di scelta della pagina del buffer da eliminare e rimpiazzare nella fix:

- steal: la vittima può essere scelta anche tra le pagine con  $C > 0$
- no-steal: la vittima può essere scelta solo tra le pagine con  $C = 0$
- force: tutte le pagine modificate da una transazione che fa commit vengono immediatamente riportate nella memoria secondaria.
- no-force: le pagine modificate da transazioni in commit vengono riportate in memoria secondaria solo con una flush quando il gestore del buffer lo ritiene

La più usata è la steal/no-force.

### Gestore del buffer e file system

Il file system conosce la directory della memoria secondaria e il uso stato di uso: tiene conto dei blocchi liberi e di quelli allocati a file. Il DBMS usa il file system per:

- creare ed eliminare un file
- aprire e chiudere una file
- leggere un blocco o una sequenza di blocchi in una pagina del buffer
- le operazioni duali in scrittura

## 5.2 Gestione delle transazioni

### 5.2.1 Transazioni

**Definizione 105 Transazione:** identifica una unità elementare di lavoro svolta da un'applicazione, cui si vogliono associare particolari caratteristiche di correttezza, robustezza e isolamento.

altra def by slide: è un programma (inteso come successione di comandi/operazioni) in esecuzione che forma un'unità logica di elaborazione sulla base di dati.  $\square$

Una transazione comprende una o più operazioni di accesso alla base di dati (inserimenti, cancellazioni, modifiche o interrogazioni) quindi una sequenza di read e write.

**Definizione 106 Sistema transazionale:** Un sistema che mette a disposizione un meccanismo per la definizione e l'esecuzione di transazioni è detto sistema transazionale. Un sistema transazionale è in grado di definire ed eseguire transazioni per conto di applicazioni concorrenti.  $\square$

### Specifiche delle transazioni

**Definizione 107 Transazione (sintatticamente):** parte di programma caratterizzata da un inizio (begintransaction, start transaction in SQL), una fine (end-transaction, non esplicitata in SQL) e al cui interno deve essere eseguito una e una sola volta uno dei seguenti comandi:

- commit work: per terminare correttamente
- rollback work per abortire la transazione

$\square$

```
start transaction;
update ContoCorrente
    set Saldo = Saldo + 10 where NumConto =
        12202;
update ContoCorrente
    set Saldo = Saldo - 10 where NumConto =
        42177;
select Saldo as A
    from ContoCorrente
    where NumConto = 42177;
if (A>=0)  then commit work
else rollback work;
```

Figura 5.2: esempio di transazione con varie decisioni

### Proprietà delle transazioni

Tutto il codice che viene eseguito all'interno di una transazione gode delle proprietà "ACID" delle transazioni:

- Atomicity
- Consistency

- Isolation
- Durability

**Definizione 108 Atomicità:** una transazione non può lasciare la base di dati in uno stato intermedio:

- un guasto o un errore prima del commit devono causare l'annullamento delle operazioni svolte
- un guasto o un errore dopo il commit non deve avere conseguenze: se necessario vanno ripetute le operazioni già fatte

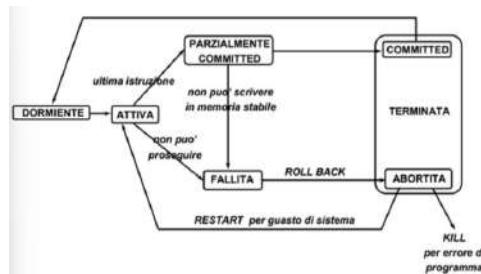
L'esito normale di una transazione è il commit (più frequente, circa il 99% dei casi); l'abort (o rollback) può essere richiesto dall'applicazione (=suicidio) oppure dal sistema (=omicidio; può essere causato da violazione dei vincoli e/o concorrenza) □

**Definizione 109 Consistenza:** La transazione rispetta i vincoli di integrità: il database è sempre in uno stato corretto: se la transazione va bene viene portato in un nuovo stato corretto, se va male annulla tutto e rimane nello stato corretto iniziale (eventuali stati non corretti si verificano solo durante le transazioni, ma al termine di esse si ha sempre uno stato corretto) □

**Definizione 110 Isolamento:** La transazione non risente degli effetti delle altre transazioni concorrenti: l'esecuzione concorrente di una collezione di transazioni deve produrre uno dei risultati che si potrebbero ottenere con una esecuzione sequenziale. Conseguenza: una transazione non espone i suoi stati intermedi □

**Definizione 111 Durabilità:** Gli effetti di una transazione andata in commit non vanno perduti ("durano per sempre"). Conseguenza: I guasti non hanno effetto sullo stato del database: uno stato consistente sarà sempre recuperato □

### Stati di una transazione

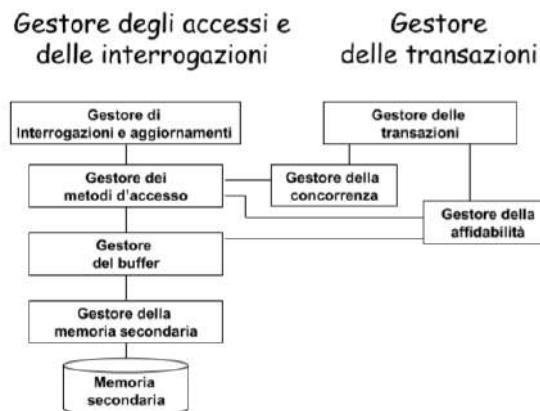


Una transazione è sempre in uno dei seguenti stati:

- active: dopo il begin-transaction è in uno stato in cui è possibile eseguire operazioni di R/W
- partially committed: lo stato raggiunto dopo che è stata eseguita l'ultima istruzione (end-transaction); il controllore dell'affidabilità deve verificare che un errore di sistema non renda impossibile registrare in modo permanente i risultati della transazione, se la verifica ha successo si passa allo stato
- committed
- failed: lo stato raggiunto dopo aver determinato che l'esecuzione non può procedere normalmente (errore SW, divisione per 0, ad es., o verifica fallita del controllore dell'affidabilità)
- aborted: la transazione ha subito un rollback e la base di dati e' stata ripristinata allo stato precedente l'inizio della transazione

## Transazioni e moduli di DBMS

- Atomicità e durabilità: gestore dell'affidabilità
- Isolamento: gestore della concorrenza
- Consistenza: gestore dell'integrità a tempo di esecuzione



### 5.2.2 Controllo dell'affidabilità

#### Prefazione

#### Possibili stati della base di dati:

- corretto = ultima versione completa e aggiornata

- valido = parte di uno stato corretto (parte dell'informazione è andata persa)
- consistente = stato valido nel quale sono rispettati i vincoli di consistenza

### **Tipi di errore o guasto:**

- logico (malfunzionamento del sw): il programma che esegue la transazione non può eseguire a causa di
  - input errato
  - eccezione
  - dato inesistente
  - divisione per zero
  - overflow
  - ...
- di sistema: l'istanza di esecuzione del programma non può proseguire per problemi di sistema (conflitto di transazioni concorrenti, deadlock), quindi bisogna rilanciare la transazione
- crash di sistema: guasto hw o sw a livello del sistema di elaborazione:
  - guasto ad una scheda
  - mancanza di alimentazione
  - problemi del sistema operativo

può quindi andare perso il contenuto del buffer
- guasto della memoria secondaria: può interessare qualche blocco o gran parte del disco
- catastrofe fisica: incendio, allagamento, terremoto...

### **Malfunzionamenti principali della base di dati:**

- malfunzionamento del disco le informazioni residenti su disco vengono perse (rottura della testina, errori durante il trasferimento dei dati)
- malfunzionamenti di alimentazione: le informazioni memorizzate in memoria centrale e nei registri vengono perse
- errori nel software: si possono generare risultati scorretti e il sistema potrebbe trovarsi in uno stato inconsistente (errori logici ed errori di sistema)

**Tipi di memoria:** Ai fini del ripristino le memorie vengono classificate come:

- Memoria volatile: le informazioni contenute vengono perse in caso di cadute di sistema (ad es. memoria principale)
- Memoria non volatile: le informazioni contenute sopravvivono a cadute di sistema, possono però essere perse a causa di altri malfunzionamenti (ad es. disco e nastri magnetici)
- Memoria stabile (non esiste): le informazioni contenute non possono essere perse

### Persistenza delle memorie:

- Memoria centrale: non è persistente, l'informazione viene distrutta da qualunque guasto del sistema
- Memoria di massa: è persistente, sopravvive ai guasti di sistema, ma può danneggiarsi l'unità di memorizzazione
- Memoria stabile: memoria che non può danneggiarsi (è una astrazione):
  - perseguita attraverso la ridondanza (dischi replicati, nastri, ...)
  - con probabilità di fallimento indipendenti

### intro

Il gestore dell'affidabilità gestisce l'esecuzione dei comandi transazionali (start transaction (B), commit work (C) e rollback work (A)) e le operazioni di ripristino (recovery) dopo i guasti (warm restart e cold restart) garantendo l'atomicità e la persistenza delle transazioni. Svolge il proprio compito attraverso l'utilizzo del *log*.

**Definizione 112 log:** è un archivio persistente su cui il gestore dell'affidabilità registra (nell'ordine temporale della loro esecuzione) le varie azioni svolte dal DBMS. □

Ogni azione sulla base di dati viene protetta tramite una azione sul log in modo che sia possibile disfare (undo) le azioni a seguito di malfunzionamenti o guasti precedenti il commit oppure rifare (redo) queste azioni in caso la buona riuscita sia incerta e le transazioni abbiano effettuato un commit.

## Architettura del controllore dell'affidabilità

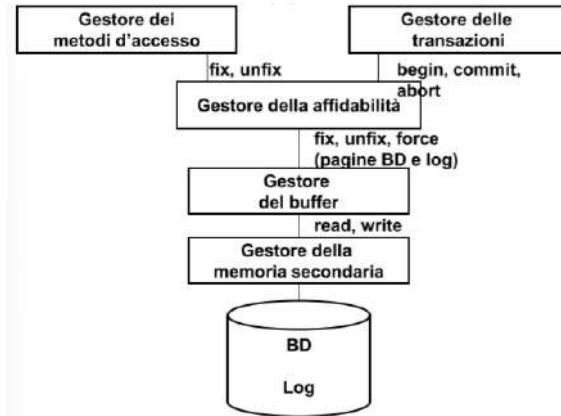


Figura 5.3: Architettura del controllore dell'affidabilità

Il controllore dell'affidabilità è responsabile di realizzare i comandi transazionali e di realizzare le operazioni di ripristino dopo i malfunzionamenti, dette rispettivamente *ripresa a caldo* e *ripresa a freddo*. Il controllore di affidabilità riceve richieste di accessi a pagine in lettura e scrittura, che passa al buffer manager, e genera altre richieste di letture e scritture di pagine necessarie a garantire la robustezza e la resistenza ai guasti. Infine, il controllore dell'affidabilità predisponde i dati necessari per eseguire i meccanismi di ripristino dai guasti, in particolare realizzando azioni di *checkpoint* e di *dump*.

### Organizzazione del log

Il log è un file sequenziale, gestito dal controllore dell'affidabilità, scritto in memoria stabile e contenente informazione ridondante che permette di ricostruire il contenuto della base di dati a seguito di guasti.

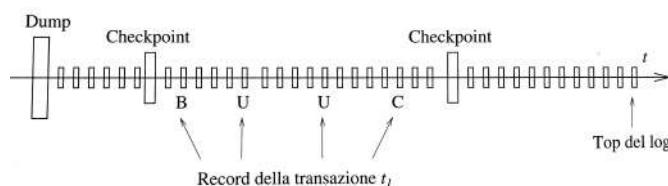
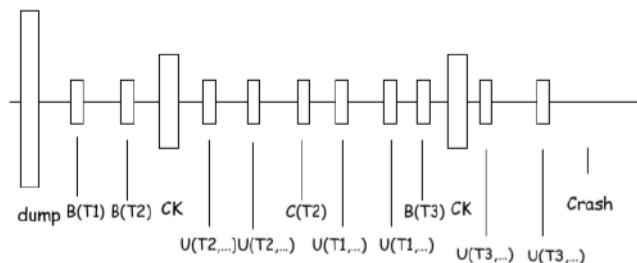


Figura 5.4: descrizione di un log

Per via della sua struttura (che segue l'ordine temporale) il log ha un blocco corrente (detto top: è l'ultimo a essere stato allocato) su cui vengono scritti i record, quando il blocco termina i record vengono scritti nel blocco successivo che diventa il corrente.

I record del log sono di due tipi:

- Record di sistema: indicano l'effettuazione di operazioni specifiche del controllore dell'affidabilità chiamate dump (abbastanza rare) e di checkpoint (più frequenti)
- Record di transazione: registrano le attività effettuate dalle transazioni nell'ordine in cui sono svolte. Pertanto, ogni transazione inserisce nel log un record di begin (corrispondente al comando start transaction), vari record relativi alle azioni effettuate (corrispondenti ai comandi insert, delete, update) e un record di commit (corrispondente al comando commit) oppure di abort (corrispondente al comando rollback).



**Struttura dei record nel log** Record per la descrizione delle attività di una transazione:

- I record di begin, commit e abort, che contengono, oltre all'indicazione del tipo di record, anche l'identificativo T della transazione.
- I record di update, che contengono l'identificativo T della transazione, l'identificativo O dell'oggetto su cui avviene l'update, e poi due valori BS e AS che descrivono rispettivamente il valore dell'oggetto O precedentemente alla modifica, detto before state, e successivamente alla modifica, detto after state.
- I record di insert e di delete sono analoghi a quelli di update, da cui si differenziano per l'assenza nei primi del before state e nei secondi dell'after state.

Nel seguito, useremo i simboli  $B(T)$ ,  $A(T)$ ,  $C(T)$  per denotare i record di begin, abort e commit e  $U(T, O, BS, AS)$ ,  $I(T, O, AS)$  e  $D(T, O, BS)$  per denotare i record di update, insert e delete. Questi record consentono di disfare e rifare le rispettive azioni sulla base di dati, attraverso operazioni specifiche, di competenza del gestore dell'affidabilità, chiamate Undo e Redo e realizzate nel modo seguente:

- Undo: per disfare una azione su un oggetto O è sufficiente ricopiare nell'oggetto O il valore BS; l'insert viene disfatto (undo) cancellando l'oggetto O.
- Redo: per rifare una azione su un oggetto O è sufficiente ricopiare nell'oggetto O il valore AS; il delete viene rifatto cancellando l'oggetto O.

Dato che le primitive di Undo e Redo sono definite tramite una azione di copia-tura, vale per esse una proprietà essenziale, detta idempotenza, per la quale l'ef-fettuazione di un numero arbitrario di undo o redo della stessa azione equivale allo svolgimento di tale azione una sola volta:

$$Undo(Undo(A)) = Undo(A) \quad Redo(Redo(A)) = Redo(A)$$

Questa proprietà è utile perché, come vedremo, si potrebbero avere errori durante le operazioni di ripristino, che portano alla ripetizione delle operazioni di Undo e Redo.

**Checkpoint e dump** In caso di guasto per effettuare un ripristino andrebbe riletto tutto il log=operazione troppo lunga. Per evitare ciò abbiamo:

**Definizione 113 Checkpoint:** operazione svolta periodicamente (dal gest. dell'affidabilità) in coordinamento con in buffer manager con lo scopo di:

- registrare quali transazioni sono attive in un certo istante
- (e dualmente) confermare che le altre o non sono iniziate o sono finite: per tutte le transazioni che hanno effettuato il commit si possono infatti trasferire dati in memoria di massa

Descrizione dell'operazione checkpoint:

- Si sospende l'accettazione delle operazioni di commit o abort da parte delle transazioni
- Si forza (force) la scrittura in memoria di massa delle pagine del buffer modificate da transazioni che hanno già fatto commit
- Si forza (force) la scrittura nel log di un record contenente gli identificatori delle transazioni attive
- Si riprendono ad accettare tutte le operazioni da parte delle transazioni

Con questo funzionamento si garantisce la persistenza delle transazioni che hanno eseguito il commit.  $\square$

**Definizione 114 Dump:** copia completa e consistente della base di dati che viene normalmente effettuata in mutua esclusione con tutte le altre transazioni quando il sistema non è operativo. La copia viene memorizzata in memoria stabile (prende il nome di *backup*). Dopo la conclusione dell'operazione di dump viene scritto nel log un record di dump, che segnala appunto la presenza di una copia fatta in un determinato istante; dopodiché il sistema può tornare al suo funzionamento normale. (nei sistemi moderni è possibile eseguire il backup con il sistema attivo: hot backup)  $\square$

Nel seguito, useremo i simboli DUMP per denotare il record di dump e  $CK(T_1, T_2, \dots, T_h)$  per denotare un record di checkpoint, dove  $T_1, T_2, \dots, T_h$  denotano gli identificatori delle transazioni attive all'istante del checkpoint.

### Esecuzione delle transazioni e scrittura del log

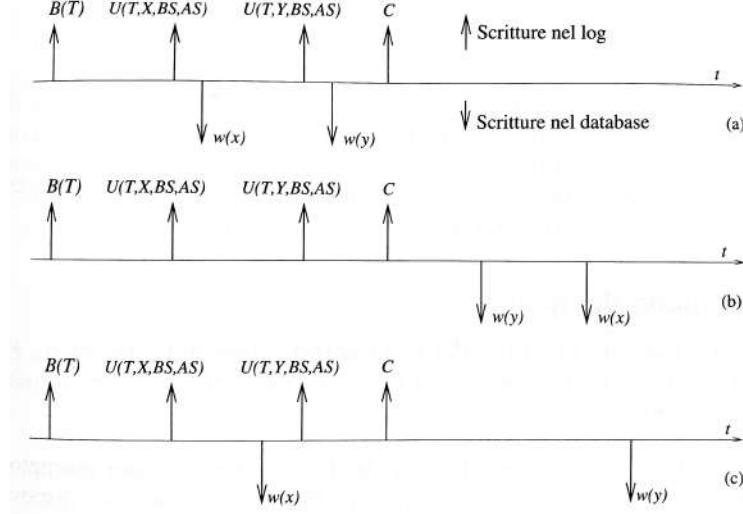
Durante il funzionamento delle transazioni il controllore dell'affidabilità deve garantire che siano seguite due regole (che definiscono i requisiti minimi che consentono di ripristinare la correttezza della base di dati a fronte di guasti):

- WAL (write ahead log): si scrive nel log (e quindi in memoria stabile) la parte BS dei record del log prima di effettuare la corrispondente operazione sul database (consente di disfare le azioni (già memorizzate in memoria di massa) di transazioni senza commit avendo in memoria stabile un valore corretto)
- Commit-precedenza: si scrive la parte AS dei record di log prima del commit (consente di rifare le azioni di una transazione che ha effettuato il commit ma le cui pagine modificate non sono ancora state trascritte dal buffer manager in memoria di massa)

Anche se le regole fanno riferimento a before state e after state separatamente, nella pratica entrambe le componenti del record di log vengono scritte assieme. Le regole semplificate risultano quindi:

- i record di log sono scritti prima dei corrispondenti record della base di dati (regola WAL)
- i record di log sono scritti prima dell'esecuzione dell'operazione di commit (regola di commit-precedenza).

**Eisito di una transazione** L'esito di una transazione è determinato irrevocabilmente quando viene scritto il *record di commit* nel log. Un guasto prima di tale istante porta ad un undo di tutte le azioni, per ricostruire lo stato originario della base di dati. Un guasto successivo non deve avere conseguenze: lo stato finale della base di dati deve essere ricostruito, con redo se necessario.



**Scrittura congiunta di log e base di dati** Modalità di scrittura:

**Definizione 115 Modalità immediata (caso a):** .

- Il DB contiene valori AS provenienti da transazioni uncommitted
- Richiede Undo delle operazioni di transazioni uncommitted al momento del guasto
- Non richiede Redo

□

**Definizione 116 Modalità differita (caso b):** .

- Il DB non contiene valori AS provenienti da transazioni uncommitted
- In caso di abort, non occorre fare niente
- Rende superflua la procedura di Undo, non ci sono scritture prima del commit.
- Richiede redo

□

**Definizione 117 Modalità mista (caso c):** .

- La scrittura può avvenire in modalità sia immediata che differita
- Richiede sia Undo che Redo

□

Tutti e tre i protocolli (noi li chiamiamo modalità) rispettano entrambe le regole wal e C-P e scrivono il record di commit in modo sincrono: si differenziano solo per il momento in cui scrivono le pagine della base di dati.

**Note (divise dalle linee)** La modalità differita di scrittura pur permettendo una procedura di recovery più semplice ed efficiente, non viene molto utilizzata in pratica. Questo perché questa modalità è più efficiente nel recovery, ma è complessivamente meno efficiente di una in cui il gestore del buffer può decidere liberamente quando scrivere in memoria secondaria (è preferibile una gestione ordinaria più efficiente rispetto ad una gestione più semplice dei guasti, poiché si assume che i guasti siano abbastanza rari.)

Nella pratica:

- La scrittura nella base di dati può avvenire in qualunque momento, anche prima del commit
- La scrittura sul log è effettuata prima della scrittura nella base di dati
- Il commit si considera effettuato quando il corrispondente record di log è scritto (prima di questa scrittura il guasto causa l'undo di tutte le operazioni, dopo il redo)

**Definizione 118 rollback di una transazione:** Quando una transazione deve essere cancellata, per un errore logico dell'operazione contenuta oppure per un'esigenza di sistema, tutte le operazioni tra il begin della transazione e l'abort devono essere disfatte, alla fine un record di abort è scritto nel log. □

Conseguenze delle politiche steal/nosteal e force/no-force: (la politica nosteal e force è la più semplice da implementare e lascia più libertà al gestore del buffer per cercare maggiore efficienza):

- Se la politica della ricerca della vittima da rimpiazzare è no-steal non si deve fare undo delle transazioni in rollback, però nel buffer ci sono tante pagine attive
- Con force non dobbiamo fare redo delle azioni di transazioni in commit di cui non sono stati trasferiti i risultati in memoria di massa, però si devono fare molti trasferimenti

## Gestione dei guasti

### Guasti

- *soft o di sistema* (sw) (errori di programma, crash di sistema, cadute di tensione):
  - si perde la memoria centrale e quindi anche il buffer
  - non si perde la memoria secondaria, cioè la base di dati e il log
- *hard o di dispositivo* (hw) (dei di dispositivi di memoria secondaria):
  - si perde anche la memoria secondaria

- non si perde la memoria stabile e quindi il log

N.B. la perdita del log è considerato un evento catastrofico e quindi non è definita alcuna strategia di recupero.

**Modello fail-stop** Il modello ideale in cui ci poniamo è detto di fail-stop: quando il sistema individua un guasto, sia di sistema sia di dispositivo, esso forza immediatamente un arresto completo delle transazioni e il successivo ripristino del corretto funzionamento del sistema operativo (boot<sup>1</sup>). Quindi, viene attivata una procedura di ripresa, denominata ripresa a caldo (warm restart) nel caso di guasto di sistema e ripresa a freddo (cold restart) nel caso di guasto di dispositivo. Al termine delle procedure di ripresa, il sistema diventa nuovamente utilizzabile dalle transazioni; il buffer è completamente vuoto e può riprendere a caricare pagine della base di dati o del log.

**Processo di restart** Obiettivo: classificare le transazioni in:

- completate (tutti i dati in memoria)
- attive ma con commit (vanno rificate, redo)
- attive senza commit (vanno annullate, undo)

**Ripresa a caldo** è articolata in 4 fasi:

- trovare l'ultimo checkpoint (ripercorrendo il log a ritroso)
- costruire gli insiemi UNDO (transazioni attive ma non committed prima del guasto, da disfare) e REDO (transazioni committed tra il CK e il guasto, da rifare): l'insieme di UNDO si inizializza con le transazioni attive al checkpoint mentre REDO come insieme vuoto. UNDO si aggiorna con l'aggiunta delle transazioni che presentano il record di begin, quando invece si trova un record di commit si aggiunge tale transazione all'insieme di REDO. (alla fine della fase insiemi nello stato corretto)
- ripercorrere il log all'indietro, fino alla più vecchia azione delle transazioni in UNDO e REDO, disfacendo tutte le azioni delle transazioni in UNDO
- ripercorrere il log in avanti, rifacendo tutte le azioni delle transazioni in REDO

---

<sup>1</sup>da wikipedia: fase iniziale dell'avviamento di un computer, precedente al caricamento del sistema operativo.

Questo meccanismo garantisce l'atomicità e la persistenza delle transazioni.

Esempio di applicazione del protocollo:

Dump, B(T1), B(T2), I(T1,O1,A1), D(T2,O2,B2), B(T3), B(T4)

U(T3,O3,B3,A3), C(T2), CK(T1, T3,T4), U(T1,O4,B4,A4), A(T3), B(T5),

D(T4,O5,B5), C(T1), C(T4), I(T5,O6,A6), GUASTO

Passi da eseguire:

1. individuare le transazioni attive al CK: si accede al record di CK:  
 $UNDO = \{T1, T3, T4\}$ ,  $REDO = \{\}$
2. compilare l'elenco delle transazioni da disfare e rifare: si percorre il log in avanti fino a completare gli insiemi  $UNDO = \{T3, T5\}$ ,  $REDO = \{T1, T4\}$
3. ripristinare il sistema: percorrere il log indietro e disfare le transazioni in UNDO:
  - (a) I(T5,O6,A6) → delete O6
  - (b) U(T3,O3,B3,A3) → O3:=B3

svolgere le azioni in REDO:

- I(T1,O1,A1) → insert O1 := A1
- U(T1,O4,B4,A4) → O4 := A4
- D(T4,O5,B5) → delete O5

Nota: Le operazioni derivanti dal rollback di una transazione (transazioni abortite) possono essere inserite nell'insieme UNDO e fatte al momento del recovery oppure essere eseguite al momento dell'abort ed essere inserite nell'insieme di REDO al momento del recovery da un guasto.

**Ripresa a freddo** fasi:

1. si accede al dump e si ricopia selettivamente la parte deteriorata della base di dati (=si ripristina). Si accede anche al più recente record di dump nel log.
2. si ripercorre il log in avanti, applicando relativamente alla parte deteriorata della base di dati sia le azioni sulla base di dati (update, insert, ...) sia le azioni di commit e abort. (si ottiene quindi la stessa situazione precedente al guasto)
3. si esegue una ripresa a caldo

Esempio (stessi record del precedente):

Descrizione dei passi da compiere per effettuare la ripresa a freddo dopo un guasto che interessa O1, O2, O3.

1. Insert O1=A1
2. Delete(O2)
3. O3=A3
4. Commit(T2)
5. Abort(T3)
6. Commit(T1)
7. Commit(T4)
8. Ripresa a caldo

### 5.2.3 Controllo della concorrenza

Un DBMS è spesso sottoposto a un grande numero di tps (transaction per second), per questo motivo risulta fondamentale la concorrenza: per supportare un grande carico applicativo (numero di tps) le transazioni non possono essere seriali. L'esecuzione concorrente può però causare anomalie e va quindi governata.

#### Architettura

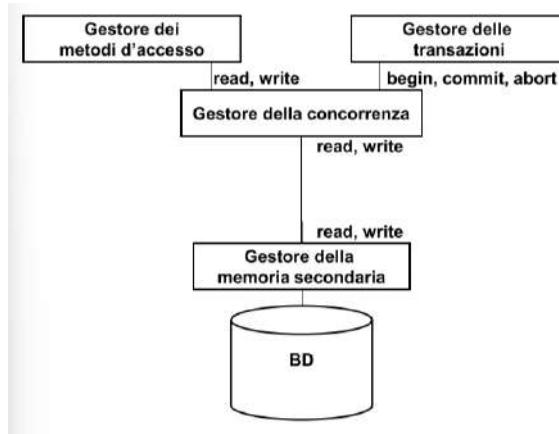


Figura 5.5: Architettura del controllore della concorrenza

Le funzionalità del controllo di concorrenza interagiscono con quelle relative alla gestione dei metodi di accesso. (in modo semplice:) Il controllore della concorrenza riceve le richieste di accesso ai dati (richieste al buffer) e decide se autorizzarle o meno, eventualmente riordinandole (per questo motivo è anche chiamato *scheduler*)

## Anomalie delle transazioni

L'esecuzione concorrente può però causare anomalie, è quindi necessario controllare la concorrenza.

**Perdita di aggiornamento (lost of update)** considero due transazioni identiche t1 e t2 definite come

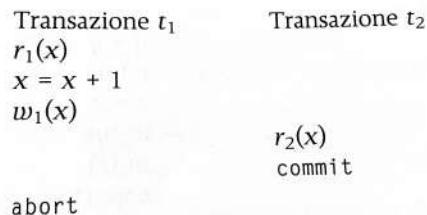
$$t_n : r_n(x), x = x + 1, w_n(x)$$

In caso di esecuzione seriale, data  $x=2$ , alla fine si ha  $x=4$ . In caso di esecuzione concorrente, come in figura:



alla fine si ha  $x=3$  perché gli effetti di t2 vengono persi (entrambe le t hanno avuto come valore iniziale di x il 2).

**Lettura sporca (dirty read)** in questo caso si considera che t1 vada in abort dopo aver scritto e t2 che legge il dato scritto da t1:



alla fine il valore di x è 2 ma t2 ha letto 3 il quale valore, essendo andata t1 in abort, è come se non fosse mai esistito, se non fosse che t2 ha potuto leggere quel dato (e magari comunicarlo all'esterno).

**Definizione 119 lettura sporca:** quando viene letto un dato che rappresenta uno stato intermedio nell'evoluzione di una transazione.  $\square$

Transazione $t_1$ $r_1(x)$ $r_1(x)$ <b>commit</b>	Transazione $t_2$ $r_2(x)$ $x = x + 1$ $w_2(x)$ <b>commit</b>
--	---

**Letture inconsistenti**  $x$  assume valore 2 dopo la prima read e 3 dopo la seconda: questo non è opportuno in quanto è invece opportuno che una transazione che accede due volte alla base di dati trovi esattamente lo stesso valore per ciascun dato letto e non riseta dell'effetto di altre transazioni.

**Aggiornamento fantasma (ghost update)** si suppone che gli oggetti  $x$ ,  $y$  e  $z$  rispettino il vincolo di integrità  $x + y + z = 1000$

Transazione $t_1$ $r_1(x)$ $r_1(y)$ $r_1(z)$ $s = x + y + z$ <b>commit</b>	Transazione $t_2$ $r_2(y)$ $y = y - 100$ $r_2(z)$ $z = z + 100$ $w_2(y)$ $w_2(z)$ <b>commit</b>
---	--

$t_2$  non altera la somma dei valori e quindi rispetta il vincolo, però  $s$  (ovvero la somma) contiene 1100. Questo è dovuto al fatto che  $t_1$  osserva solo in parte gli effetti di  $t_2$ .

**Inserimento fantasma (phantom)** .

$t_1$	$t_2$
<code>bot</code> "legge gli stipendi degli impiegati del dip A e calcola la media"	<code>bot</code> "inserisce un impiegato in A" <b>commit</b>
<code>bot</code> "legge gli stipendi degli impiegati del dip A e calcola la media" <b>commit</b>	

## Teoria del controllo di concorrenza

**Definizione 120 Transazione (def. formale):** sequenza di azioni di lettura o scrittura (omettiamo ora da questo modello ogni riferimento alle operazioni di manipolazione dei dati). es.

$$t_1 : r_1(x)r_1(y)w_1(x)w_1(y)$$

□

Dato che le transazioni (rappresentabili come una sequenza di R/W) vengono eseguite in modo concorrente, le operazioni di R/W vengono richieste da transazioni differenti in interleaving.

**Definizione 121 schedule:** Uno schedule è una sequenza di R/W relative all'insieme delle transazioni concorrenti in un certo istante. Formalmente, uno schedule S1 è unasequenza del tipo:

$$S_1 : r_1(x)r_2(z)w_1(x)w_2(z)$$

dove r1(x) rappresenta la lettura dell'oggetto x da parte della transazione t1 e w2(z) rappresenta la scrittura dell'oggetto z da parte della transazione t2. Le operazioni compaiono nello schedule nell'ordine temporale di esecuzione sulla base di dati. □

Il **controllo della concorrenza** è eseguito dallo scheduler, che tiene traccia di tutte le operazioni eseguite sulla base di dati dalle transazioni e decide se accettare o rifiutare le operazioni che vengono via via richieste. Per il momento, assumiamo che l'esito (commit/abort) delle transazioni sia noto a priori (modello non reale): in questo modo è possibile ignorare le transazioni che producono un abort togliendo quindi dallo schedule tutte le loro azioni e concentrarsi solo sulle transazioni che producono un commit (in questo caso lo schedule si dice una commit-proiezione dell'esecuzione delle operazioni R/W)(Si noti che tale assunzione non consente di trattare alcune anomalie (lettura sporca)).

Recap:

- obiettivo: evitare le anomalie
- soluzione: scheduler (=sistema che accetta o rifiuta, anche tramite riordino, le operazioni richieste dalle transazioni)

**Definizione 122 Schedule seriale:** Uno schedule S si dice seriale se, per ogni transazione t, tutte le azioni di t compaiono in sequenza, senza essere inframmezzate da azioni di altre transazioni. es:

$$S_2 : r_1(x)w_1(x)r_2(z)w_2(z)$$

□

**Definizione 123 Schedule serializzabile (view-serializzabile):** L'esecuzione di uno schedule (commit-proiezione)  $S_i$  è corretta quando produce lo stesso risultato di un qualunque schedule seriale  $S_j$  definito dalle stesse transazioni di  $S_i$ . In questo caso, lo schedule  $S_i$  è detto serializzabile. L'insieme degli schedule view-serializzabili è indicato con VSR.  $\square$

Questa definizione ci porta ad avere la necessità di definire l'equivalenza tra schedule.

**Definizione 124 view-equivalenza:** Definiamo la relazione *legge-da* tra le operazioni  $r_i(x)$  e  $w_j(x)$  ( $r_i(x)$  è una lettura che *legge da*  $w_j(x)$ ) presenti in uno schedule  $S$  se  $w_j(x)$  precede  $r_i(x)$  in  $S$  e non c'è nessun  $w_k(x)$  con  $k \neq j$  tra di loro. Definiamo quindi una  $w_i(x)$  in  $S$  detta *scrittura finale* su  $x$  se è l'ultima scrittura sull'oggetto  $x$  in  $S$ .

**Def.** due schedule sono view-equivalenti ( $S_i \approx_V S_j$ ) se hanno la stessa relazione *legge-da* e le stesse scritture finali su ogni oggetto.  $\square$

Se però determinare la view equivalenza di due schedule è un problema a complessità polinomiale, determinare la view serializzabilità è NP completo (gli schedule seriali da confrontare a quello dato sono in numero esponenziale). Abbiamo quindi bisogno di un concetto di equivalenza più stretto (che non copre tutti i casi di equivalenza tra schedule) ma che sia utilizzabile nella pratica.

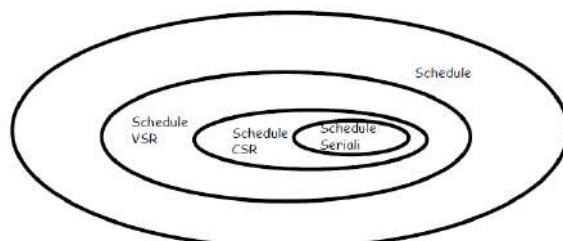
**Definizione 125 Conflict-equivalenza: Definizione preliminare:** Un'operazione  $a_i$  è in conflitto con un'altra  $a_j$  ( $i \neq j$ ), se operano sullo stesso oggetto e almeno una di esse è una scrittura. Due casi:

- conflitto read-write (rw o wr)
- conflitto write-write (ww).

**Def:** due schedule sono conflict equivalenti ( $S_i \approx_C S_j$ ) se includono le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi.  $\square$

**Definizione 126 Conflict-serializzabilità:** Uno schedule è conflict-serializzabile se è conflict-equivalente ad un qualche schedule seriale. L'insieme degli schedule conflict-serializzabili è indicato con CSR.  $\square$

Si può dimostrare che la classe degli schedule CSR è strettamente contenuta nella classe degli schedule VSR (CSR implica VSR)



**Verifica di conflict-serializzabilità** Si fa per mezzo del *grafo dei conflitti*:

- un nodo per ogni transazione  $t_i$
- un arco (orientato) da  $t_i$  a  $t_j$  se c'è almeno un conflitto fra un'operazione  $a_i$  e un'operazione  $a_j$  tale che  $a_i$  precede  $a_j$

**Teorema:** uno schedule è in CSR se e solo se il grafo è aciclico.

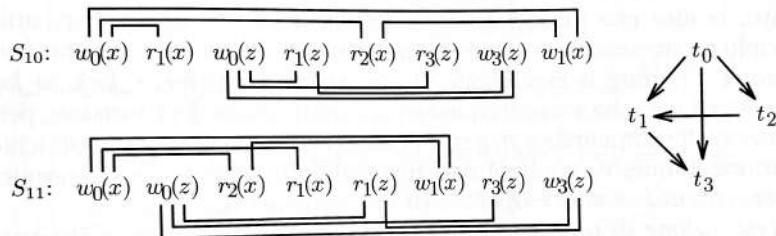


Figura 5.6:  $S_{10}$  è conflict-serializzabile (ne sono posti in evidenza i conflitti) e  $S_{11}$  è il corrispondente schedule equivalente seriale

Questa strategia di verifica risulta però irrealizzabile dal momento in cui il sistema diventa di grandi dimensioni.

- In un sistema con 100 tps e che mediamente accedono a 10 pagine e durano 5 secondi, vanno gestiti in ogni istante grafici con 500 nodi, tenendo traccia dei 5000 accessi delle 500 transizioni attive.
- Il grafo dei conflitti continua a modificarsi dinamicamente, rendendo difficoltose le decisioni dello scheduler.
- Quindi non è praticabile mantenere il grafo, aggiornarlo e verificarne l'aciclicità on-line ad ogni richiesta di operazione.
- N.B. la tecnica risulta del tutto impraticabile nel caso di basi di dati distribuite (il grafo deve essere ricostruito a partire da archi riconosciuti dai diversi server del sistema).

Figura 5.7: esempio

Quindi nella pratica si utilizzano tecniche di scheduling che garantiscono la c-s a priori senza dover costruire il grafo.

**Lock** principio :

1. Tutte le letture sono precedute da un lock e seguite da unlock, in questo caso si parla di lock *condiviso* (su un dato possono essere attivi più lock di questo tipo)
2. Tutte le scritture sono precedute da un lock e seguite da unlock, in questo caso si parla di lock *esclusivo* (non può coesistere con altri lock)

Il *lock manager* (che è parte dello scheduler) riceve queste richieste dalle transazioni e le accoglie o rifiuta.

**Definizione 127 t. ben formata rispetto al locking:** quando una transazione segue i due principi si dice tale.  $\square$

**Definizione 128 lock upgrade:** se una t. deve contemporaneamente leggere e scrivere essa può richiedere solo un lock di tipo esclusivo, oppure, passare al momento opportuno da un lock condiviso a uno esclusivo incrementando il livello di lock (lock upgrade)  $\square$

Il gestore della concorrenza concede i lock in base ai lock precedentemente concessi ad altre transazioni.

**Definizione 129 Risorsa acquisita o rilasciata:** quando una richiesta di lock è concessa si dice che la corrispondente risorsa viene acquisita dalla transazione richiedente, all'atto dell'unlock la risorsa viene rilasciata.  $\square$

**Definizione 130 transazione in stato di attesa:** quando una richiesta di lock non viene concessa la t. richiedente viene messa in stato di attesa. L'attesa termina quando la risorsa viene sbloccata.  $\square$

I lock già concessi vengono memorizzati in tabelle di lock, gestite dal lock manager.

La politica che viene seguita dal lock manager per concedere i lock è rappresentata nella tabella dei conflitti in cui le righe identificano le richieste, le colonne lo stato della risorsa richiesta, il primo valore nella cella l'esito della richiesta e il secondo valore nella cella lo stato che verrà assunto dalla risorsa dopo l'esecuzione della primitiva.

Richiesta	Stato risorsa		
	libero	r_locked	w_locked
r_lock	OK / r_locked	OK / r_locked	No / w_locked
w_lock	OK / w_locked	No / r_locked	No / w_locked
unlock	error	OK / dipende	OK / libero

I tre "no" presenti nella tabella rappresentano i conflitti che si possono presentare, quando si richiede una lettura o una scrittura su un oggetto già bloccato in scrittura, o una scrittura su un oggetto già bloccato in lettura (da qui lock condiviso in lettura).

- Viene garantita la mutua esclusione sulla risorsa, ma non la serializzabilità.
  - E' possibile imporre un ordine alle richieste di acquisizione e rilascio di una risorsa che automaticamente garantiscono la serializzabilità?
- ```

begin (T1)
w1_lock(B);
r1(B);
B:=B-50;
w1(B);
unlock(B);
w1_lock(A);
r1(A);
A:=A+50;
w1(A);
unlock(A);
commit

```

Figura 5.8: ordine di lock e unlock

La restrizione che quindi è necessario porre all'ordinamento delle richieste di lock prende il nome di

**Definizione 131 Locking a due fasi (2PL=2 phase locking):** è un principio basato su due regole:

- "proteggere" tutte le R e W con un lock (prima fase attraversata da una t. in cui acquisisce lock per le risorse a cui deve accedere: fase crescente)
- una transazione, dopo aver rilasciato un lock, non può acquisirne altri finché tutti quelli che ha acquisito non sono stati rilasciati (seconda fase di una t. in cui rilascia: fase calante).

□

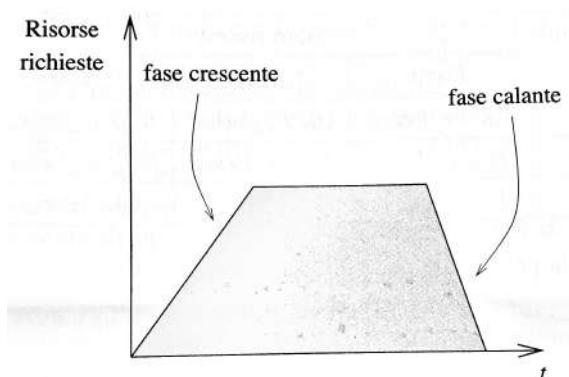


Figura 5.9: rappresentazione del 2PL. nota: la fase crescente (in cui si acquisiscono i lock per le risorse a cui si deve accedere) può contenere anche i lock upgrade

Questo principio è usato da quasi tutti i sistemi e garantisce "a priori" la conflict-serializzabilità.

**Nota 9 :** upgrading e downgrading dei lock: *L'upgrade si può fare solo nella fase di acquisizione dei lock, il downgrade nella fase di rilascio.*

**Nota 10 :** 2PL e CSR: *ogni schedule 2PL è anche CSR ma non il viceversa (2PL  $\Rightarrow$  CSR). La classe di schedule 2PL è quindi contenuta nella classe CSR.*

**Le anomaloalie** E' facile vedere che 2PL risolve le anomalie di perdita di aggiornamento, di aggiornamento fantasma e di letture inconsistenti.

| $t_1$           | $t_2$          | $x$    | $y$     | $z$     |
|-----------------|----------------|--------|---------|---------|
| $r\_lock_1(x)$  |                | free   | free    | free    |
| $r_1(x)$        |                | 1:read |         |         |
|                 | $w\_lock_2(y)$ |        | 2:write |         |
|                 | $r_2(y)$       |        |         |         |
| $r\_lock_1(y)$  |                |        | 1:wait  |         |
|                 | $y = y - 100$  |        |         |         |
|                 | $w\_lock_2(z)$ |        |         | 2:write |
|                 | $r_2(z)$       |        |         |         |
|                 | $z = z + 100$  |        |         |         |
|                 | $w_2(y)$       |        |         |         |
|                 | $w_2(z)$       |        |         |         |
|                 | commit         |        |         |         |
|                 | $unlock_2(y)$  |        | 1:read  |         |
| $r_1(y)$        |                |        |         |         |
| $r\_lock_1(z)$  |                |        |         | 1:wait  |
|                 | $unlock_2(z)$  |        |         | 1:read  |
| $r_1(z)$        |                |        |         |         |
| $s = x + y + z$ |                |        |         |         |
| commit          |                |        |         |         |
| $unlock_1(x)$   |                | free   |         |         |
| $unlock_1(y)$   |                |        | free    |         |
| $unlock_1(z)$   |                |        |         | free    |

Figura 5.10: esempio di prevenzione di un ghost update tramite l'uso del 2PL

2PL presenta però altre anomalie:

- Il fallimento di una transazione che ha scritto una risorsa deve causare il fallimento di tutte le transazioni che hanno letto il valore scritto (lettura sporche)
- Attese incrociate (o deadlock): due transazioni detengono ciascuna una risorsa e aspettano la risorsa detenuta dall'altra. In generale, la probabilità di deadlock è bassa, ma non nulla.

| Cascading rollbacks                                                                                                                                                                                                                                                                                                  | Deadlock                                                                                                                                                             |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> • begin(T1); • w1_lock(A); • r1(A); • r1_lock(B); • r1(B); • w1(A); • unlock(A); • abort       begin(T2);       w2_lock(A);       r2(A);       w2(A);       unlock(A); ...       begin(T3);       r3_lock(A);       r3(A); ...   </pre> <p>Quando T1 fallisce, il fallimento si deve trasmettere a T2 e T3</p> | <pre> • begin(T1) • w1_lock(B); • r1(B); • B:=B-50; • w1(B); begin(T2) r2_lock(A); r2(A); r2_lock(B); wait T1 • w1_lock(A); • wait T2 • r1(B); • unlock (B)   </pre> |

**Definizione 132 Locking a 2 fasi stretto (strict 2PL):** condizione aggiuntiva: I lock possono essere rilasciati solo dopo il commit. Questo elimina il rischio di letture sporche e quindi di rollback in cascata.  $\square$

**Nota 11:** dato uno schedule S2PL esiste almeno uno schedule VSR equivalente ad esso (che risulta quindi anche CSR)

**Controllo di concorrenza basato su timestamp** È una tecnica alternativa al 2PL e utilizza i:

**Definizione 133 Timestamp (TS):** identificatore (associato a ciascun evento temporale) che definisce un ordinamento totale sugli eventi di un sistema.  $\square$

Il controllo mediante TS avviene quindi in questo modo:

- Ogni transazione ha un timestamp che rappresenta l'istante di inizio della transazione (il TS è generato dall'orologio di sistema).
- Uno schedule è accettato solo se riflette l'ordinamento seriale delle transazioni indotto dai timestamp.

(Questo metodo è quindi il più semplice da realizzare ma richiede che le t. siano serializzate in base all'ordine dei rispettivi TS.)

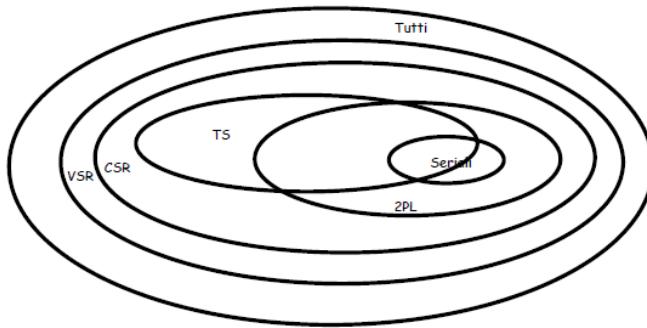
In maniera più dettagliata:

- Lo scheduler ha due contatori RTM(x) e WTM(x) per ogni oggetto
- Lo scheduler riceve richieste di letture e scritture (con indicato il timestamp della transazione):
  - read(x,ts): se ts < WTM(x) allora la richiesta è respinta e la transazione viene uccisa; altrimenti, la richiesta viene accolta e RTM(x) è posto uguale al maggiore fra RTM(x) e ts
  - write(x,ts): se ts < WTM(x) o ts < RTM(x) allora la richiesta è respinta e la transazione viene uccisa, altrimenti, la richiesta viene accolta e WTM(x) è posto uguale a ts

- alla fine risulta che il metodo Ts ha comportato l'uccisione di molte t.

**Nota 12:** ad uno schedule eseguibile con ts esiste un **unico** schedule seriale equivalente

**Nota 13 :** 2PL e TS: Gli schedule TS sono automaticamente CSR: corrispondono ad una esecuzione seriale (quella in cui le transazioni sono eseguite nell'ordine in cui sono iniziata), tuttavia 2PL e TS sono incomparabili.



**ATTENZIONE:** L'ordine seriale delle transazioni è fissato prima che le operazioni vengano richieste, tutti gli altri ordinamenti non sono accettati. Quando T1 comincia prima di T2, potrebbe essere abilitato uno schedule 2PL o CSR equivalente ad uno seriale T2 T1; col TS non è possibile, al limite T1 viene uccisa e poi fatta ripartire dopo T2.

- In 2PL le transazioni sono poste in attesa quando non è possibile acquisire un lock, in TS uccise e rilanciate (Le ripartenze sono di solito più costose delle attese: conviene il 2PL)
- 2PL può causare deadlock, TS no (mediamente si uccide una transazione ogni due conflitti, ma la probabilità di insorgenza di deadlock è molto minore della probabilità di un conflitto: conviene il 2PL)

### Risoluzione dello stallo

**Definizione 134** stallo (libro=blocco critico): corrisponde ad un ciclo nel grafo delle attese □

3 tecniche di risoluzione:

1. **Timeout:** Le transazioni rimangono in attesa di una risorsa per un tempo prefissato. Se, trascorso tale tempo, la risorsa non è ancora stata concessa, alla richiesta di lock viene data risposta negativa. In tal modo una transazione in potenziale stato di deadlock viene tolta dallo stato di attesa e di norma abortita. Tecnica molto semplice, usata dalla gran parte dei sistemi commerciali.
2. **Rilevamento dello stallo:** ricerca di cicli nel grafo delle attese.

**3. Prevenzione dello stallo:** Prevenzione: uccisione di transazioni "sospette".

(sul libro fase 2 e 3 sono invertite)

**Come scegliere il timeout:**

- Un valore troppo elevato tende a risolvere tardi i blocchi critici, dopo che le transazioni coinvolte hanno trascorso diverso tempo in attesa.
- Un valore troppo basso rischia di interpretare come blocchi anche situazioni in cui una transazione sta attendendo la disponibilità di una risorsa destinata a liberarsi, uccidendo la transazione e sprecando il lavoro già svolto.

**Come scegliere la transazione da uccidere:**

- Politiche interrompenti: un conflitto può essere risolto uccidendo la transazione che possiede la risorsa (in tal modo, essa rilascia la risorsa che può essere concessa ad un'altra transazione). Criterio aggiuntivo: uccidere le transazioni che hanno svolto meno lavoro (si spreca meno).
- Politiche non interrompenti: una transazione può essere uccisa solo nel momento in cui effettua una nuova richiesta.

**Problema aggiuntivo dal criterio aggiuntivo:**

- Una transazione, all'inizio della propria elaborazione, accede ad un oggetto richiesto da molte altre transazioni, così è sempre in conflitto con altre transazioni e, essendo all'inizio del suo lavoro, viene ripetutamente uccisa.
- Non c'è deadlock, ma starvation. Possibile soluzione: mantenere invariato il tempo di partenza delle transazioni abortite e fatte ripartire, dando in questo modo priorità alle transazioni più anziane.

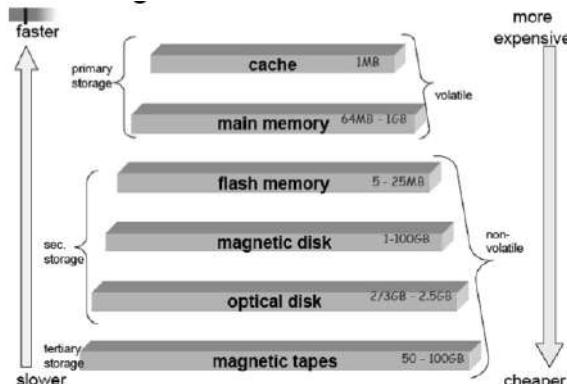
**SQL 1999** mi sa che è inutile

## 5.3 Organizzazione fisica

### 5.3.1 Intro

**Definizione 135 DBMS:** sistema per gestire collezioni di dati **grandi, persistenti e condivise** garantendo **affidabilità e privatezza**. Deve essere **efficiente** (utilizzando al meglio le risorse di spazio e tempo del sistema) ed **efficace** (rendendo produttive le attività dei suoi utilizzatori) □

**DBMS e file system** Il file system è il componente del sistema operativo che gestisce la memoria secondaria. I DBMS ne utilizzano le funzionalità per creare ed eliminare file e per leggere e scrivere singoli blocchi o sequenze di blocchi contigui. Il DBMS gestisce i file allocati come se fossero un unico grande spazio di memoria secondaria e costruisce, in tale spazio, le strutture fisiche con cui implementa le relazioni. L'organizzazione dei file, sia in termini di distribuzione dei record nei blocchi sia relativamente alla struttura all'interno dei singoli blocchi è gestita direttamente dal DBMS. Il DBMS crea file di grandi dimensioni che utilizza per memorizzare diverse relazioni (al limite, l'intero database). È possibile che un file contenga i dati di più relazioni e che le varie tuple di una relazione siano in file diversi. Spesso, ma non sempre, ogni blocco è dedicato a tuple di un'unica relazione. I programmi possono fare riferimento solo a dati in memoria principale, quindi i dati in memoria secondaria possono essere utilizzati solo se prima trasferiti in memoria principale (questo spiega i termini "principale" e "secondaria") serve un'interazione fra memoria principale e secondaria che limita il più possibile gli accessi alla secondaria.



**Prestazioni di una memoria** Dato un indirizzo di accesso, le prestazioni di memoria secondaria si misurano in termini della somma tra il tempo che la testina impiega per raggiungere la traccia di interesse, la latenza (tempo per accedere al primo byte del blocco di interesse) e il tempo di trasferimento (tempo necessario a muovere tutti i dati del blocco).

Tempi di accesso:

- accesso a memoria secondaria:
  - tempo di posizionamento della testina (10-50ms)
  - tempo di latenza (5-10ms)
  - tempo di trasferimento (1-2ms) (in media meno di 10ms)
- Il tempo di un accesso a memoria secondaria è quattro o più ordini di grandezza maggiore di quello per operazioni in memoria centrale.

**Memoria principale e secondaria** I dispositivi di memoria secondaria sono organizzati in blocchi di lunghezza (di solito) fissa (ordine di grandezza: alcuni KB). Le uniche operazioni sono la lettura e la scrittura dei dati di un blocco. Accessi a blocchi “vicini” costano meno (contiguità): tempo di posizionamento + latenza=0. La memoria principale è organizzata in pagine.

**Buffer management** Buffer:

- area di memoria centrale, gestita dal DBMS (preallocata) e condivisa fra le transazioni
- organizzato in pagine di dimensioni pari o multiple di quelle dei blocchi di memoria secondaria (1KB- 100KB)
- Se assumiamo che coincidano pagina e blocco, il caricamento di una pagina del buffer richiede una lettura in memoria secondaria, mentre salvare una pagina corrisponde ad una scrittura

Dati gestiti dal buffer manager:

- il buffer
- una directory che per ogni pagina mantiene (ad esempio):
  - il file fisico e il numero del blocco
  - due variabili di stato:
    - \* un contatore che indica quanti programmi utilizzando la pagina
    - \* un bit che indica se la pagina è "sporca", cioè se è stata modificata

**Tuple e blocchi:** I file sono logicamente organizzati in record. I record sono mappati nei blocchi di memoria secondaria. Le tuple di una relazione (record di file) stanno in blocchi contigui. A volte in un blocco ci sono tuple di relazioni diverse ma correlate (i join sono favoriti). I blocchi (componenti "fisici" di un file) e le tuple o record (componenti "logici" di una relazione) hanno dimensioni in generale diverse:

- la dimensione del blocco dipende dal file system
- la dimensione del record dipende dalle esigenze dell'applicazione, e può anche variare nell'ambito di un file

**Organizzazione delle tuple** Ci sono varie alternative, anche legate ai metodi di accesso; in genere le tuple sono sistematamente nei file, inoltre:

- se la lunghezza delle tuple è fissa, la struttura può essere semplificata
- alcuni sistemi possono spezzare le tuple su più blocchi (necessario per tuple grandi)

### Fattore di blocco

- numero di record in un blocco

- $L_R$ : dimensione di un record (per semplicità costante nel file: "record a lunghezza fissa")
- $L_B$ : dimensione di un blocco
- se  $L_B > L_R$  possiamo avere più record di un blocco:

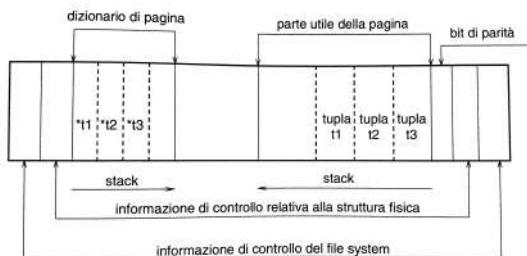
$$\lfloor L_B/L_R \rfloor$$

- lo spazio residuo può essere:

- utilizzato (record "spanned" o impaccati)
- non utilizzato ("unspanned")

| Esercizio                                                                                                                                                                                                                                                                | Soluzione                                                                                                      |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|
| <p>Calcolare il fattore di blocco e il numero di blocchi occupati da una relazione contenente <math>T = 500000</math> tuple (record) di lunghezza fissa pari a <math>R = 100</math> byte in un sistema con blocchi di dimensione pari a <math>B = 1</math> kilobyte.</p> | $D_T = T * R \text{ byte}$<br>$N_B = D_T / B$<br>$N_B = 5000000 / 1024$<br>$F_B = B / R$<br>$F_B = 1024 / 100$ |

### Organizzazione delle tuple nelle pagine .



## Operazioni sulla pagina

- Inserimento/modifica di una tupla (Può richiedere l'allocazione di nuovo spazio)
- Cancellazione
- Accesso ad una tupla o ad un campo di una tupla

**Il file system usato dal DBMS** I DBMS tramite i sistemi operativi gestiscono il file system per memorizzare il database. Si occupano quindi di fare l'accesso alle strutture fisiche che contengono i dati.

### 5.3.2 Strutture primarie per l'organizzazione di file

**Strutture primarie e secondarie** Le tuple organizzate all'interno dei blocchi dei file costituiscono le Strutture primarie, cioè quelle che contengono propriamente i dati. Esistono anche blocchi contenenti Strutture secondarie, che sono quelle che favoriscono l'accesso ai dati senza contenerli.

**Strutture primarie** Possono avere una organizzazione detta sequenziale. L'organizzazione sequenziale può essere:

- seriale: ordinamento fisico ma non logico
- ordinata: con ordinamento delle tuple coerente con quello di un campo
- accesso calcolato: posizione individuate tramite indice

**Organizzazione seriale** chiamata anche entry sequenced, file heap, file disordinato. Gli inserimenti vengono effettuati

- in coda (con riorganizzazioni periodiche)
- al posto di record cancellati

La sequenza delle tuple è indotta dall'ordine di immisione.

#### Strutture sequenziali

**Strutture sequenziali ordinate** Le strutture ordinate permettono ricerche binarie. Il problema è mantenere l'ordinamento.

## Con accesso calcolato

I file hash permettono un accesso diretto molto efficiente: la tecnica si basa su quella utilizzata per le tavole hash in memoria centrale.

**Definizione 136 Tavola hash:** Obiettivo: accesso diretto ad un insieme di record sulla base del valore di un campo (detto chiave, che per semplicità supponiamo identificante, ma non è necessario). Se i possibili valori della chiave sono in numero paragonabile al numero di record allora usiamo un array; ad esempio: università con 1000 studenti e numeri di matricola compresi fra 1 e 1000 o poco più e file con tutti gli studenti. Se i possibili valori della chiave sono molti di più di quelli effettivamente utilizzati, non possiamo usare l'array (spreco); ad esempio: 40 studenti e numero di matricola di 6 cifre (un milione di possibili chiavi).

Senza sprecare spazio: funzione hash:

- associa ad ogni valore della chiave un "indirizzo", in uno spazio di dimensione leggermente superiore rispetto a quello strettamente necessario
- poiché il numero di possibili chiavi è molto maggiore del numero di possibili indirizzi, la funzione non può essere iniettiva e quindi esiste la possibilità di collisioni (chiavi diverse che corrispondono allo stesso indirizzo)
- le buone funzioni hash distribuiscono le chiavi in modo causale e uniforme, riducendo le probabilità di collisione (che si riduce aumentando lo spazio ridondante)

□

| M      | M mod 50 |
|--------|----------|
| 60600  | 0        |
| 66301  | 1        |
| 205751 | 1        |
| 205802 | 2        |
| 200902 | 2        |
| 116202 | 2        |
| 200604 | 4        |
| 66005  | 5        |
| 116455 | 5        |
| 200205 | 5        |
| 201159 | 9        |
| 205610 | 10       |
| 201260 | 10       |
| 102360 | 10       |
| 205460 | 10       |
| 205912 | 12       |
| 205762 | 12       |
| 200464 | 14       |
| 205617 | 17       |
| 205667 | 17       |
| 200268 | 18       |
| 205619 | 19       |
| 210522 | 22       |
| 205724 | 24       |
| 205977 | 27       |
| 205478 | 28       |
| 200430 | 30       |
| 210533 | 33       |
| 205887 | 37       |
| 200138 | 38       |
| 102338 | 38       |
| 102690 | 40       |
| 115541 | 41       |
| 206092 | 42       |
| 205693 | 43       |
| 205845 | 45       |
| 200296 | 46       |
| 205796 | 46       |
| 200498 | 48       |
| 206049 | 49       |

Figura 5.11: 40 record, tavola hash con 50 posizioni: 1 collisione a 4, 2 collisioni a 3, 5 collisioni a 2

**Risoluzione delle collisioni** varie tecniche:

- posizioni successive disponibili
- tabella di overflow (gestita in forma collegata)
- funzioni hash "alternative"

**Nota 14:** • *le collisioni ci sono (quasi) sempre*

- *le collisioni multiple hanno probabilità che decresce al crescere della molteplicità*
- *la moltemplicità media delle collisioni è molto bassa*

**Hash su file** L'idea è la stessa ma si sfrutta l'organizzazione in blocchi e il fatto che l'accesso è al blocco. In questo modo si "ammortizzano" le probabilità di collisione

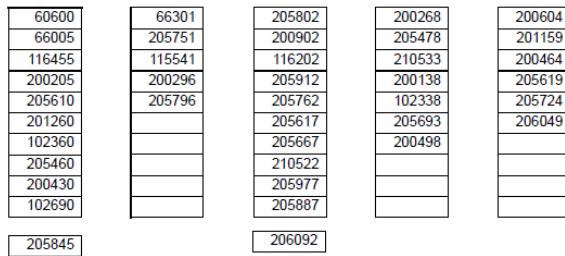


Figura 5.12: esempio di file hash: dati di dati dell'esempio precedente file hash con fattore di blocco 10; 5 blocchi con 10 posizioni ciascuno, la funzione hash restituisce un indirizzo tra 0 e 4: due soli overflow.

Il record che configlge si memorizza nel blocco successivo: Quando si tratterà di trovarlo si dovranno fare due accessi a blocchi, In tutti gli altri casi si dovrà fare un solo accesso.

Il file hash quindi:

- È l'organizzazione più efficiente per l'accesso diretto basato su valori della chiave con condizioni di uguaglianza (accesso puntuale): costo medio di poco superiore all'unità (il caso peggiore è molto costoso)
- le collisioni (overflow) sono di solito gestite con blocchi collegati aggiunti a quelli iniziali
- non è efficiente per ricerche basate su intervalli (né per ricerche bastate su altri attributi)
- i file hash "degenerano" se si riduce lo spazio: sovrabbondante: funzionano solo con file la cui dimensione non varia molto nel tempo

### 5.3.3 Strutture ad albero (non sequenziali)

Sono strutture basate sull'uso di puntatori ma l'accesso è in base ai valori di uno o più campi. Si utilizzano sia come strutture primarie che secondarie.

#### Indici

**Definizione 137 indice:** struttura per l'accesso (efficiente) ai record sulla base dei valori di un campo (o di una "concatenazione di campi") detto chiave (o, meglio, pseudochiave, perché non è necessariamente identificante). Un indice I di un file f è un altro file, con record a due campi: chiave e indirizzo (dei record di f o dei relativi blocchi), ordinato secondo i valori della chiave Ad es. l'indice analitico di un libro: lista di coppie (termine,pagina), ordinata alfabeticamente sui termini, posta in fondo al libro e separabile da esso □

## Tipi di indice .

- indice primario: su un campo sul cui ordinamento è basata la memorizzazione  
**def libro:** Dato un file  $f$  con un campo chiave  $k$  l'indice viene detto primario quando l'indice contiene al suo interno i dati oppure è realizzato su un file ordinato sullo stesso campo su cui è definito l'indice stesso.
- indice secondario: su un campo con ordinamento diverso da quello dell'ordinamento di memorizzazione  
**def libro:** Dato un file  $f$  con un campo chiave  $k$  l'indice viene detto secondario quando l'indice è un altro file in cui ciascun record è logicamente composto di due campi: uno contenente un valore della chiave  $k$  del file  $f$  e l'altro contenente l'/gli indirizzo/i fisici dei record di  $f$  che hanno quel valore di chiave.
- indice denso: contiene un record per ciascun record del file a cui si riferisce
- indice sparso: contiene un numero di record inferiore rispetto a quelli del file

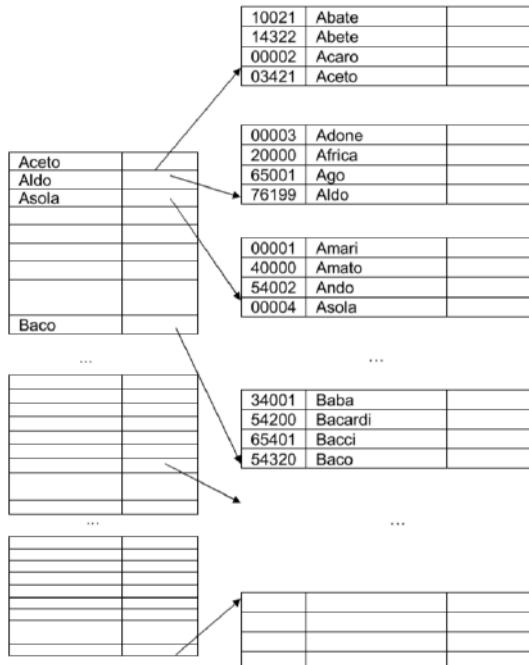


Figura 5.13: un indice primario sparso

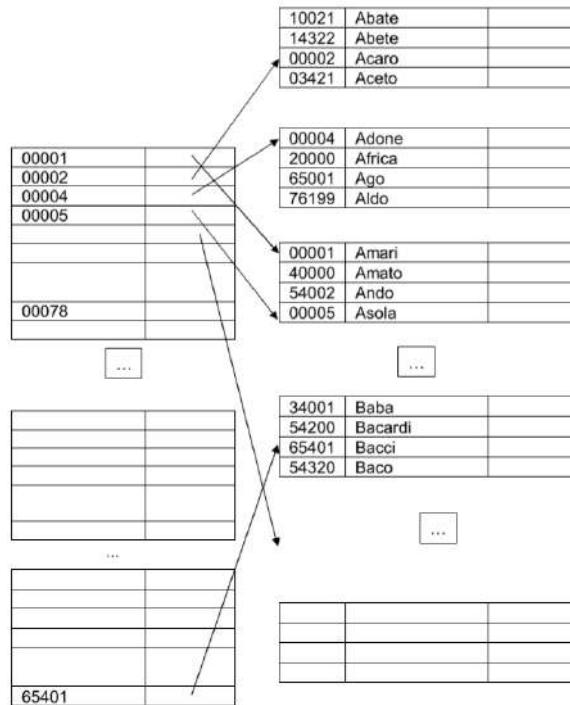


Figura 5.14: un indice secondario denso

**Nota 15 :** commenti: .

- *Un indice primario può essere sparso (non tutti i valori della chiave compaiono nell'indice) (Esempio, sempre rispetto ad un libro: indice generale (cap.1, sez.1))*
- *Gli indici secondari sono densi perché tutti i valori della chiave secondaria devono essere raggiungibili*
- *Ogni file può avere al più un indice primario e un numero qualunque di indici secondari (su campi diversi). (Esempio: una guida turistica può avere l'indice dei luoghi e quello degli artisti)*

**Caratteristiche degli indici .**

- accesso diretto (sulla chiave) efficiente (sia puntuale sia per intervalli)
- scansione sequenziale ordinata efficiente
- modifiche della chiave, inserimenti, eliminazioni inefficienti (come nei file ordinati)

Tecniche per alleviare i problemi:

- marcature per le eliminazioni
- riempimento parziale
- blocchi collegati (non contigui)
- riorganizzazioni periodiche

### Dimensioni dell'indice .

- T numero di record nel file
- B dimensione dei blocchi
- R lunghezza dei record (fissa)
- K lunghezza del campo chiave
- P lunghezza degli indirizzi (ai blocchi)

Numero di blocchi per il file (circa):  $N_B = T * R / B$

Numero di blocchi per un indice denso:  $N_D = T * (K + P) / B$

Numero di blocchi per un indice sparso:  $N_S = N_B * (K + P) / B$

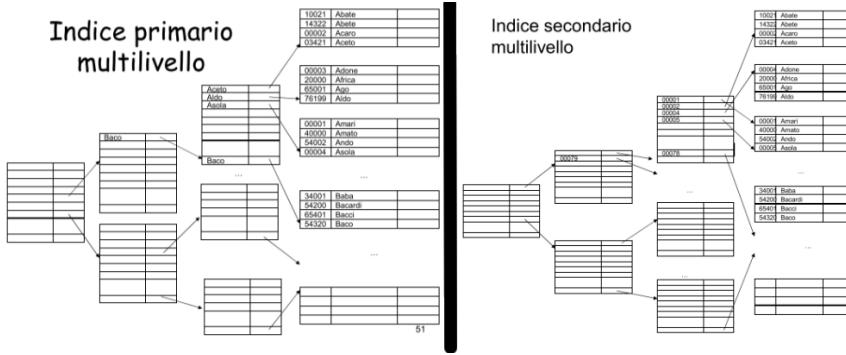
**Nota 16 :** Osservazioni su indici secondari: *si possono usare, come detto, puntatori ai blocchi oppure puntatori ai record:*

- *i puntatori ai blocchi sono più compatti*
- *i puntatori ai record permettono di semplificare alcune operazioni (effettuate solo sull'indice, senza accedere al file se non quando indispensabile)*

**Indici multilivello** Gli indici sono file essi stessi e quindi ha senso costruire indici sugli indici, per evitare di fare ricerche fra blocchi diversi. Possono esistere più livelli fino ad avere il livello più alto con un solo blocco; i livelli sono di solito abbastanza pochi, perché l'indice è ordinato, quindi l'indice sull'indice è sparso e perché i record dell'indice sono piccoli.

$N_j$  numero di blocchi al livello j dell'indice (circa):

$$N_j = N_{j-1} * (K + P) / B$$



**Problemi degli indici** Le strutture di indice basate su strutture ordinate sono poco flessibili in presenza di elevata dinamicità. Gli indici utilizzati dai DBMS sono in generale indici dinamici multilivello efficienti anche in caso di aggiornamenti. Vengono memorizzati e gestiti come B-tree (intuitivamente: alberi di ricerca bilanciati) (Alberi binari di ricerca, Alberi n-ari di ricerca, Alberi n-ari di ricerca bilanciati).

### Tipi di albero

**Albero binario di ricerca** Albero binario etichettato in cui per ogni nodo il sottoalbero sinistro contiene solo etichette minori di quella del nodo e il sottoalbero destro etichette maggiori. Tempo di ricerca (e inserimento), pari alla profondità: – logaritmico nel caso “medio” (assumendo un ordine di inserimento casuale).

### Albero di ricerca di ordine P

- Ogni nodo ha (fino a) P figli e (fino a) P-1 etichette, ordinate
- Nell’i-esimo sottoalbero abbiamo tutte etichette maggiori della (i-1)-esima etichetta e minori della (i+1)-esima
- Ogni ricerca o modifica comporta la visita di un cammino radice- foglia
- In strutture fisiche, un nodo può corrispondere ad un blocco
- Il legame tra nodi è dato da puntatori che collegano le pagine
- Ogni nodo ha numero di discendenti abbastanza grande per cui gli alberi hanno un numero limitato di livelli; la maggior parte delle pagine è nei nodi foglia
- gli alberi sono bilanciati

**B-tree** Un B-tree è un albero di ricerca che viene mantenuto bilanciato, grazie a:

- Riempimento parziale (mediamente 70%)
- Riorganizzazioni (locali) in caso di sbilanciamento

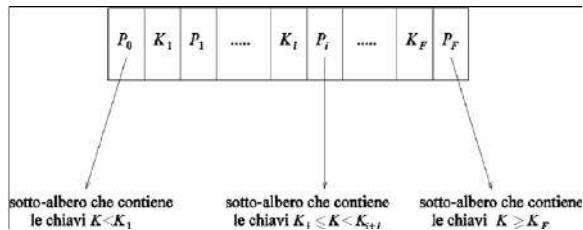


Figura 5.15: Organizzazione dei nodi del B-tree: Sequenza di F valori ordinati di chiave; Ogni  $K_i$  seguito da un puntatore  $P_i$ ; F dipende dall'ampiezza della pagina e dalla dimensione occupata dai valori di chiave e di puntatore

## Ricerca sul B-tree

- **Dato un valore V**
  - Si seguono i puntatori partendo dalla radice. Ad ogni nodo intermedio
    - Se  $V < K_1$  si segue il puntatore  $P_0$
    - Se  $V \geq K_F$  si segue il puntatore  $P_F$
    - Altrimenti si segue il puntatore  $P_j$  t.c.  $K_j \leq V < K_{j+1}$
  - La ricerca prosegue fino ai nodi foglia dell'albero

**Inserimenti e eliminazioni** Inserimenti ed eliminazioni provocano aggiornamento degli indici e sono precedute da una ricerca fino ad una foglia:

- Per gli inserimenti, se c'è posto nella foglia, ok, altrimenti il nodo va suddiviso, con necessità di un puntatore in più per il nodo genitore; se non c'è posto, si sale ancora, eventualmente fino alla radice. Il riempimento rimane sempre superiore al 50%
- Per le eliminazioni, è possibile avere una riduzione di nodi.
- Modifiche del campo chiave vanno trattate come eliminazioni seguite da inserimenti

**B tree e B+ tree** B+ tree: le foglie sono collegate in una lista e sono molto usati nei SBMs perché permettono una ricerca efficiente su intervalli. B tree: i nodi intermedi possono avere puntatori direttamente ai dati.

#### 5.3.4 Strutture fisiche e indici nei DBMS relazionali

##### Definizione degli indici in SQL

Non è standard ma presente in forma simile nei vari DBMS:

- `create[unique]index IndexName on TableName(AttributeList)`
- `drop index IndexName`

##### Strutture fisiche nei DBMS relazionali

- struttura primaria
  - disordinata (heap, "unclustered")
  - ordinata ("clustered")
  - hash ("clustered")
- indici:
  - organizzazione sequenziale (statica) di solito su struttura ordinata
  - b-tree (dinamico)

##### Oracle

##### DB2

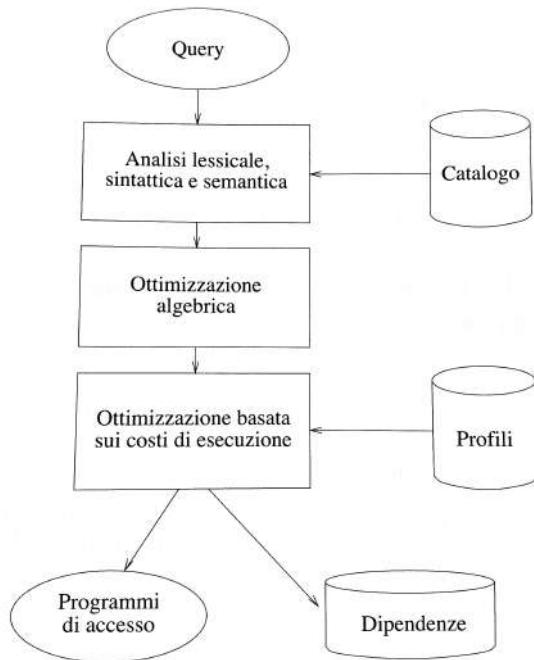
##### SQL server

#### 5.3.5 Gestione delle interrogazioni: ottimizzazione

**Definizione 138 Query processor (o ottimizzatore):** è un modulo del DBMS: le interrogazioni sono espresse ad alto livello (insiemi di tuple, poca proceduralità) e l'ottimizzatore sceglie la strategia realizzativa a partire dall'istruzione SQL.

(l'ottimizzatore agisce a tempo di compilazione) □

## Processo di ottimizzazione



Dopo l'analisi lessicale ecc. avviene l'ottimizzazione algebrica.

**Ottimizzazione algebrica** I DBMS cercano di eseguire espressioni equivalenti a quelle date, ma meno "costose".

Euristica fondamentale: selezioni e proiezioni il più presto possibile (per ridurre le dimensioni dei risultati intermedi):

- "push selection down"
- "push projections down"

poi avviene un'ottimizzazione che dipende sia dalla tipologia dei metodi di accesso sia dal modello dei costi assunto. Infine avviene la generazione del codice che utilizza i metodi di accesso ai dati: si ottengono cioè dei *programmi di accesso* in formato oggetto o interno che richiedono l'uso delle strutture dati fornite dal sistema (tra cui gli indici).

## Profili della relazioni

Ogni DBMS possiede informazioni quantitative (relative alle caratteristiche delle tabelle), organizzate in strutture dati e dette *profili delle relazioni* che

sono memorizzate nel catalogo (dizionario dei dati) e aggiornate con comandi del tipo “update statistics”. I profili contengono alcune delle seguenti informazioni:

- cardinalità di ciascuna relazione (tabella)
- dimensioni delle tuple
- dimensioni dei valori (attributi)
- numero di valori distinti degli attributi
- valore minimo e massimo di ciascun attributo

Dopo l'ottimizzazione algebrica (si ottengono tutte le espressioni minimali equivalenti) va fatta quella in base ai costi. In questa fase finale dell'ottimizzazione, occorre valutare il numero di trasferimenti in memoria da fare e stimare le dimensioni dei risultati intermedi.

### Esecuzione delle operazioni

I DBMS implementano gli operatori dell'algebra relazionale (o meglio, loro combinazioni) per mezzo di operazioni di livello abbastanza basso. Operatori fondamentali:

- accesso diretto
- scansione

A livello più alto:

- ordinamento

A livello ancora più alto:

- join, l'operazione più costosa

**Definizione 139 Ottimizzazione basata sui costi:** è un problema articolato con scelte relative a:

- operazioni da eseguire (es.: scansione o accesso diretto?)
- i dettagli del metodo (es.: quale metodo di join)
- ordine delle operazioni (es. join di tre relazioni; ordine?)

Architetture parallele e distribuite aprono ulteriori gradi di libertà. □

**Definizione 140 Accesso diretto:** Si usa il termine accesso diretto quando è possibile leggere o scrivere un record senza dover necessariamente esaminare il file in modo sequenziale ma è possibile ottenere in altro modo l'indirizzo del blocco in cui il record si trova. □

Può essere eseguito solo se le strutture fisiche lo permettono (indici, strutture hash).

**Definizione 141 Accesso diretto basato su indice:** È efficace per :

- interrogaizioni sulla "chiave dell'indice" "puntuali" ( $A_i=v$ ) e su intervallo ( $v_1 \leq A_i \leq v_2$ )
- per predicati congiuntivi: si sceglie il più selettivo per l'accesso diretto e si verifica poi sugli altri dopo la lettura (e quindi in memoria centrale)
- per predicati disgiuntivi: servono indici su tutti, ma conviene usarli se molto selettivi e facendo attenzione ai duplicati

□

**Definizione 142 Accesso diretto basato su hash:** È efficace:

- per interrogazioni sulla "chiave dell'indice" "puntuali" ( $A_i=v$ ) e NON su intervallo ( $v_1 \leq A_i \leq v_2$ )
- Per predicati congiuntivi e disgiuntivi, vale lo stesso discorso fatto per gli indici

□

**Definizione 143 Indice hash su più campi:**

□

**Definizione 144 Ricerca (scansione):** L'operatore di selezione esprime la ricerca su una relazione; è possibile implementarlo tramite un algoritmo di ricerca completa la cui complessità media, se la relazione ha  $n$  tuple, è lineare e uguale a  $n/2$ . Tutti i blocchi vanno trasferiti dalla memoria secondaria al buffer

□

### Altri metodi

Se le tuple sono ordinate e la selezione è fatta sull'attributo su cui la relazione è ordinata e i blocchi sono memorizzati contigui, si può ottenere un numero medio di trasferimenti logaritmico  $\log_2 n$ . Si possono usare gli indici

**Definizione 145 Ordinamento:** Ci sono query che richiedono un risultato ordinato. L'ordinamento serve per implementare efficientemente alcune operazioni come il join, ad esempio se la relazione sta tutta in memoria si può usare il quicksort altrimenti il mergesort

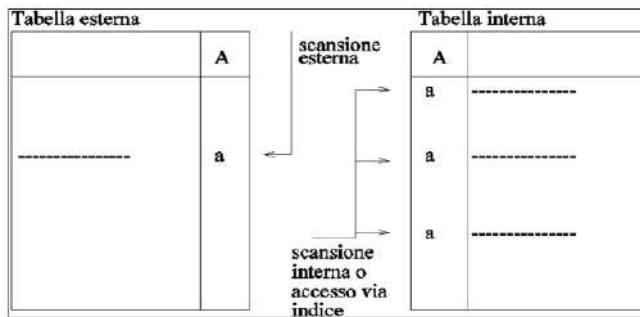
□

**Metodi di join** sono:

- Nested loop
- Merge scan
- Hash-based

ognuno ha un costo in termini di accessi al disco

**Definizione 146 Nested loop:** per ogni tupla nella tabella esterna si esaminano tutte le tuple di quella interna per verificare la condizione di join. Date R ed S, si hanno due possibilità R esterna o R interna. Il **costo**, in termini di trasferimenti in memoria, dipende dal numero di accessi e dal fatto che la tabella interna sia piccola  $\square$



**Esempio 1 Costo per nested loop join:** Supponiamo la tabella R abbia 10.000 record che occupano complessivamente 400 blocchi. La tabella S contiene 5.000 record che occupano 100 blocchi. Se R è esterna e nel buffer si può mettere solo un record alla volta, il numero di trasferimenti sarà  $N=10000*100+400$ . Se invece nel buffer entra tutta S,  $N=400+100$   $\square$

**Definizione 147 Merge scan:** Si ordinano le tabelle in base agli attributi di join. Si trova l'elemento della seconda tabella da cui partire rispetto al primo elemento della prima tabella e poi si continua da lì. Il costo è l'ordinamento. Si usa per il join naturale o l'equi-join. Ogni blocco deve essere letto una sola volta (assumendo che tutte le tuple per un dato valore di join stiano insieme nel buffer (duplicati)). Il costo del metodo merge scan è:

- $B_R + B_S$  trasferimenti in memoria
- + il costo dell'ordinamento se le relazioni sono non ordinate

$\square$

**Definizione 148 hash join:** Utile solo per join naturale e equi-join la funzione hash  $h$  sull'attributo di join è usata per partizionare le tuple di entrambe le relazioni. Le partizioni sono inserite in tabelle aggiuntive,  $h$  produce partizioni di S tali che ognuna sta in memoria, R è partizionata di conseguenza. Serve memoria ed è migliore del nested loop nel caso di equijoin  $\square$

**Costo dell'hash join:** Le relazioni R e S vengono partizionate sulla base del valore della funzione hash, questo richiede la lettura e la scrittura completa delle due relazioni, ovvero  $2(B_R + B_S)$  trasferimenti di blocco. Per ogni tupla  $t_r$  in ogni partizione di R, si considera la tupla  $t_s$  nella partizione

corrispondente secondo la funzione hash, quindi si legge ciascuna partizione una volta ( $B_R + B_S$ ) trasferimenti di blocco. Costo complessivo:  $3(B_R + B_S)$  trasferimenti di blocco.

### Costo di una query

**Misura del costo di una query** Per concludere, molti fattori contribuiscono al costo di una query, cioè il tempo necessario per avere la risposta:

- Gli accessi al disco, il tempo di CPU o il tempo di rete
- L'accesso al disco è il tempo predominante ed è anche facilmente calcolabile considerando
  - numero di scansioni
  - numero di letture
  - numero di scritture

Per semplicità consideriamo un fattore  $t$  unico come tempo per accedere un blocco

## Costo di una query

- Oltre al numero di trasferimenti bisogna considerare anche la memoria che serve per memorizzare i risultati intermedi
- Ad esempio l'hash join necessita di tabelle intermedie
- Operazioni a più operandi devono essere eseguite un passo alla volta

93

93

## Congiunzione di condizioni

- Una congiunzione di operazioni di selezione può essere scomposta in una sequenza di selezioni semplici
  - $\sigma_{\theta_1 \wedge \theta_2}(R) = \sigma_{\theta_1}(\sigma_{\theta_2}(R))$
- Le operazioni di selezione sono commutative
  - $\sigma_{\theta_1}(\sigma_{\theta_2}(R)) = \sigma_{\theta_2}(\sigma_{\theta_1}(R))$
- l'ordine viene scelto in base alla dimensione del risultato intermedio, prima quella più selettiva

94

94

47

## Ordine dei join

- l'ordine in cui si fanno i join dipende dalla dimensione dei risultati intermedi
- Date le relazioni  $R1, R2, R3$ , per la associatività  $(R1 \bowtie R2) \bowtie R3 = R1 \bowtie (R2 \bowtie R3)$
- se  $R2 \bowtie R3$  ha una dimensione molto grande, mentre  $R1 \bowtie R2$  è piccolo, possiamo scegliere di eseguire  $(R1 \bowtie R2) \bowtie R3$  in modo da dover memorizzare una relazione intermedia più piccola.

95

95

## Ordine dei join cont.

- $\Pi_{customer\_name}((\sigma_{branch\_city= "Brooklyn"}(branch)) \bowtie (account \bowtie depositor))$ 
  - $account \bowtie depositor$  è probabilmente una relazione con molte tuple, ma solo pochi di tutti i clienti di banca hanno un conto in una filiale di Brooklyn, è quindi meglio calcolare prima
  - $\sigma_{branch\_city= "Brooklyn"}(branch) \bowtie account$

96

96

48

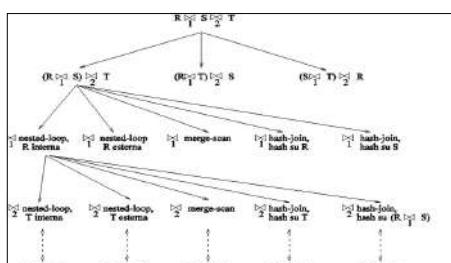
## Il processo di ottimizzazione

- Si costruisce un albero di decisione con le varie alternative ("piani di esecuzione")
- Si valuta il costo di ciascun piano
- Si sceglie il piano di costo minore
- L'ottimizzatore trova di solito una "buona" soluzione, non necessariamente l'"ottimo"

97

97

## Un albero di decisione



98

98

49

### Esempio

- Calcoliamo il piano di esecuzione migliore per la seguente interrogazione dal punto di vista della dimensione dei risultati intermedi

$\pi_{Tit}(\sigma_{(Cat=D \vee Cat=B) \wedge An=X \wedge Naz=Y}(F \triangleright \triangleleft Ac \triangleright \triangleleft I))$

$F(Fid, Titolo, Anno, Categoria,..)$   
 $Ac(Aid, Nazionalità,..)$   
 $I(Fid, Aid,..)$

- 

99

99

- $N(F)=30000$
- $N(A)=2000$
- $N(I)=600000$
- $N(Fid, I)=30000$
- $N(Aid, I)=2000$
- $N(Nazionalità, Ac)=4$
- $N(Categoria, F)=5$
- $N(Anno, F)=20$

100

100

50

## Ottimizzazione algebrica

- $\pi_T(\pi_{T, \text{FId}}(\sigma_{(Cat=D \vee Cat=B) \wedge An=X}(F)) \triangleright \triangleleft \pi_{A\text{Id}}(\sigma_{NaZ=Y}(Ac)) \triangleright \triangleleft \pi_{Fid, A\text{Id}}(I))$

101

101

## Dimensione delle varie relazioni

- $R1 = \sigma_{(Cat=D \vee Cat=B)}(F)$ 
  - $R11 = \sigma_{(Cat=D)}(F)$        $R12 = \sigma_{(Cat=B)}(F)$
  - $n_{R11} = N(F)/N(Categoria, F) = 30000/5 = 6000 = n_{R12}$
  - $n_{R1} = 6000 * 2 = 12000$
- $R2 = \sigma_{An=X}(F)$ 
  - $n_{R2} = N(F)/N(An, F) = 30000/20 = 1500$

102

102

51

### *Cont.*

- L'ordine più conveniente è
  - $R3 = \sigma_{(C=D \vee C=B)}(\sigma_{A=X}(F))$
- Il numero finale delle tuple è
  - $n_{R3} = n_{R2} / N(\text{Categoria}, R2) * 2 = 1500 / 5 * 2 = 300 * 2$
- $R4 = \pi_{T, FId}(R3)$  Fid chiave, ma dopo la selezione non tutti i valori di FId in F stanno in R4, di conseguenza non valgono più i vincoli di integrità referenziale
  - $n_{R4} = n_{R3} = 600$

103

103

### *Cont.*

- $R5 = (\sigma_{Naz=X}(Ac))$ 
  - $n_{R5} = N(Ac) / N(\text{Nazionalità}, Ac) = 2000 / 4 = 500$
- $R6 = \pi_{AId}(R5)$ 
  - $n_{R6} = n_{R5} = 500$  Aid chiave
- $R7 = \pi_{FId, AId}(I)$ 
  - $n_{R7} = N(I) = 600000$

104

104

52

### *Cont.*

- $\pi_T(R4 \triangleright\triangleleft R6 \triangleright\triangleleft R7)$
- $R4 \triangleright\triangleleft R6$ 
  - $500*600=300000$  tuple non c'è relazione tra F e Ac, prodotto cartesiano
- $R6 \triangleright\triangleleft R7$  il vincolo di integrità tra I e Ac non vale più, però Aid è chiave in R6 quindi il numero di tuple è  $<|R7|$ , più precisamente
  - $\min(500*600000/2000, 600000*1)=$   
 $\min(150000, 600000)=150000$
- Perché?

105

105

### *Cont.*

- $500*600000/2000$ 
  - Rappresenta i 500 Aid che si combinano (ognuno) con il gruppo dentro I con Aid uguale, la cui dimensione è 600000 diviso i diversi Aid in I, ovvero 2000
- $600000*1$ 
  - Rappresenta ogni elemento di I che si combina con i corrispondenti di R6 che sono 500 diviso il numero di Aid diversi, che sono pure 500 essendo Aid chiave.

106

106

53

- $R4 \triangleright \triangleleft R7$  come sopra si è perso il vincolo di integrità referenziale tra I ed F su Fid, ma Fid è chiave in  $R4$ , quindi il numero di tuple sarà  $< |R7|$

–  $\min(600*600000/30000, 600000*1) =$   
 $\min(12000, 600000) = 12000$

107

107

## Cardinalità del join

- Se il join di  $R1, R2$  è completo (i.e., ogni tupla di entrambe le relazioni contribuisce ad almeno una tupla del risultato)
  - allora il numero delle tuple risultanti sarà maggiore od uguale al massimo fra  $|R1|$  e  $|R2|$
  - Se però gli attributi di join contengono una chiave di  $R2$ , allora il numero delle tuple finali sarà minore od uguale a  $|R1|$ .
  - Se gli attributi di join sono la chiave primaria di  $R2$  ed esiste un vincolo di integrità referenziale tra questi in  $R1$  e tale chiave di  $R2$ , allora il numero delle tuple finali sarà uguale a  $|R1|$ .

108

54

### *Cont.*

- Quindi l'ordine sarà
- $\pi_T((R4 \triangleright\triangleleft R7) \triangleright\triangleleft R6)$

109

109

### **Progettazione fisica**

- La fase finale del processo di progettazione di basi di dati
- **input**
  - lo schema logico e informazioni sul carico applicativo
- **output**
  - schema fisico, costituito dalle definizione delle relazioni con le relative strutture fisiche (e molti parametri, spesso legati allo specifico DBMS)

110

55

123

## Progettazione fisica nel modello relazionale

- La caratteristica comune dei DBMS relazionali è la disponibilità degli indici:
  - la progettazione logica spesso coincide con la scelta degli indici (oltre ai parametri strettamente dipendenti dal DBMS)
- Le chiavi (primarie) delle relazioni sono di solito coinvolte in selezioni e join: molti sistemi prevedono (oppure suggeriscono) di definire indici sulle chiavi primarie
- Altri indici vengono definiti con riferimento ad altre selezioni o join "importanti"
- Se le prestazioni sono insoddisfacenti, si "tara" il sistema aggiungendo o eliminando indici
- È utile verificare se e come gli indici sono utilizzati con il comando SQL `show plan`

111

56

## Capitolo 6

# Transazioni in SQL

### istruzioni fondamentali:

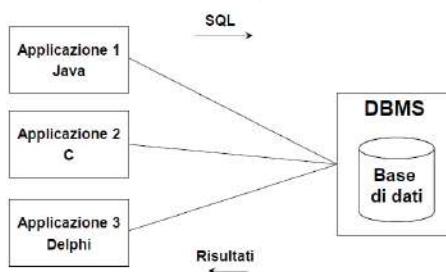
- begin transaction: specifica l'inizio della transazione (le operazioni non vengono eseguite sulla base di dati)
- commit work: le operazioni specificate a partire dal begin transaction vengono eseguite sulla base di dati
- rollback work: si rinuncia all'esecuzione delle operazioni specificate dopo l'ultimo begin transaction

### 6.1 SQL nei linguaggi di programmazione

In applicazioni complesse l'utente non vuole eseguire solo comandi SQL ma programmi, sono quindi necessarie altre funzionalità per gestire:

- input (scelte dell'utente e parametri)
- output (con dati che non sono relazioni o se si vuole una presentazione complessa)
- il controllo

Applicazioni ed SQL: architettura



### 6.1.1 Differenze tra SQL e altri linguaggi:

- accesso ai dati e correlazione:
  - linguaggio: dipende dal paradigma e dai tipi disponibili (es. scansione liste)
  - SQL: join
- tipi di base:
  - linguaggio: numeri, stringhe, bool...
  - SQL: CHAR, VARCHAR, DATE...
- costruttori di tipo:
  - linguaggio: dipende dal paradigma
  - SQL: relazioni e ennuple

### 6.1.2 Tecniche principali

- SQL immerso (embedded sql):
  - sviluppato dagli anni 70
  - sql statico
- sql dinamico
- Call Level Interface (CLI)
  - più recente
  - SQL/CLI, ODBC, JDBC

**Definizione 149 SQL immerso:** le istruzioni sql sono immerse nel programma redatto nel linguaggio "ospite".

Un precompilatore (legato al DBMS) viene usato per analizzare il programma e tradurlo in un programma nel linguaggio ospite (sostituendo le istruzioni sql con chiamate alle funzioni di una API del DBMS) □

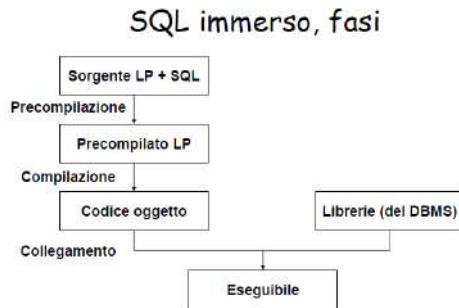


Figura 6.1: il precompilatore è specifico della combinazione linguaggio-DBMS-sistema operativo

### SQL immerso, un esempio

```

#include<stdlib.h>
main(){
    exec sql begin declare section;
    char *NomeDip = "Manutenzione";
    char *CittaDip = "Pisa";
    int NumeroDip = 20;
    exec sql end declare section;
    exec sql connect to utente@librobd;
    if (sqlca.sqlcode != 0) {
        printf("Connessione al DB non riuscita\n");
    } else {
        exec sql insert into Dipartimento
            values(:NomeDip,:CittaDip,:NumeroDip);
        exec sql disconnect all;
    }
}
  
```

Note immagine 6.1.2:

- EXEC SQL denota le porzioni di interesse del precompilatore:
  - definizioni dei dati
  - istruzioni sql
- le variabili del programma possono essere usate come "parametri" nelle istruzioni sql (precedute da ":") dove sintatticamente sono ammesse costanti
- sqlca è una struttura dati per la comunicazione fra programma e dbms
- sqlcode è un campo di sqlca che mantiene il codice di errore dell'ultimo comando sql eseguito:
  - zero: successo
  - altro valore: errore o anomalia

**Definizione 150 Conflitto di impedenza (impedance mismatch):** linguaggi di programmazione: operazioni su singole variabili o oggetti, si può accedere agli elementi di una tabella uno alla volta

SQL: operazioni su tabelle che restituiscono tabelle (es assegnare un result set a una variabile) □

Interrogazioni in sql immerso: conflitto di impedenza: il risultato di una select è costituito da zero o più ennupla; se il risultato sono zero o una ennupla può essere gestito in un record, se sono più di una come si fa?

Metodo del cursore: tecnica per trasmettere al programma una ennupla alla volta

**Definizione 151 Cursore:** accede a tutte le ennupla di una interrogazione in modo globale (tutte insieme o a blocchi, è il dbms che sceglie la strategia efficiente) trasmettendole al programma una alla volta □

### Operazioni sui cursori

```
Definizione del cursore
declare NomeCursore [ scroll ] cursor for Select ...
Esecuzione dell'interrogazione
open NomeCursore
Utilizzo dei risultati (una ennupla alla volta)
fetch NomeCursore into ListaVariabili
Disabilitazione del cursore
close cursor NomeCursore
Accesso alla ennupla corrente (di un cursore su singola
relazione a fini di aggiornamento)
current of NomeCursore
nella clausola where
```

#### note:

- per aggiornamenti e interrogazioni scalari (cioè che restituiscono una sola ennupla) il cursore non serve
- i cursori possono far scendere la programmazione ad un livello troppo basso, pregiudicando la capacità dei dbms di ottimizzare le interrogazioni (se nidifichiamo due o più cursori rischiamo di reimplementare il join)

**Definizione 152 SQL dinamico:** Non sempre le istruzioni sql sono note quando si scrive il rprogramma, a tale scopo è stata definita una tecnica diversa chiamata dynamic sql che permette di eseguire istruzioni sql costruite dal programma (o ricevute dal programma attraverso parametri o da input). Non è banale gestire i parametri e la struttura dei risultati (non noti a priori) □

## SQL dinamico

- Le operazioni SQL possono essere:
  - eseguite immediatamente
    - `execute immediate SQLStatement`
  - prima "preparate":
    - `prepare CommandName from SQLStatement`
    - e poi eseguite (anche più volte):
      - `execute CommandName [ into TargetList ] [ using ParameterList ]`

**Definizione 153 Call Level Interface:** indica genericamente interfacce che permettono di inviare richieste a dbms per mezzo di parametri trasmessi a funzioni di una libreria per realizzare il colloquio con la base di dati.

Funzionamento generale:

- si usa il servizio cli che crea la connessione col dbms
- tramite la connessione si invia il comando sql da eseguire
- si ottiene la relazione risultato e la si esamina tramite le primitive della cli

□

## sql immerso vs cli

- sql immerso permette:
  - precompilaizone (e quindi efficienza)
  - uso di sql completo
- cli
  - indipendente dal dbms
  - permette di accedere a più basi di dati anche eterogenee

## **Parte II**

# **Pistolesi**

# Capitolo 7

## DB e la gestione dei dati

### 7.1 L'interrogazione

Una base di dati contiene un'enorme quantità di dati, organizzata secondo precisi criteri; proprio per questo anche l'estrazione di precisi gruppi di dati dev'essere fatta seguendo specifici criteri.

La formulazione di una **Interrogazione**, diventa quindi necessaria per l'estrazione di informazioni di interesse.

Il linguaggio utilizzato nell'articolare tali interrogazioni è l'**SQL**, un linguaggio dichiarativo.

**Definizione 154 Linguaggio dichiarativo:** A differenza dei linguaggi procedurali, in cui vengono descritti come "istruzioni" i procedimenti per ottenere un certo risultato, nei linguaggi dichiarativi si dichiara le proprietà specifiche del risultato desiderato. □

### 7.2 MySQL

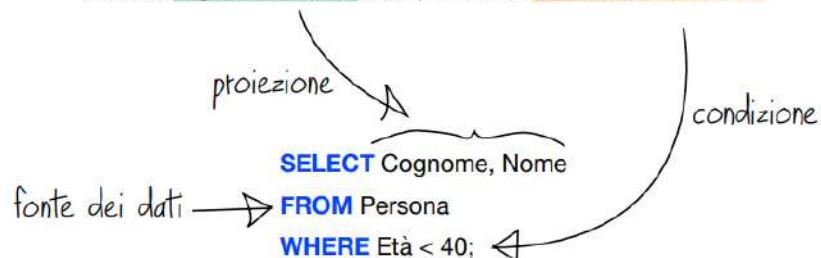
MySQL è un **DBMS relazionale** (quindi lavora con le tabelle), che rispetta la metodologia server client.

Con questo sistema le nostre "Query" assumeranno la forma:

- **SELECT**, la proiezione sugli attributi.
- **FROM**, la manipolazione della tabella.
- **WHERE**, la selezione delle tuple.

così che frasi sensate dichiarative vengano trasformate in:

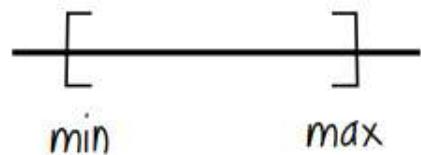
Indicare cognome e nome delle persone di età inferiore a 40 anni



**Elemento di codifica 1 Asterisco \*\*:** Elemento da usare nel **SELECT**, indica la proiezione su tutti gli attributi presenti sui dati.

**Elemento di codifica 2 Operatori logici:** nel **WHERE** la selezione delle tuple può essere fatta (metodo più utile) usando operatori logici (si considerino come espressioni booleane).

- "AND" logico
- "OR" logico
- '=' , operatore di uguaglianza (non assegnamento, non siamo in procedurale)
- '<', '>', '<=' , '>=' , operatori di confronto logici
- **Caso particolare:** "BETWEEN" si usa in alternativa di <= e >= per indicare un intervallo di valori(chiuso).



**Elemento di codifica 3 costrutto DISTINCT:** Dal momento che una Query non esclude che nel risultato vengano ripetuti elementi non chiave (perché dovrebbe), questo costrutto nel **SELECT** elimina i duplicati dal risultato, da usare ovviamente se non ci interessano.

**Elemento di codifica 4 IS NOT NULL/IS NULL:** Entrambe queste diciture sono condizioni da porre nel **WHERE**, servono a escludere/selezionare tuple con certi attributti dal valore NULL (il significato di NULL già dovresti saperlo).

## 7.3 Gestione delle date

In SQL la data è il tempo trascorso da una certa reference (anno 0).

**Elemento di codifica 5 DATE, TIMESTAMP e DATE\_FORMAT:** estraggono il dato temporale rispettivamente nei formati **YYYY-MM-DD** e **YYYY-MM-DD HH:MM:SS**, vien da se che sulle date sono applicabili operatori logici di confronto e uguaglianza; inoltre è possibile manipolare la forma di rendimento di una data attraverso **DATE\_FORMAT**, estraendo solo gli elementi a cui siamo interessati.

la sintassi di utilizzo è: **DATE\_FORMAT(attribute,'\*|\*|...|\*|\*)** , dove '\*' è un elemento dei seguenti.

| Formato | Descrizione                      |
|---------|----------------------------------|
| %Y      | anno (4 cifre)                   |
| %y      | anno (2 cifre)                   |
| %M      | nome del mese                    |
| %m      | mese (2 cifre)                   |
| %d      | giorno del mese (00-31)          |
| %W      | nome del giorno                  |
| %w      | giorno della settimana {0,...,6} |
| %T      | orario (hh:mm:ss)                |

**Elemento di codifica 6 Estrazione di giorno, mese, anno:** I costrutti **DAY**, **MONTH** e **YEAR** estraggono da una data rispettivamente giorno, mese e anno.

**Elemento di codifica 7 La data odierna:** grazie alla variabile di sistema **CURRENT\_DATE**, è possibile usare in una query la data odierna riconosciuta dal sistema (indipendentemente da quando la query viene scritta).

**Elemento di codifica 8:** Differenze fra due date]Le date in SQL non possono essere sottratte utilizzando operatori aritmetici, tutta via esistono i costrutti **DATEDIFF** (numero di giorni che separano due date) e **PERIOD\_DIFF** (numero di mesi che separano due date espresse in **DATE\_FORMAT(attr, '%Y|%m')**).

**Elemento di codifica 9 Sommare intervalli:** i costrutti **DATE\_ADD** e **DATE\_SUB** servono a sommare e sottrarre intervalli di tempo, come argomenti hanno una data e un **INTERVAL** (**INTERVAL int (oneof YEAR, MONTH, DAY)**).

Il risultato dell'operazione è una nuova data.

La forma completa della chiamata è **data = DATE\_ADD(data2, INTERVAL int (oneof...))**, nonostante ciò la somma e la sottrazione di intervalli può anche essere fatta attraverso operatori aritmetici.

**Elemento di codifica 10 Funzioni utility sulle date:** Lista di funzioni su date che potrebbero essere utili

| Funzione     | Risultato                           |
|--------------|-------------------------------------|
| dayname()    | nome del giorno                     |
| monthname()  | nome del mese                       |
| dayofweek()  | giorno della settimana in {1,...,7} |
| weekday()    | giorno della settimana in {0,...,6} |
| last_day()   | ultimo giorno del mese della data   |
| dayofyear()  | mese (2 cifre)                      |
| weekofyear() | numero della settimana (0,...,53)   |
| yearweek()   | anno e settimana                    |

## 7.4 Operatori di aggregazione

'SQUASH', questi operatori eseguono calcoli sui valori assunti da un certo attributo su un insieme di tuple, particolarità fondamentale: tutto collassa in un solo record formato da un solo attributo **numerico**.

**Elemento di codifica 11 Conteggio:** Serve a contare il numero di record presenti nella tabella argomento (tabella, sottoinsieme o combinazione), il costrutto usato è **COUNT**.

- **COUNT (\*)**, conta i record di tutto il risultato.
- **COUNT (DISTINCT attr)**, conta il numero di record con attr non uguale.

**Elemento di codifica 12 Somma:** costrutto **SUM**, somma i valori assunti dall'attributo passato per argomento su un insieme di record. **SUM(attr)**

**Elemento di codifica 13 Media aritmetica:** costrutto **AVG**, calcola la media aritmetica dei valori assunti dall'attributo passato per argomento su un insieme di record. **AVG(attr)**

N.B. è possibile rinominare l'attributo risultato (che avrà un solo record per definizione di 'SQUASH') con il costrutto **AS nome**.

**Elemento di codifica 14 Massimo e Minimo:** elementi che individuano il valore massimo o il valore minimo fra i valori assunti dall'attributo passato per argomento in un insieme di record.

- **MAX (attr)**
- **MIN (attr)**

**REGOLE DI UTILIZZO DELLO SQUASH:** è fondamentale ricordarsi che nel momento in cui eseguiamo uno squash, perdiamo tutti i dati della tabella su cui stiamo operando, in quanto inevitabilmente collassa in un solo record contenente un singolo valore numerico.  
 Quindi fare una richiesta del genere (foto sotto) non ha alcun significato, anzi è un'errore grave.

```
SELECT MAX(Reddotto), Nome, Cognome
FROM Paziente;
```

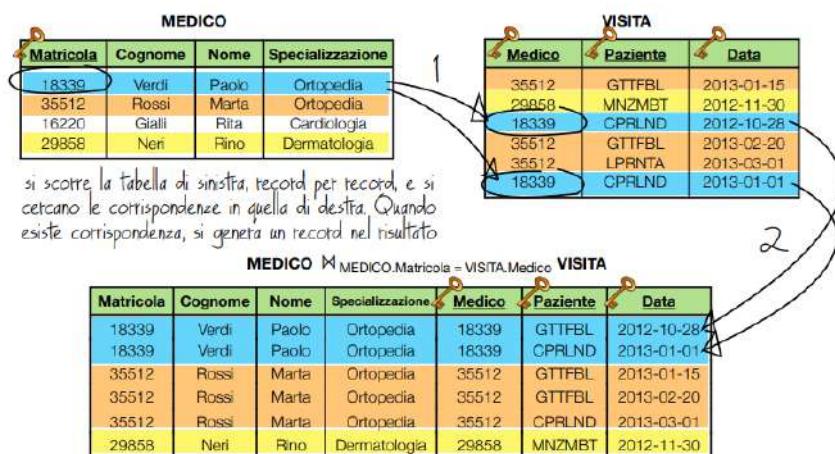
## 7.5 La gestione delle tabelle (manipolazione del FROM)

Per ovvie questioni di organizzazione e efficienza le informazioni all'interno di un database sono frammentate in più tabelle; non è affatto raro che sia necessario riunire diversi pezzi da diverse tabelle per ottenere una determinata informazione.

Esistono infatti comandi che permettono di manipolare più tabelle a nostro piacimento:

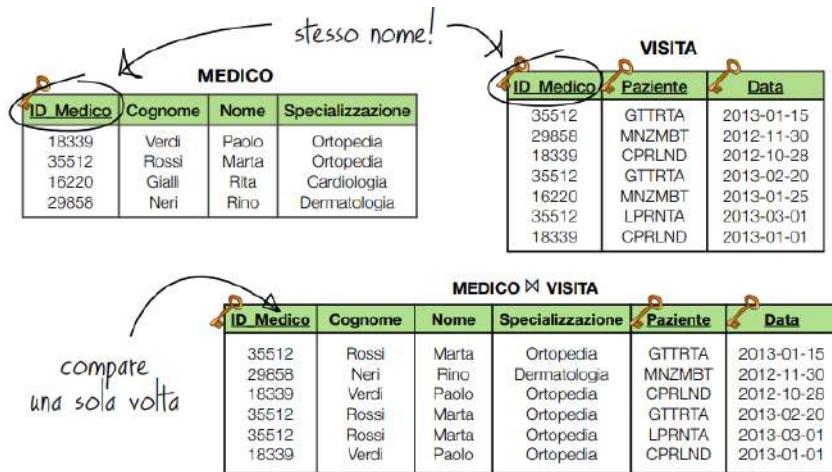
**Elemento di codifica 15 combinare più tabelle, il JOIN:** Serve a combinare due tabelle, ne esistono diversi tipi che operano con criteri differenti.

- **INNER JOIN:** Date due tabelle, combina ogni record della prima con tutti i record della seconda che verificano una determinata condizione.(su attributi)



**Sintassi MySQL:** FROM Tab1 INNER JOIN tab2 ON attr1 (operatore logico) attr2

- **JOIN NATURALE:** Combina i record della prima tabella con i record della seconda tabella aventi valori uguali sugli attributi **omonimi**, quindi attributi equivalenti, ma con nome diverso non verranno contati, è quindi necessario in questi casi usare ridefinizioni.  
Attenzione però, è altrettanto ovvio che due attributi omonimi debbano rappresentare lo stesso concetto.



Sintassi MySQL: FROM tab1 NATURAL JOIN tab2

- **Prodotto cartesiano:** Restituisce tutte le possibili combinazioni di ciascun record della prima tabella con tutti i record della seconda tabella; il join può essere pensato come un prodotto cartesiano con condizione.

Sintassi MySQL: FROM tab1 CROSS JOIN tab2

Il risultato di questa operazione sarà una tabella avente:

- $n\_attr3 = n\_attr1 + n\_attr2$
- $n\_record3 = n\_record1 * n\_record2$

- **Join esterni:** Date due tabelle, combinano ogni record della prima con tutti i record della seconda che soddisfano una condizione, mantenendo tutti i record di una delle due tabelle; nonostante siano quindi simili all'inner join, il risultato di un outer join non scarta i record che non soddisfano la condizione, così da non perdere informazioni che potrebbero essere utili. I record che non trovano corrispondenze vengono ovviamente completati con valori NULL.

- **LEFT OUTER JOIN:** Combina ogni record della tabella sinistra con i record della tabella destra che soddisfano una condizione, mantenendo tutti i record della tabella di sinistra.

| VISITA |          |            | MEDICO    |         |       |                  |
|--------|----------|------------|-----------|---------|-------|------------------|
| Medico | Paziente | Data       | Matricola | Cognome | Nome  | Specializzazione |
| 22222  | GTTRTA   | 2010-01-15 | 18339     | Verdi   | Paolo | Ortopedia        |
| 18339  | CPRLND   | 2012-10-28 | 35512     | Rossi   | Marta | Ortopedia        |
| 35512  | GTTFBBL  | 2013-02-20 | 16220     | Gialli  | Rita  | Cardiologia      |
| 18339  | CPRLND   | 2013-01-01 |           |         |       |                  |

VISITA  $\bowtie$  VISITA.Medico = MEDICO.Matricola MEDICO

| Medico | Paziente | Data       | Matricola | Cognome | Nome  | Specializzazione |
|--------|----------|------------|-----------|---------|-------|------------------|
| 22222  | GTTRTA   | 2010-01-15 | NULL      | NULL    | NULL  | NULL             |
| 18339  | CPRLND   | 2012-10-28 | 18339     | Verdi   | Paolo | Ortopedia        |
| 35512  | GTTFBBL  | 2013-02-20 | 35512     | Rossi   | Marta | Ortopedia        |
| 18339  | CPRLND   | 2013-01-01 | 18339     | Verdi   | Paolo | Ortopedia        |

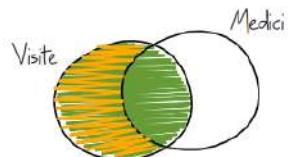
- **RIGHT OUTER JOIN:** Fa lo stesso mantenendo tutti i record della tabella a destra del join.  
Esempio:

Indicare le visite effettuate da medici che **non lavorano più** presso la clinica

```

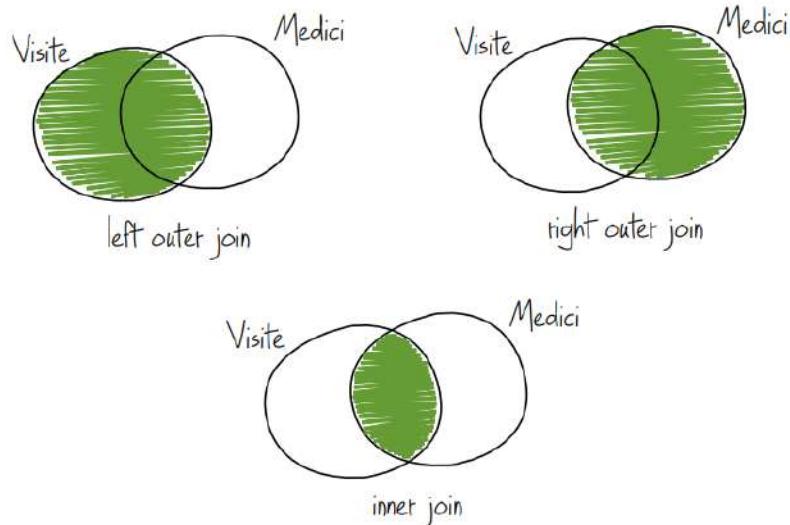
SELECT V.*
FROM Visita V
LEFT OUTER JOIN
Medico M ON V.Medico = M.Matricola
WHERE M.Matricola IS NULL
    
```

Left outer join ricava la parte verde. Da questa, il predicato where ottiene la parte arancio, cioè i record non joinabili:



**Sintassi MySQL:** FROM tab1 LEFT/RIGHT OUTER JOIN tab 2 ON condizione

Riepilogo con diagrammi sui join:



- **Join multiplo:** Il join può essere fatto su più di due tabelle, per quanto possa risultare comodo è inevitabile che sia facilmente vettore per errori nella scrittura della query in MySQL. Richiede più attenzione

**NOTA BENE:** Le selezioni sui record e le proiezioni sugli attributi avvengono dopo che le tabelle sono state manipolate a dovere, presta attenzione a come evolve l'informazione.

## 7.6 Derived table

Sono tabelle volatili che possono essere incapsulate nel FROM. Sono particolarmente utili per ottenere dei risultati intermedi, dei checkpoint.

Una derived table è il risultato di una query, che viene incapsulata in un'altra query; diventa però necessario utilizzare un alias (AS) per riferirsi alla tabella.

```

SELECT DISTINCT V1.Medico
FROM Visita V1 LEFT OUTER JOIN
(
    SELECT V2.Medico    Derived table
    FROM Visita V2
    WHERE DAYOFWEEK(V2.Data) = 4
) AS D
ON V1.Medico = D.Medico
WHERE D.Medico IS NULL;

```

## 7.7 lezione 3

### 7.7.1 Join multiplo

Il join può essere fatto (con attenzione) su un numero di tabelle maggiore di due.

**Nota 17 :** Ambiguità: *l'ambiguità si verifica se nel risultato ci sono più attributi con lo stesso nome*

La (probabile) presenza di ambiguità rende necessaria la ridenominazione che può avvenire ad esempio anche nella proiezione (ma non solo)

**Nota 18:** *anche gli alias dichiarati nel FROM effettuano una ridenominazione (essi non sono sempre necessari ma migliorano la leggibilità)*

Per leggibilità, specialmente nelle query complesse, usare **SEMPRE** la ridenominazione nel FROM

### 7.7.2 Self join

Combina i record di una tabella con i record della stessa tabella che rispettano una determinata condizione (può essere anche esterno).

### 7.7.3 Subquery

Sono query che possono essere incapsulate in un'altra query in modo correlato o non correlato. Rappresentano un'alternativa al join per query su più tabelle (a volte sono più semplici da capire e da scrivere).

## Noncorrelated subquery

**Definizione 155 Noncorrelated subquery:** Si incapsulano nel WHERE per costruire risultati (eliminati poi alla fine dell'esecuzione) che servono come elementi per determinare il risultato della query esterna.

**Nota 19:** i record della subquery non dipendono dalla query esterna

□

**Definizione 156 Risultato di una N. subquery:** Un record fa parte del risultato (=quello che vede l'utente) se un sottoinsieme dei suoi attributi si trova anche nel result set della subquery

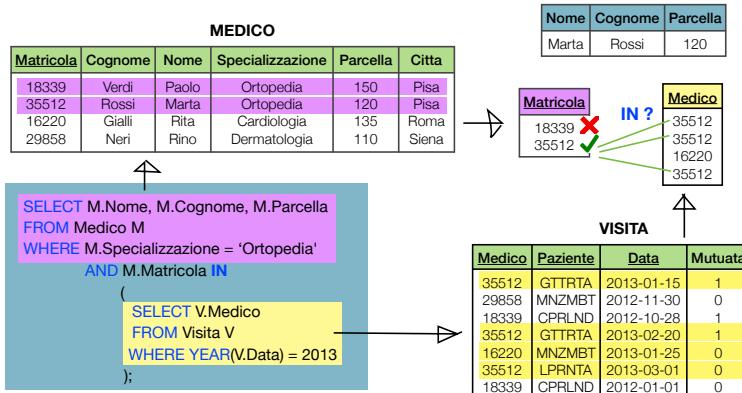
□

Per utilizzare le Noncorrelated subquery utilizziamo:

**Elemento di codifica 16 IN:** permette di controllare la presenza di un record all'interno del risultato della subquery, mentre NOT IN permette di controllare che non ci sia.

Damn! How does it work?

Indicare nome, cognome e parcella degli ortopedici che hanno effettuato almeno una visita nell'anno 2013



© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

**Definizione 157 Equivalenza join-subquery:** Data una query con subquery è sempre possibile passare alla versione basata su join e viceversa

**Nota 20:** il query optimizer sfrutta questa equivalenza per riscrivere le query il cui codice avrebbe bassa performance

□

Quando devo usare il join piuttosto che le subquery?

**Definizione 158 annidamento multiplo:** non c'è (teoricamente) limite al numero di query che si possono annidare l'una nell'altra

□

**Definizione 159 Visibilità:** In una subquery si possono riferire tutti gli attributi delle query esterne, a qualsiasi livello di annidamento esterno essi si trovino (non è vero il viceversa)  $\square$

### Subquery scalari

**Definizione 160:** restituiscono un unico record composto da un unico attributo  $\square$



**Nota 21:** Una query può essere risolta anche combinando le subquery e join

# Capitolo 8

## lezione 4

### 8.1 Raggruppamento

**Definizione 161 Raggruppamento:** Suddivide un insieme di record in gruppi di record (la tabella target viene quindi frammentata), all'interno di ognuno dei quali il valore di uno o più attributi (di raggruppamento) è costante record per record

**Nota 22:** *ogni gruppo collassa in un unico record **sempre***

□

Indicare la parcella media dei medici **di ciascuna specializzazione**

l'operatore è applicato gruppo per gruppo  
(calcola un valore riassuntivo per il gruppo!)

SELECT Specializzazione, AVG(Parcella) AS ParcellaMedia  
FROM Medico  
**GROUP BY** Specializzazione;

assume lo stesso valore in un gruppo

assume valori diversi in un gruppo,  
non si può proiettare senza aggregazione

la clausola "group by" permette di frammentare la tabella

**Nota 23 :** regola fondamentale raggruppamento: *quando si usa il raggruppamento ogni gruppo genera un solo record nel result set*

**Nota 24:** *La proiezione può contenere solo attributi presenti nel predicato di raggruppamento (perché sono i soli ad assumere un valore costante in ogni gruppo). Gli altri attributi proiettati devono essere argomento di funzioni di aggregazione!!!*

### 8.1.1 Condizioni sui gruppi

**Definizione 162 Condizioni sui gruppi:** Sono espresse esclusivamente tramite operatori di aggregazione<sup>1</sup> e permettono di scartare gruppi, qualora non siano soddisfatte.

**Nota 25:** tali condizioni sono controllate gruppo per gruppo non record per record

□

Indicare le specializzazioni della clinica **con più di due medici**

```
SELECT Specializzazione  
FROM Medico  
GROUP BY Specializzazione  
HAVING COUNT(*) > 2;
```

*ma serve distinct?/*

Figura 8.1: non serve il distinct perché specializzazione è l'attributo di raggruppamento e quindi non ha duplicati

### 8.1.2 Condizioni sui gruppi vs. condizioni sui record

Le condizioni nel **WHERE** sono applicate ai record prima del raggruppamento. Le condizioni **HAVING** sono applicate ai gruppi dopo il raggruppamento.

Indicare le specializzazioni con **più di due medici di Pisa**.

```
SELECT Specializzazione  
FROM Medico  
WHERE Citta = 'Pisa'  
GROUP BY Specializzazione  
HAVING COUNT(*) > 2;
```

---

<sup>1</sup>perché esprimono una caratteristica di un gruppo, cioè globale, sommaria, riepilogativa degli attributi variabili (non di raggruppamento) dei record che lo compongono

### 8.1.3 Raggruppamento su più attributi

Considerati i soli pazienti di Pisa, indicarne nome e cognome, e la spesa sostenuta per le visite di ciascuna specializzazione, nel triennio 2008-2010

```
SELECT M.Specializzazione, P.Nome, P.Cognome, SUM(M.Parcella)
FROM Visita V INNER JOIN Paziente P ON V.Paziente = P.CodFiscale
INNER JOIN Medico M ON V.Medico = M.Matricola
WHERE P.Citta = 'Pisa'
AND YEAR(V.Data) BETWEEN 2008 AND 2010
GROUP BY M.Specializzazione, P.CodFiscale;
```



la proiezione non contiene un sottoinsieme degli attributi di raggruppamento, ma P.Nome e P.Cognome possono essere proiettati perché esiste la dipendenza funzionale CodFiscale  $\rightarrow$  Nome, Cognome

Considerati i soli pazienti di Pisa, indicarne nome e cognome, e la spesa sostenuta per le visite di ciascuna specializzazione, nel triennio 2008-2010

```
SELECT M.Specializzazione, P.Nome, P.Cognome, SUM(M.Parcella)
FROM Visita V INNER JOIN Paziente P ON V.Paziente = P.CodFiscale
INNER JOIN Medico M ON V.Medico = M.Matricola
WHERE P.Citta = 'Pisa'
AND YEAR(V.Data) BETWEEN 2008 AND 2010
GROUP BY M.Specializzazione, P.CodFiscale, P.Nome, P.Cognome;
```

per ricordarci che siamo consci del fatto che stiamo proiettando coerentemente, si arricchisce il predicato di raggruppamento con gli attributi da proiettare che non fanno parte del group by, ma dipendono funzionalmente da CodFiscale

**Nota 26 :** regola sul raggruppamento su più attributi: *nel select possono solo esserci attributi che siano anche nel group by o il cui valore sia costante, dati gli attributi nel group by*

## 8.2 CTE

**Definizione 163 Common Table Expression:** Sono result set dotati di identificatore che possono essere usati prima di una query per costruire risultati intermedi. (sono parti di codice i cui risultati sono stoccati in memoria, nominati, e usati dalla query immediatamente dopo).

Si scrivono, separati da virgola, subito prima della query che li usa, tramite la keyword WITH. □

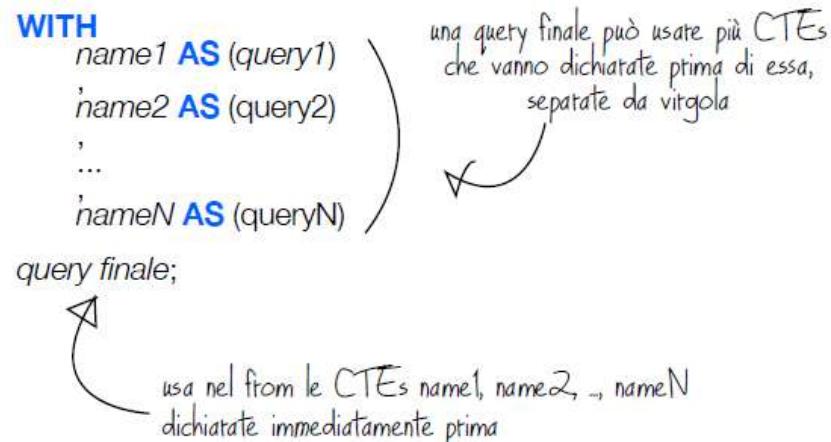


Figura 8.2: sintassi di WITH per dichiarare CTEs

Indicare il numero di pazienti di Siena, mai visitati da ortopedici

```

WITH pazienti_visitati_ortopedici AS (
    SELECT V.Paziente
    FROM Visita V
    INNER JOIN
        Medico M ON V.Medico = M.Matricola
    WHERE M.Specializzazione = 'Ortopedia'
)
SELECT COUNT(*)
FROM Paziente P
WHERE P.Citta = 'Siena'
    AND P.CodFiscale NOT IN (
        SELECT *
        FROM pazienti_visitati_ortopedici
    );
  
```

The diagram shows a complex query using a CTE. It starts with **WITH** followed by a CTE definition. This is followed by a final query that uses the CTE. Handwritten annotations identify the CTE and the part of the final query that uses it.

*CTE*

*Query che usa la CTE*

Indicare il numero di pazienti di Siena, mai visitati da ortopedici

**WITH** ortopedici AS

CTE1 → (

```
    SELECT M.Matricola AS Medico
    FROM Medico M
    WHERE M.Specializzazione = 'Ortopedia'
```

) , paz\_visitati\_ortopedici AS

CTE2 (usa la CTE1) → (

```
    SELECT V.Paziente AS CodFiscale
    FROM Visita V NATURAL JOIN ortopedici O
```

)

Query che usa la CTE2 → (

```
    SELECT COUNT(*)
    FROM Paziente P
    NATURAL LEFT OUTER JOIN
    paz_visitati_ortopedici PVO
    WHERE P.Citta = 'Siena'
    AND PVO.Matricola IS NULL;
```

# Capitolo 9

## Lezione 5

### 9.1 Correlated subquery

**Definizione 164 correlated subquery:** il loro risultato, uno per ogni record della query esterna, dipende da ciascun record della query esterna (è come una chiamata di funzione) □

**Nota 27:** *una correlated subquery è eseguita per ogni record della query esterna*

**Nota 28:** *a differenza delle noncorrelated all'interno della subquery c'è un elemento proveniente dalla query esterna*

### Costrutto EXISTS

**Definizione 165 Costrutto EXISTS:** permette di verificare che il result set di una correlated subquery contenga **almeno un record**. La sua negazione controlla che il result set sia vuoto. □

Una visita di controllo è una visita in cui un medico visita un paziente già **visitato precedentemente** almeno una volta. Indicare medico, paziente e data delle visite di controllo del mese di Gennaio 2016

```
SELECT V1.Medico, V1.Paziente, V1.Data
  FROM Visita V1
 WHERE MONTH(V1.Data) = 1
       AND YEAR(V1.Data) = 2016
       AND EXISTS
        (
          SELECT *
            FROM Visita V2
           WHERE V2.Medico = V1.Medico
                 AND V2.Paziente = V1.Paziente
                 AND V2.Data < V1.Data
        )
```

### Correlated subquery nel SELECT

Una correlated subquery può essere inserita anche nel SELECT per calcolare "al volo" un valore (DEVE essere scalare) da inserire, come attributo, nel risultato

Considerato ciascun paziente di sesso maschile, indicarne il nome e  
**il numero di visite effettuate**

```
SELECT P.CodiceFiscale, (SELECT COUNT(*)          correlated subquery
                           FROM Visita V
                           WHERE V.Paziente = P.CodFiscale) AS TotaleVisite
FROM Paziente P
WHERE P.Sesso = 'M'
```

query esterna

## 9.2 Divisione

**Definizione 166 divisione:** Operatore insiemistico derivato utile per interrogazioni che contengono condizioni esaustive espresse mediante l'avverbio tutti

□

Indicare i pazienti visitati **da tutti i medici**

```
SELECT P.CodFiscale
FROM Paziente P
WHERE NOT EXISTS
(
    SELECT *
    FROM Medico M
    WHERE NOT EXISTS
    (
        SELECT *
        FROM Visita V
        WHERE V.Medico = M.Matricola
              AND V.Paziente = P.CodFiscale
    )
);
```

Prendi ogni paziente P per il quale  
non trovi nemmeno un medico M  
che non abbia visitato P  
(cioè nemmeno un medico qui)



Figura 9.1: divisione con exists

Indicare i pazienti visitati **da tutti i medici**

```
SELECT V.Paziente
FROM Visita V
GROUP BY V.Paziente
HAVING COUNT(DISTINCT V.Medico) = (SELECT COUNT(*)
                                     FROM Medico);
```

↑  
medici diversi che hanno visitato il paziente

totale dei medici

Figura 9.2: divisione con raggruppamento

### 9.3 Differenza

**Definizione 167 differenza:** esprime il complemento di un insieme rispetto a un altro. È introdotta da proposizioni eccettuative (tranne, eccetto che...); da avverbi come “solamente”, “esclusivamente”; da congiunzioni come “ma”. In MySQL si esprime con NOT IN, NOT EXISTS, oppure tramite OUTER JOIN.

□

I pazienti visitati **solamente** dal dott. Verdi

```
SELECT DISTINCT V.Paziente
FROM Visita V INNER JOIN Medico M
ON V.Medico = M.Matricola
WHERE M.Cognome = 'Verdi'
AND V.Paziente NOT IN (
    SELECT Paziente
    FROM Visita V2 INNER JOIN Medico M2
    ON V2.Medico = M2.Matricola
    WHERE M2.Cognome <> 'Verdi');
```

Figura 9.3: differenza con NOT IN e correlated subquery

### 9.4 Modificatori

Modificano il confronto che precede una subquery, imponendone la validità per tutti oppure almeno un record del result set della subquery.

**Elemento di codifica 17 ANY:** il record esterno va nel risultato se la condizione ha successo per ALMENO UN record della subquery.

Indicare il nome e cognome dei medici la cui parcella è superiore a quella di almeno un cardiologo di Pisa

```
SELECT Nome, Cognome
FROM Medico
WHERE Parcella > ANY (
    SELECT Parcella
    FROM Medico M2
    WHERE M2.Specializzazione = 'Cardiologia'
        AND M2.Citta = 'Pisa'
);
```

**Elemento di codifica 18 ALL:** il record esterno va nel risultato se la condizione ha successo per TUTTI i record della subquery.

## 9.5 Query complesse

In un database ci sono enormi quantità di informazioni nascoste che, se individuate, permettono un'analisi profonda dei dati e della realtà.  
Una query complessa si risolve attraverso i seguenti passi:

1. **leggere** approfonditamente il testo più e più volte
2. **dividere** il problema in sotto-problemi
3. **disegnare** cherchi concettuali di ragionamento
4. **Analizzare** logicamente i sotto-problemi
5. **Tradurre** in codice ogni sotto-problema
6. **Pregare** Santa Rita
7. **Legare** logicamente le soluzioni dei sotto-problemi
8. **Scrivere** il codice della query risolutiva

## 9.6 Data manipulation

Parte del linguaggio che permette di inserire, cancellare e aggiornare interi record o valori di attributi delle tabelle di un database tramite i comandi INSERT, DELETE e UPDATE.

### 9.6.1 Inserimento

Considerata una tabella, permette di inserire un nuovo record i cui valori degli attributi possono essere sia statici (specificati dall'utente) che ricavati (ricavati dal database)

`INSERT INTO Paziente  
VALUES ('PSSLVR65R67G702U', 'Passerotti', 'Elvira', 'F',  
'1965-10-27', 'Pisa', 1500);`

Non importa specificare  
lo schema se inserisco tutti i campi  
rispettando l'ordine dello schema

Nel caso in cui non inserisci tutti i campi:

`INSERT INTO Visita(Medico, Paziente, Data)  
VALUES (010, 'slq6', CURRENT_DATE - INTERVAL 3 DAY);`

Attributi oggetto dell'inserimento

L'informazione perduta (mutuata o no)  
sarà impostata automaticamente al default value  
(il default value di un attributo si impone all'atto della creazione  
della tabella, lo vedremo più avanti...)

Figura 9.4: esempio di utilizzo del default value

#### Sintassi INSERT

```
INSERT INTO Tabella (Attributo1, Attributo2, ..., AttributoN)  
VALUES (Valore1, Valore2, ..., ValoreN);
```

**Nota 29:** "Tabella" è la tabella target, "Attributo1..." sono gli attributi da impostare e sono opzionali come già spiegato, "Valore1..." sono i valori da assegnare agli attributi

## Inserimento con valori ricavati

```
INSERT INTO Tabella [(Attributo1, Attributo2,...,AttributoN)]  
Query_di_selezione
```

Tabella 9.1: OCCHIO perché la query di selezione deve avere una proiezione coerente con lo schema della tabella target

### 9.6.2 Aggiornamento

Permette di modificare il valore di uno o più attributi di uno o più record con valori statici oppure ricavati (l'insieme di record si seleziona mediante una condizione).

Tabella 9.2: Sintassi UPDATE

```
UPDATE Tabella  
SET Attributo1 = Valore1, ..., AttributoN = ValoreN)  
WHERE Condizione
```

Tabella 9.3: "Tabella" è la tabella target, "Attributo1 = Valore1..." sono i nuovi valori, "condizione" è la condizione che può essere anche molto espansiva (subquery, join, ...)

Modificare in "mutuata" tutte le visite del mese corrente, effettuate da pazienti nati prima del 1925

```
UPDATE Visita  
SET Mutuata = 1  
WHERE MONTH(Data) = MONTH(CURRENT_DATE)  
    AND YEAR(Data) = YEAR(CURRENT_DATE)  
    AND Paziente IN (SELECT CodFiscale  
                      FROM Paziente  
                     WHERE YEAR(DataNascita) < 1925);
```

esprimibile con subquery

Figura 9.5: esempio di aggiornamento. (per le condizioni stessa sintassi delle query di selezione)

### 9.6.3 Cancellazione

Permette di cancellare uno o più record dipendentemente dalla veridicità di una condizione, anche articolata.

Tabella 9.4: Sintassi DELETE

**DELETE FROM** Tabella  
**WHERE** Condizione

Il dottor Ettore Grigi ha lasciato la clinica. Rimuovere tutte le sue visite dal database, supponendo che il medico non avesse omonimi nella clinica.

```
DELETE FROM Visita
WHERE Medico = (SELECT Matricola
                  FROM Medico
                  WHERE Nome = 'Ettore' AND
                        Cognome = 'Grigi');
```

Figura 9.6: esempio delete

**Nota 30 :** IMPORTANTE: *Nel DELETE (e anche nell'UPDATE) non si può mettere la tabella target nel FROM della query contenuta nella clausola WHERE)*

2 metodi risolutivi

Rimuovere dal database tutti i medici di Pisa che non hanno effettuato visite mutuate il mese scorso.

```
DELETE FROM Medico
WHERE Matricola IN (
    SELECT * FROM (
        SELECT M1.Matricola
        FROM Medico M1
        LEFT OUTER JOIN
            SELECT V2.Medico AS MedicoMutuato
            FROM Visita V2 INNER JOIN Medico M2
            ON V2.Medico = M2.Matricola
            WHERE M2.Citta = 'Pisa' AND V2.Mutuata IS TRUE
                AND YEAR(V2.Data) = YEAR(CURRENT_DATE)
                AND MONTH(V2.Data) = MONTH(CURRENT_DATE)-1
        ) AS D
        ON M1.Matricola = D.MedicoMutuato
        WHERE D.MedicoMutuato IS NULL
    ) AS D2 );
```

Occhio non vede, cuore non duole...

Rimuovere dal database tutti i medici di Pisa che non hanno effettuato visite mutuate il mese scorso.

```
DELETE M1.*  
FROM Medico M1  
LEFT OUTER JOIN (  
    SELECT V2.Medico AS MedicoMutuato  
    FROM Visita V2 INNER JOIN Medico M2 ON V2.Medico = M2.Matricola  
    WHERE M2.Citta = 'Pisa' AND V2.Mutuata IS TRUE  
        AND YEAR(V2.Data) = YEAR(CURRENT_DATE)  
        AND MONTH(V2.Data) = MONTH(CURRENT_DATE)-1  
) AS D  
ON M1.Matricola = D.MedicoMutuato  
WHERE D.MedicoMutuato IS NULL;
```

# Capitolo 10

## Lezione 6

### 10.1 Stored procedure

**Definizione 168 Stored procedure:** sono programmi dichiarativo-procedurali memorizzati nel DBMS. Esse invocate tramite chiamata e possono ricevere e/o restituire valori.

**Nota 31:** sono supportate dalla versione 5.0 di MySQL

□

Se nell'accesso diretto ai dati (quello visto fino a ora) i dati devono essere accessibili e si deve padroneggiare l'uso di SQL, nell'accesso mediante stored procedure, che avviene tramite "call", i dati e il codice sono mascherati (quindi sicurezza e protezione da attacchi).

Architettura multi-tier

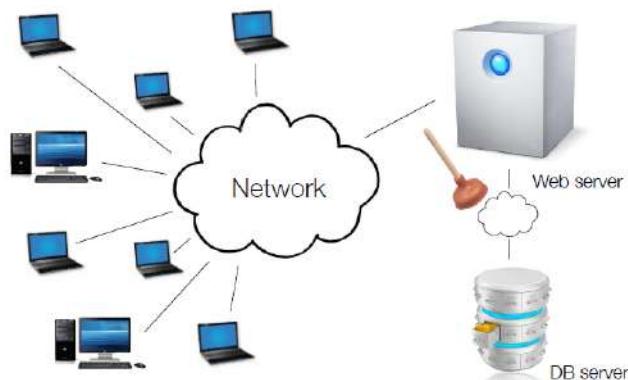


Figura 10.1: grazie alle S.P. avviene uno stasamento tra web server e DB server

### 10.1.1 Prestazioni

Mediane le stored procedure le applicazioni inviano solo una chiamata e non il codice della query. In questo modo il carico è spostato sul server (DBMS) e il traffico della rete è drasticamente ridotto (e l'utente non deve scrivere codice SQL complesso).

### 10.1.2 Sicurezza

Le applicazioni possono essere autorizzate a eseguire stored procedure, ma avere accesso vietato alle tabelle.

### 10.1.3 Riuso del codice

Una stored procedure è come un servizio che gli utenti delle applicazioni (i quali devono avere i permessi per invocare le procedure) che usano il database possono utilizzare senza scrivere codice.

Scrivere una stored procedure che stampi le specializzazioni mediche offerte dalla clinica

```
DROP PROCEDURE IF EXISTS mostra_specializzazioni;
DELIMITER $$

CREATE PROCEDURE mostra_specializzazioni()
BEGIN
    SELECT DISTINCT Specializzazione
    FROM Medico;
END $$

DELIMITER ;
```

Un select statement in una stored procedure equivale a "stampare" su standard output

Figura 10.2: esempio: prima riga: in sql prima di creare qualcosa si cancellano sempre eventuali omonimi preesistenti.

Seconda riga: permette di cambiare il delimitatore di fine statement (consente di dire al DBMS "finché non trovi \$\$ non compilare" in quanto di base è il ";")

Tra il begin e l'end troviamo il codice dichiarativo (che non viene passato a differenza del solo nome della procedure).

La procedure può essere quindi compilata e memorizzata (storing).

#### 10.1.4 chiamata

Tabella 10.1: Sintassi CALL

**CALL nome\_stored\_procedure();**

Tabella 10.2: la chiamata esegue la stored procedure e ottiene il risultato restituito dall'esecuzione del body

## 10.2 Variabili

### 10.2.1 Variabili locali

Sono usate all'interno di una stored procedure, per memorizzare informazioni intermedie di ausilio

**Nota 32 :** IMPORTANTE: *devono essere dichiarate tutte insieme all'inizio del body*

Tabella 10.3: Sintassi DECLARE

**DECLARE nome\_variabile tipo(size) DEFAULT valore\_default**

Tabella 10.4: tipizzazione forte: non si possono dichiarare variabili senza specificarne il tipo (int, double, char...)

Size: capacità della variabile (se non settata è il valore default del tipo di dato)

Valore default: senza default value il valore iniziale è NULL

### Assegnamento

È possibile assegnare un valore a una variabile in due modalità: istruzione **SET** oppure **SELECT + INTO**.

**Nota 33:** *i due modi sono alternativi: dipende dai gusti*

Supporre di essere nel body di una stored procedure e creare una variabile contenente il minimo numero di visite mensili da effettuare, impostato a 20.

```
DECLARE min_visite_mensili INT DEFAULT 0;
```

```
:
```

```
SET min_visite_mensili = 20;
```



la sintassi imporrebbe l'uso di `:=`, ma i due punti si  
possono essere omessi se si usa l'istruzione `SET`

Figura 10.3: Assegnamento statico con SET.

nota: la sintassi imporrebbe l'uso di `":=` ma i due punti possono essere omessi se si usa l'istruzione SET

Supporre di essere nel body di una stored procedure e creare una variabile contenente il numero di visite effettuate nel mese in corso

```
DECLARE visite_mese_attuale INT DEFAULT 0;
```

```
:
```

```
SELECT COUNT(*) INTO visite_mese_attuale  
FROM Visita V  
WHERE MONTH(V.Data) = MONTH(CURRENT_DATE)  
    AND YEAR(V.Data) = YEAR(CURRENT_DATE);
```

oppure

```
SELECT COUNT(*)  
FROM Visita V  
WHERE MONTH(V.Data) = MONTH(CURRENT_DATE)  
    AND YEAR(V.Data) = YEAR(CURRENT_DATE)  
INTO visite_mese_attuale;
```

Figura 10.4: assegnamento calcolato con SELECT + INTO

Creare una variabile contenente il numero di visite effettuate nel mese in corso

```
DECLARE visite_mese_attuale INT DEFAULT 0;

SET visite_mese_attuale =
(
    SELECT COUNT(*)
    FROM Visita V
    WHERE MONTH(V.Data) = MONTH(CURRENT_DATE)
        AND YEAR(V.Data) = YEAR(CURRENT_DATE)
);
```

Figura 10.5: Assegnamento calcolato con SET

### 10.2.2 Variabili user-defined

Sono inizializzate dall'utente senza necessità di dichiarazione, e il loro ciclo di vita equivale alla durata della connection a MySQL server. Non necessitando di dichiarazione hanno tipizzazione debole (non si specifica il tipo della variabile, essa può contenere qualsiasi tipo di dato e tipi di dato diversi in istanti diversi). Il loro contenuto è visibile ovunque ma solo dall'utente che le ha inizializzate, sono case sensitive e il loro identificatore deve iniziare con '@'.

**Nota 34:** Una variabile locale o user-defined è sempre scalare: non può contenere result set.

## 10.3 Parametri

Una stored procedure MySQL accetta parametri di tipo **ingresso**, **uscita** e **ingresso-uscita** che permettono di comunicare col chiamante.

### 10.3.1 Ingresso (IN)

Un parametro in ingresso può essere letto (=passaggio per valore), ma non modificato.

**Nota 35:** i parametri sono in ingresso per default (se non specificato diversamente)

Scrivere una stored procedure che stampi la parcella media di una specializzazione specificata come parametro

```
DROP PROCEDURE IF EXISTS parcella_media_spec;

DELIMITER $$

CREATE PROCEDURE parcella_media_spec(IN _specializzazione VARCHAR(100))
BEGIN
    SELECT AVG(M.Parcella)
    FROM Medico M
    WHERE M.Specializzazione = _specializzazione;
END $$

DELIMITER ;

CALL parcella_media_spec('Ortopedia'); ← chiamata
```

### 10.3.2 Uscita

Un parametro di uscita può essere modificato per assumere il valore del risultato della stored procedure

**Nota 36:** nella chiamata si possono usare variabili user-defined (quelle che iniziano con '@')

Scrivere una stored procedure che restituisca il numero di pazienti visitati da medici di una data specializzazione, ricevuta come parametro

```
DROP PROCEDURE IF EXISTS tot_pazienti_visitati_spec;

DELIMITER $$

CREATE PROCEDURE tot_pazienti_visitati_spec(
    IN _specializzazione VARCHAR(100),
    OUT totale_pazienti_ INT)
BEGIN
    SELECT COUNT(DISTINCT V.Paziente) INTO totale_pazienti_
    FROM Visita V
    INNER JOIN
        Medico M ON V.Medico = M.Matricola
    WHERE M.Specializzazione = _specializzazione;
END $$

DELIMITER ;

CALL tot_pazienti_visitati_spec('Neurologia', @quantiPazienti);

SELECT @quantiPazienti;
```

### 10.3.3 Ingresso-Uscita

La stored procedure riceve il parametro e può leggerlo e modificarlo, la modifica è visibile al chiamante (equivalente al passaggio per reference)

## 10.4 Istruzioni condizionali

Le istruzioni condizionali (IF e CASE) permettono di esprimere condizioni, modificando il flusso di esecuzione

### 10.4.1 Istruzione IF

Tabella 10.5: Sintassi IF

```
IF if_condition THEN
    – blocco istruzioni if true
ELSEIF elseif_1_condition THEN
    – blocco istruzioni elseif_1
    .
    .
    .
ELSEIF elseif_N_condition THEN
    – blocco istruzioni elseif_N
ELSE
    – blocco istruzioni else
END IF;
```

Tabella 10.6: da riga 3 a penultima = parte facoltativa. Il punto e virgola finale è obbligatorio

```

1  DROP PROCEDURE IF EXISTS visite_sopra_soglia;
2
3  DELIMITER $$ 
4  CREATE PROCEDURE visite_sopra_soglia(IN _t INT, IN _s VARCHAR(100), OUT passed BOOLEAN)
5  BEGIN
6      DECLARE visite_mese_attuale INT DEFAULT 0;
7      SET visite_mese_attuale = (
8          SELECT COUNT(*)
9              FROM Visita V
10             INNER JOIN
11                 Medico M ON V.Medico = M.Matricola
12            WHERE M.Specializzazione = _s
13                AND MONTH(V.`Data`) = MONTH(CURRENT_DATE)
14                AND YEAR(V.`Data`) = YEAR(CURRENT_DATE)
15      );
16
17      IF visite_mese_attuale > _t THEN
18          SET passed = TRUE;
19      ELSEIF visite_mese_attuale < _t THEN
20          SET passed = FALSE;
21      ELSE
22          SET passed = NULL;
23      END IF;
24  END $$ 
25  DELIMITER ;
26
27  CALL visite_sopra_soglia(10, 'Otorinolaringoiatria', @controllo);

```

Figura 10.6: esempio: Scrivere una stored procedure che riceva come parametro un intero t e una specializzazione s e restituisca in uscita true se il totale delle visite della specializzazione s nel mese in corso è superiore all'intero t, false se è inferiore e NULL se è uguale.

### Istruzione CASE

Tabella 10.7: Sintassi CASE

```

CASE
WHEN condition_1 THEN
    - blocco istruzioni_1
    .
    .
    .
WHEN condition_N THEN
    - blocco istruzioni_N
END CASE;

```

## 10.5 Più variabili di uscita

```
1 DELIMITER $$  
2 CREATE PROCEDURE primaVisitaPaziente(IN paziente VARCHAR(16),  
3                                     OUT dataPrimaVisita DATE,  
4                                     OUT cognomeMedico VARCHAR(100),  
5                                     OUT nomeMedico VARCHAR(100))  
6 BEGIN  
7     SELECT MIN(V.`Data`) INTO dataPrimaVisita  
8     FROM Visita V  
9     WHERE V.Paziente = paziente;  
10    SELECT M.Cognome, M.Nome INTO cognomeMedico, nomeMedico  
11    FROM Visita V INNER JOIN Medico M ON V.Medico = M.Matricola  
12    WHERE V.Paziente = paziente  
13        AND V.`Data` = dataPrimaVisita  
14    LIMIT 1;  
15 END $$  
16 DELIMITER ;
```

Figura 10.7: Scrivere una stored procedure che restituisca la data in cui un paziente, il cui codice fiscale è passato come parametro, è stato visitato per la prima volta, e il nome e cognome del medico che lo ha visitato in tale circostanza. In caso di più medici, per semplicità, selezionarne uno. (Considerando che Data in Visita fa parte della chiave, in uno stesso giorno più medici possono visitare lo stesso paziente)

## 10.6 Istruzioni iterative

Le istruzioni iterative (WHILE, REPEAT E LOOP) permettono di ripetere blocchi di codice, dipendentemente dalla veridicità di una condizione (la condizione può valutare anche funzioni)

### 10.6.1 WHILE

Tabella 10.8: Sintassi WHILE

**WHILE condition DO**  
– blocco istruzioni  
**END WHILE;**

Tabella 10.9: la condizione è controllata prima di ogni iterazione, il body è eseguito finché la condizione è true

```

1  DROP PROCEDURE IF EXISTS stampa_interi;
2
3  DELIMITER $$ 
4  CREATE PROCEDURE stampa_interi(IN i INT)
5  BEGIN
6      DECLARE s VARCHAR(255) DEFAULT '1';
7      DECLARE counter INT DEFAULT 2;
8
9      WHILE counter <= i DO
10         SET s = CONCAT(s,',',counter);
11         SET counter = counter + 1;
12     END WHILE;
13
14     SELECT s;
15  END $$ 
16  DELIMITER ;
17
18  CALL stampa_interi(3);
```

Figura 10.8: esempio: Scrivere una stored procedure che riceve in ingresso un intero *i* e stampa a video i primi *i* interi separati da virgola, in ordine crescente

### 10.6.2 REPEAT

Tabella 10.10: Sintassi REPEAT

**REPEAT**  
 – blocco istruzioni  
**UNTIL** condition  
**END REPEAT;**

Tabella 10.11: la condizione è controllata dopo ogni iterazione, il body è eseguito finché non si verifica la condizione (UNTIL contiene una condizione d'uscita)

### 10.6.3 LOOP

Tabella 10.12: Sintassi LOOP

```
loop_label:LOOP
  – blocco istruzioni e chek di condizioni
END LOOP;
```

Tabella 10.13: le condizioni di uscita sono gestite dal programmatore

## 10.7 Istruzioni di salto

Le istruzioni LEAVE e ITERATE (sono sempre usate in concomitanza con blocchi if o case) permettono di interrompere un ciclo o passare all'iterazione successiva, rispettivamente.

**Nota 37:** *leave è l'equivalente di break mentre iterate di continue*

```

1  DROP PROCEDURE IF EXISTS stampa_dispari;
2  DELIMITER $$ 
3  CREATE PROCEDURE stampa_dispari(IN i INT)
4  ▼BEGIN
5      DECLARE s VARCHAR(255) DEFAULT '';
6      DECLARE counter INT DEFAULT 1;
7
8  ▼  IF i>0 THEN
9      SET s = '1';
10 ▲ END IF;
11
12 ▼ scan: LOOP
13     SET counter = counter + 1;
14
15 ▼  IF counter = i+1 THEN
16      LEAVE scan;
17 ▲ END IF;
18
19 ▼  IF (counter % 2) = 0 THEN
20      ITERATE scan;
21  ELSE
22      SET s = CONCAT(s, ', ', counter);
23 ▲ END IF;
24 ▲ END LOOP;
25
26  SELECT s;
27 ▲END $$ 
28  DELIMITER ;

```

Figura 10.9: Scrivere una stored procedure che riceve in ingresso un intero  $i$  e stampa a video i numeri dispari da 1 a  $i$ , separati da virgola

## 10.8 Cursori

Scorrono i record di un result set (come farebbe un puntatore), solo in avanti, per effettuare delle azioni all'interno di istruzioni iterative.

**Nota 38 :** warning: *usare con giudizio*

Dichiarazione :

Tabella 10.14: Sintassi DECLARE CURSOR

```
DECLARE NomeCursore CURSOR FOR  
SQL query;
```

Tabella 10.15: i cursori si possono dichiarare solo immediatamente dopo la dichiarazione di tutte le variabili

**Apertura fetch e chiusura** Per usare un cursore lo si deve aprire (open), poi è possibile effettuare il prelievo (fetch) riga per riga, dal risultato, assegnando i valori degli attributi a una lista di variabili, infine lo si deve chiudere (close).

Tabella 10.16: Sintassi OPEN

```
OPEN NomeCursore;
```

Tabella 10.17: Sintassi FETCH

```
FETCH NomeCursore INTO ListaVariabili
```

Tabella 10.18: Sintassi CLOSE

```
CLOSE NomeCursore;
```

## 10.9 Handler

Sono gestori di situazioni, utili (tra l'altro) quando si usano i cursori per riconoscere la fine del result set.

**Nota 39:** *possono (e devono) essere definiti subito dopo le definizioni delle variabili e dei cursori*

esprime l'evento "oggetto non trovato", dove  
l'oggetto è un record



**DECLARE CONTINUE HANDLER FOR NOT FOUND**  
**SET finito = 1;**



È il tipico handler usato per gestire il "fine corsa"  
nella scansione di un result set effettuata all'interno di  
un ciclo dove si scorre un cursore

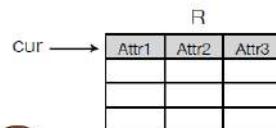
Figura 10.10: esempio: not found handler = handler che gestisce l'evento "not found" (CONTINUE è il tipo dell'handler: significa che non "scoppia il mondo" quando suona la campanella del not found) (NOT FOUND: key word che significa "non ho trovato oggetti dopo quello processato")

### Funzionamento del *not found handler*

**DECLARE finito INTEGER DEFAULT 0;**

finito

**DECLARE cur CURSOR FOR**  
-- query che restituisce il result set R;



**DECLARE CONTINUE HANDLER FOR NOT FOUND**  
**SET finito = 1;**  
/\*



ciclo di fetch dove si scorre il cursore di una posizione  
si preleva un record, e lo si processa. Se, avanzando  
di una posizione, non c'è un nuovo record la campanella  
suona, l'handler la sente, e setta la variabile finito a 1  
\*/

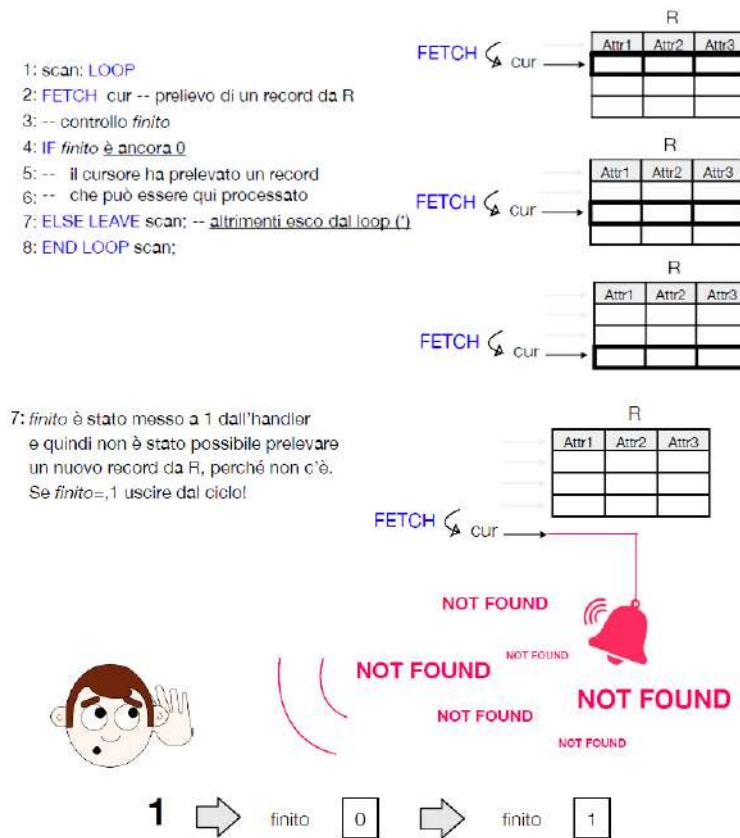


Figura 10.11: funzionamento ciclo fetch

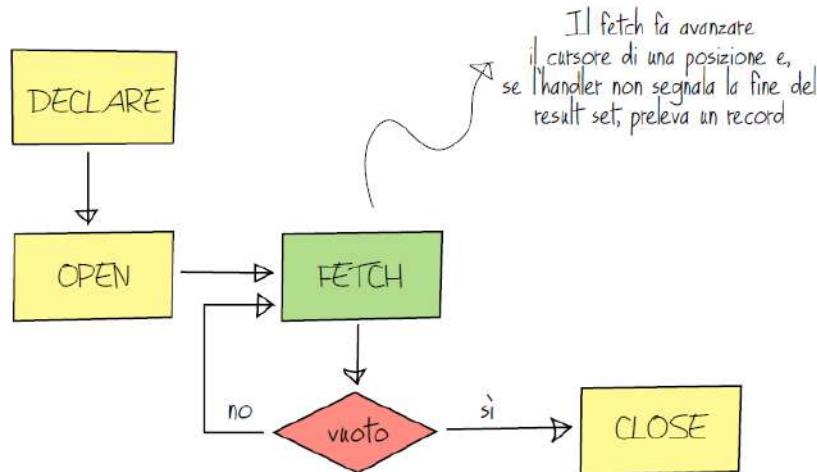


Figura 10.12: funzionamento di un cursore

```

1  DELIMITER $$ 
2  CREATE PROCEDURE PazientiSingoloMedico(IN specializzazione CHAR,
3  									  OUT codiciFiscali VARCHAR(255))
4  BEGIN
5    DECLARE finito INTEGER DEFAULT 0;
6    DECLARE codiceFiscale VARCHAR(255) DEFAULT '';
7
8    -- dichiarazione del cursore
9    DECLARE cursoreCodici CURSOR FOR
10      SELECT V.Paziente
11        FROM Visita V INNER JOIN Medico M ON V.Medico = M.Matricola
12        WHERE M.Specializzazione = specializzazione
13        GROUP BY V.Paziente
14        HAVING COUNT(DISTINCT V.Medico) = 1;
15
16    -- dichiarazione handler
17    DECLARE CONTINUE HANDLER
18      FOR NOT FOUND SET finito = 1;
19
20    OPEN cursoreCodici;
21
22    -- ciclo di fetch per il prelievo
23    preleva: LOOP
24      FETCH cursoreCodici INTO codiceFiscale;
25      IF finito = 1 THEN
26        LEAVE preleva;
27      END IF;
28      SET codiciFiscali = CONCAT(codiceFiscale, ';', codiciFiscali);
29    END LOOP preleva;
30    CLOSE cursoreCodici;
31  END $$ 
32  DELIMITER ;

```

Figura 10.13: stored procedure con cursore: Scrivere una stored procedure che restituisca il codice fiscale dei pazienti visitati da un solo medico per una data specializzazione, organizzati in una stringa formattata del tipo “codFiscale1, codFiscale2, … , codFiscaleN”

```

SET @codiciPazienti = '';
CALL PazientiSingoloMedico('Ortopedia', @codiciPazienti);
SELECT @codiciPazienti;

```

il risultato viene inserito in @codiciPazienti  
come unica stringa formattata

Figura 10.14: chiamata per l'esempio precedente

## 10.10 Stored function

Restituiscono un solo valore e sono richiamabili anche da statement SQL. Sono utili per incapsulare formule o aggregazioni personalizzate.

**Nota 40:** le funzioni di aggregazione sono infatti delle stored function già pronte

Tabella 10.19: Sintassi Stored function

```
CREATE FUNCTION function_name(parametro1, ..., parametroN)
RETURNS datatype DETERMINISTIC | NOT DETERMINISTIC ;
```

Tabella 10.20: crea una funzione function\_name, avente i parametri parametro1,..., parametroN, che restituisce un risultato di tipo datatype, deterministico o no.

**Definizione 169 deterministico:** Una function è deterministica se restituisce un risultato invariante a fronte delle chiamate effettuate con gli stessi valori per i parametri d'ingresso. (per esempio una function che calcola la media è deterministica) □

**Nota 41:** Specificare deterministic o not d. serve al fatto che il dbms si riesce a popolare una sorta di cache che permette di migliorarne le prestazioni

Esempio: Scrivere una function che, preso in ingresso un numero di visite effettuate da un medico, restituisca: ‘low’ se il numero di visite è inferiore a 20; ‘medium’ se il numero di visite è compreso fra 20 e 50; ‘high’ se il numero di visite supera 50.

```
DELIMITER $$  
DROP FUNCTION IF EXISTS rank;  
CREATE FUNCTION rank(totaleVisite INT)  
RETURNS VARCHAR(6) DETERMINISTIC  
BEGIN  
    -- variabile risultato  
    DECLARE ranking VARCHAR(6) DEFAULT "";  
  
    -- assegnamento del ranking su condizione  
    CASE  
        WHEN totaleVisite < 20 THEN  
            SET ranking = 'low';  
        WHEN totaleVisite BETWEEN 20 AND 50 THEN  
            SET ranking = 'medium';  
        WHEN totaleVisite > 50 THEN  
            SET ranking = 'high';  
    END CASE;  
    -- restituzione del risultato  
    RETURN ranking;  
END $$  
DELIMITER ;
```

**Chiamata** Una function può essere chiamata dalle query SQL, dalle stored procedure, dai trigger e dagli event.

**Utilizzo di una function** Esempio: Scrivere una stored procedure che restituisca la posizione in classifica nel mese in corso di un medico, passato come parametro, sfruttando la function *rank()*.

```
DELIMITER $$  
DROP PROCEDURE IF EXISTS classificaMedico;  
CREATE PROCEDURE classificaMedico(IN matricola VARCHAR(5),  
                                    OUT classe VARCHAR(6))  
BEGIN  
    DECLARE visiteMeseCorrente INT DEFAULT 0;  
    DECLARE matricolaValida INT DEFAULT 0;  
    SELECT COUNT(*) INTO matricolaValida  
    FROM Medico M  
    WHERE M.Matricola = matricola;  
    IF matricolaValida = 0 THEN  
        BEGIN  
            SET classe = 'NULL';  
            SIGNAL SQLSTATE '45000'  
            SET MESSAGE_TEXT ='Medico inesistente!';  
        END;  
    ELSE  
        BEGIN  
            SELECT COUNT(*) INTO visiteMeseCorrente  
            FROM Visita V  
            WHERE V.Medico = matricola  
                AND MONTH(V. Data) = MONTH(CURRENT_DATE)  
                AND YEAR(V.'Data') = YEAR(CURRENT_DATE) ;  
            SELECT rank(visiteMeseCorrente) INTO classe;  
        END;  
    END IF;  
END $$  
DELIMITER ;
```

ma se dicesse: Scrivere una stored procedure che produca una classifica di tutti i medici della clinica sfruttando la function *rank()* ?

## 10.11 Temporary table

Sono tabelle temporanee (sono cancellate alla fine della sessione) che mantiene risultati utili all'interno di una sessione.

**Esempio 2:** Scrivere una stored procedure che produca una classifica di tutti i medici della clinica sfruttando la function *rank()* □

```
DELIMITER $$  
DROP PROCEDURE IF EXISTS classificaMedici;
```

```

CREATE PROCEDURE classificaMedici()
BEGIN
    CREATE TEMPORARY TABLE IF NOT EXISTS _Classifica(
        Medico varchar(100) NOT NULL,
        Rank varchar(10) NOT NULL,
        PRIMARY KEY (Medico)
    ) ENGINE=InnoDB DEFAULT CHARSET=latin1;
    TRUNCATE TABLE _Classifica;
    INSERT INTO _Classifica
        SELECT V.Medico, rank(COUNT(*))
        FROM Visita V
        WHERE MONTH(V.Data) = MONTH(CURRENT_DATE)
            AND YEAR(V.'Data') = YEAR(CURRENT_DATE)
        GROUP BY V.Medico;
END $$
```

**DELIMITER ;**

**Nota 42:** il comando "truncate table" permette di svuotare la tabella

**Chiamata .**

```

CALL classificaMedici();
SELECT * FROM _Classifica;
```

# Capitolo 11

## Lezione 7

### 11.1 Database attivi

Sono dotati di una parte reattiva che permette loro di reagire ai cambiamenti (cause scatenanti (istruzioni DML o istanti temporali) mediante comportamenti specifici (particolari operazioni sui dati)

**Evento** Causa (inserimenti, cancellazioni, modifiche o il sopraggiungere di istanti temporali) che scatena l'esecuzione di una azione (operazione) sui dati

**Condizione** Predicato booleano (opzionale) valutato dopo l'evento e, se vero, provoca l'esecuzione dell'azione

**Azione** Blocco di istruzioni eseguite specificate mediante il linguaggio dichiarativo-procedurale MySQL

**Nota 43:** *Quando si verifica l'evento, se la condizione è soddisfatta, viene eseguita l'azione*

### 11.2 Trigger

**Definizione 170 Trigger:** È un insieme di istruzioni eseguite al verificarsi di una causa scatenante (un comando DML). Un trigger è capace di gestire vincoli (detti vincoli di integrità generici o business rule), oppure è usato per aggiornare ridondanze presenti nel database (quando l'utente aggiorna il database la modifica è propagata alle ridondanze coinvolte mediante un trigger). □

#### 11.2.1 Tipi di trigger

Indicano quando "scatta" il trigger (esempio before: prima di fare qualcosa...):

Con BEFORE si indica un'azione di preprocessing, mentre AFTER denota un'azione a posteriori, o comunque "collaterale".

**Nota 44 :** Preprocessing: *operazioni sugli attributi della row che si sta inserendo/modificando, controllo di vincoli...*

**Nota 45 :** a posteriori: *una conseguenza, come ad esempio un'operazione su un'altra tabella*

### 11.2.2 Business rule

Permettono di vietare determinati scenari incoerenti in cui potrebbero trovarsi i dati. Possibili scenari incoerenti sono: vincoli temporali, attributi fuori dominio, vincoli legati alla realtà aziendale in cui i dati si collocano (vera accezione di business rule).

|                                                      |                                                         |
|------------------------------------------------------|---------------------------------------------------------|
| 1.   DROP TRIGGER IF EXISTS blocca_non_mutuate;      | 1. cancella blocca_non_mutuate se esiste                |
| 2.   DELIMITER \$\$                                  | 2.                                                      |
| 3.                                                   | 3.                                                      |
| 4.   CREATE TRIGGER blocca_non_mutuate               | 4. crea il trigger blocca_non_mutuate                   |
| 5.   BEFORE INSERT ON Visita                         | 5. il quale, <b>prima</b> di ogni inserimento in Visita |
| 6.   FOR EACH ROW                                    | 6. per tutte le row (visite) che si inseriscono         |
| 7.   BEGIN                                           | 7. faccia ciò che segue                                 |
| 8.                                                   | 8.                                                      |
| 9.   DECLARE non_mutuate_mese INTEGER DEFAULT 0;     | 9.                                                      |
| 10.                                                  | 10.                                                     |
| 11.   SET non_mutuate_mese =                         | 11. metti in non_mutuate_mese                           |
| 12.    (                                             | 12.                                                     |
| 13.      SELECT COUNT(*)                             | 13. il conto                                            |
| 14.      FROM Visita V                               | 14. delle visite                                        |
| 15.      WHERE V.Medico = NEW.Medico                 | 15. del medico di cui si sta inserendo la visita        |
| 16.      AND MONTH(V.Data) = MONTH(CURRENT_DATE)     | 16. fatte in questo mese,                               |
| 17.      AND YEAR(V.Data) = YEAR(CURRENT_DATE)       | 17. di questo anno,                                     |
| 18.      AND V.Mutuata = 1                           | 18. mutuate                                             |
| 19.    );                                            | 19.                                                     |
| 20.    IF non_mutuate_mese = 10 THEN                 | 20. se le visite sono già 10, allora                    |
| 21.      SIGNAL SOLSTATE '45000'                     | 21. solleva un errore che impedirà l'inserimento        |
| 22.      SET MESSAGE_TEXT = 'Visita non inseribile'; | 22. stampando 'Visita non inseribile'                   |
| 23.    END IF;                                       | 23.                                                     |
| 24.                                                  | 24.                                                     |
| 25. END \$\$                                         | 25.                                                     |
| 26. DELIMITER ;                                      | 26.                                                     |

Figura 11.1: esempio della business rule: Ogni mese, le visite non mutuate di un medico non possono essere più di 10

**Nota 46 :** NEW: *nuova keyword: è il nuovo record che sto provando a inserire, il quale viene bloccato dal trigger e controllato, inoltre nell'esempio estraggo alcune informazioni dal nuovo record per verificare alcune cose (sempre prima di inserirlo)*

**Nota 47:** 45000 è il codice dell'errore generico, inoltre il lancio di questo segnale provoca l'abort dell'inserimento (infatti il nuovo record era ancora "in ostaggio" del trigger)

**Nota 48:** nel where dell'esempio manca: AND V.Mutuata = false"

### 11.2.3 Trigger di tipo before

Tenendo conto della busines rule dell'esempio precedente:

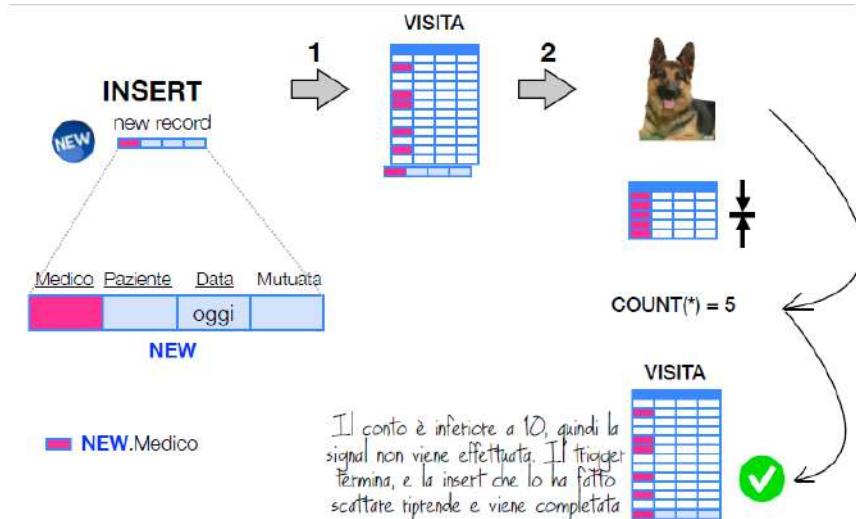


Figura 11.2: successo

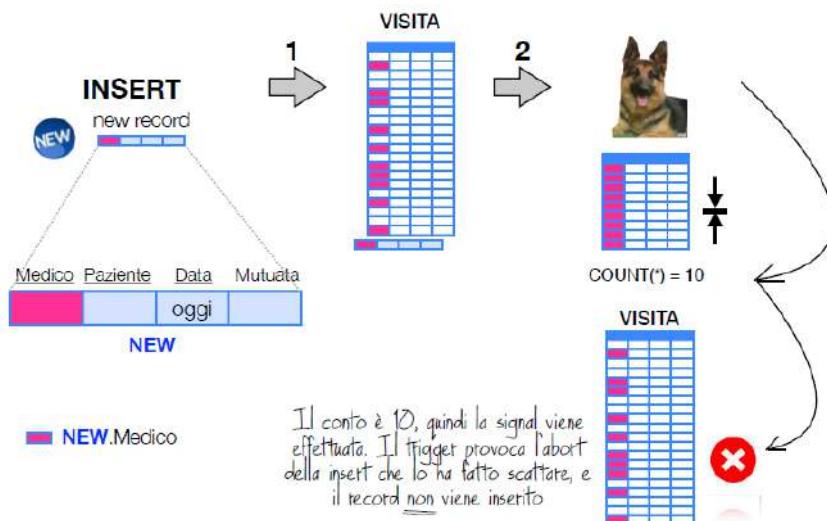


Figura 11.3: Blocco della insert

### Sintassi MySQL per i trigger .

```
DROP TRIGGER IF EXISTS nome_trigger;
CREATE TRIGGER nome_trigger
[BEFORE | AFTER] [INSERT | UPDATE | DELETE] ON TabellaTarget
FOR EACH ROW blocco_istruzioni
```

in italiano:

Crea il trigger nome\_trigger

Prima (dopo) che una o più row siano (siano state) inserite o modificate nella TabellaTarget, o cancellate da essa, per ciascuna, esegui il blocco \_ istruzioni.

**Sintassi MySQL per i trigger multi-statement .**

```
DROP TRIGGER IF EXISTS nome_trigger;
DELIMITER $$ 
CREATE TRIGGER nome_trigger
[BEFORE | AFTER] [INSERT | UPDATE | DELETE] ON TabellaTarget
FOR EACH ROW
BEGIN
    Istruzione1;
    Istruzione2;
    ...
    IstruzioneN;
END $$ 
DELIMITER ;
```

**Nota 49:** Istruzione1 ecc. sono statement dichiarativo-procedurali terminati da ';' .  
L'azione finisce dopo l'END \$\$

#### 11.2.4 Trigger multi-statement

L'azione svolta dal trigger è composta da più blocchi di istruzioni (terminati ognuno da un ';') ed è specificata mediante una sintassi dichiarativo-procedurale

**Nota 50:** MySQL deve capire che dopo il punto e virgola inizia un nuovo blocco di istruzioni del trigger

Esempio di business rule: Ogni mese, le visite non mutuate di un medico non devono superare quelle mutuate.

```
DROP TRIGGER IF EXISTS checkValiditaVisita;
DELIMITER $$ 
CREATE TRIGGER checkValiditaVisita BEFORE INSERT ON Visita
FOR EACH ROW
BEGIN
    DECLARE visite_mutuate_mese INTEGER DEFAULT 0;
    DECLARE visite_non_mutuate_mese INTEGER DEFAULT 0;
    SET visite_mutuate_mese =
        ( SELECT COUNT(*) + IF(NEW.Mutuata = 1, 1, 0)
        FROM Visita V
        WHERE V.Medico = NEW.Medico
        AND V.Mutuata = 1
        AND MONTH('Data') = MONTH(CURRENT_DATE)
        AND YEAR('Data') = YEAR(CURRENT_DATE)
    );
    SET visite_non_mutuate_mese =
```

```

( SELECT COUNT(*) - visite_mutuate_mese + IF(NEW.Mutuata = 0, 1, 0)
  FROM Visita V
 WHERE V.Medico = NEW.Medico
   AND MONTH('Data') = MONTH(CURRENT_DATE)
   AND YEAR('Data') = YEAR(CURRENT_DATE)
 IF visite_non_mutuate_mese >= visite_mutuate_mese THEN
   SIGNAL SQLSTATE '45000'
   SET MESSAGE_TEXT = 'Limite massimo visite mutuate superato';
 END IF;
END $$;
DELIMITER ;

```

### 11.3 Gestione di una ridondanza

Esempio: Gestire un attributo ridondante (lo si può trovare nella tabella visita) nella tabella Paziente contenente la data nella quale un paziente è stato visitato l'ultima volta (Se un paziente viene visitato la data dell'ultima visita cambia!).

**Definizione 171 ridondanza:** Sono attributi il cui valore è ricavabile da altri attributi di tabelle del database.

**Nota 51:** *ridondante significa "in più", cioè se ne può anche fare a meno, non si perde informazione*

□

Consideriamo l'attributo ridondante "ultima visita" da aggiungere alla tabella Paziente.

Esempio: Gestire un attributo ridondante nella tabella Paziente contenente la data nella quale un paziente è stato visitato l'ultima volta.

- evento: inserimento di una nuova row nella tabella Visita
- condizione: non c'è (se un paziente viene visitato due volte lo stesso giorno posso evitare di mettere un if, tanto ci viene scritta comunque la current date)
- azione: aggiornare l'attributo Ultima Visita nella tabella Paziente con la data corrente

```

DROP TRIGGER IF EXISTS aggiorna_ultima_visita;
CREATE TRIGGER aggiorna_ultima_visita
AFTER INSERT ON Visita FOR EACH ROW
  UPDATE Paziente
    SET UltimaVisita = CURRENT_DATE
  WHERE CodFiscale = NEW.Paziente;

```

**Nota 52 :** IMPORTANTE: come nel *DELETE* e *NELL'UPDATE* dove la target table non può essere inserita nel from, se qui modificassi la tabella visita scatterebbe ancora il trigger (nel caso fosse update e non insert on visita)

**Nota 53:** il delimiter serve quando il nostro flusso di istruzioni che ci servono sono multi-statement

**Nota 54 :** importante: per each row significa per ogni row inserita

### 11.3.1 Trigger di tipo after

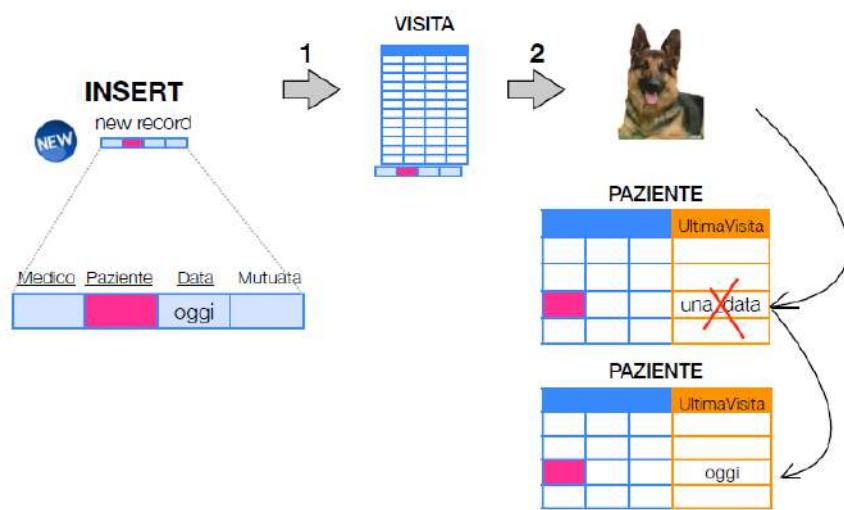


Figura 11.4: il cane vede che è stato inserito un nuovo record in visita e quindi fa l'update della data sul paziente.  
Con i tipo after non c'è il va bene o è andata male.

## Evento, condizione, azione

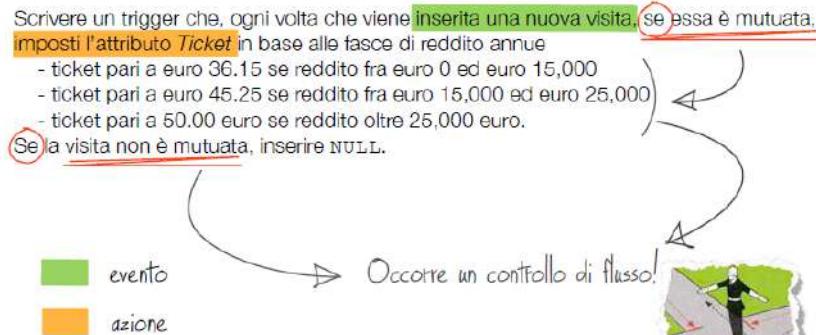


Figura 11.5: esempio

nell'esempio sopra bisogna usare before o after?  
BEFORE perché l'attributo Ticket va settato prima di inserire la visita.

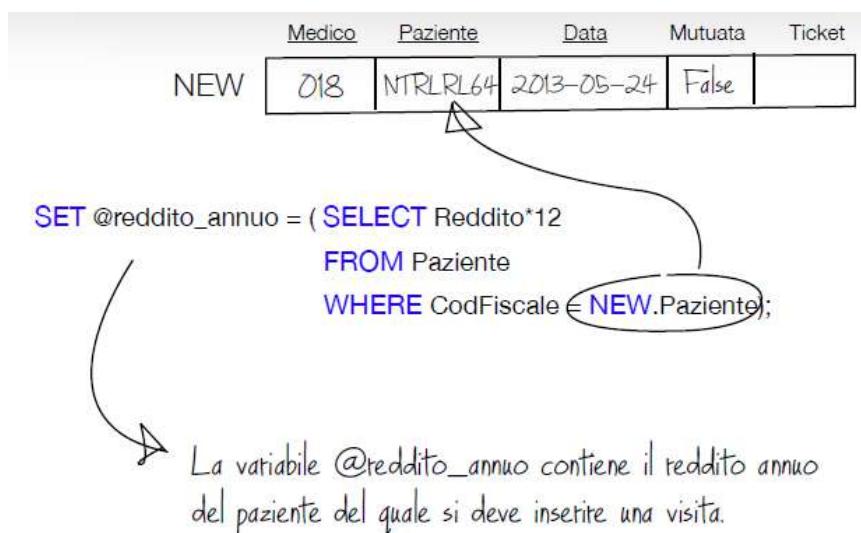


Figura 11.6: Calcolo reddito annuo

Dall'esempio precedente:

```
DELIMITER $$  
CREATE TRIGGER ImpostaTicket  
BEFORE INSERT ON Visita  
FOR EACH ROW  
  
BEGIN
```

```

/* variabile contenente il reddito annuo del paziente */
SET @redditoAnnuo = (SELECT Reddito*12
                      FROM Paziente
                      WHERE CodFiscale = NEW.Paziente
                    );

/* controllo della fascia di reddito e settaggio del ticket */
IF NEW.Mutuata IS TRUE THEN
  IF @reddito_annuo BETWEEN 0 AND 14999 THEN
    SET NEW.Ticket = 36.15;
  ELSEIF @reddito_annuo BETWEEN 15000 AND 25000 THEN
    SET NEW.Ticket = 45.25;
  ELSE
    SET NEW.Ticket = 50.00;
  END IF;
ELSE
  SET NEW.Ticket = NULL;
END IF;

END $$

DELIMITER ;

```

recap di cosa abbiamo visto:

- modalità di esecuzione di operazioni collaterali a seguito di istruzioni dml
  - 1 tipo: vincoli: trigger di tipo before che se il vincolo fallisce provocano il non commit dell'operazione
  - 2 trigger di tipo after
  - 3 particolare di trigger before che non è un vincolo: serve per impostare il val di un record che sto cercando di inserire e agisco direttamente sul record modificando senza distruggerlo

**Nota 55:** *Se i trigger non fossero implementati nel dbms il carico di svolgere tali controlli/operazioni sarebbe sulle applicazioni stesse*

## 11.4 Event

**Definizione 172 Event:** Sono stored programs eseguiti in base a condizioni dettate dal tempo (i quali possono anche essere ricorrenti) ("scattano al sopraggiungere di un istante temporale") □

Esempio: Creare e mantenere giornalmente aggiornata una ridondanza nella tabella Medico contenente, per ciascuno, il totale di visite effettuate

**Nota 56:** "giornalmente" è un evento ricorrente , "totale visite effettuate" è la quantità da aggiornare

TotaleVisite è ridondante perché si può ottenere come:

```
SELECT M.*, COUNT(*) AS TotaleVisite
FROM Visita V
INNER JOIN
Medico M ON V.Medico = M.Matricola
GROUP BY M.Matricola;
```

La ridondanza deve contenere una copia conforme e aggiornata di un'informazione già deducibile dai dati.

**Nota 57:** *occhio perché una ridondanza contiene il dato corretto solo nell'istante di tempo in cui viene aggiornata tramite event, se ad esempio effettuo una visita subito dopo l'orario di aggiornamento del TotaleVisite quest'ultimo conterrà un dato sbagliato fino al suo successivo aggiornamento, per avere quindi una ridondanza sincronizzata si deve utilizzare un trigger.*

Con gli event si ottiene quindi un'informazione che magari è molto difficile calcolare e quindi si accetta il compromesso di non tenerla sincronizzata (essendone comunque consapevoli)

Esempio:

```
CREATE EVENT AggiornaTotaliVisite
ON SCHEDULE EVERY 1 DAY -- periodicita
STARTS '2020-04-24 23:55:00' -- prima esecuzione
DO
    UPDATE Medico
    SET TotaleVisite = TotaleVisite + (SELECT      COUNT(*)
   FROM Visita V
   WHERE V.Medico = Matricola AND
   V.Data = CURRENT_DATE);
```

Sintassi MySQL .

```
ON SCHEDULE EVERY numero DAY | MONTH | YEAR | SECOND | MINUTE | HOUR
```

**Nota 58:** "numero" è la periodicità  $P$  con cui avviene l'esecuzione

**Varianti starts-ends** Un recurring event viene eseguito a intervalli regolari di tempo dall'istante "starts" all'istante "ends"

```
ON SCHEDULE EVERY numero DAY | MONTH | YEAR | SECOND | MINUTE | HOUR
STARTS 'data_ora' ENDS 'data_ora'
```

**Nota 59:** *il momento di starts potrebbe essere diverso dal momento della creazione, senza starts l'event parte dal momento di creazione. Sia start che ends sono opzionali e la indipendenti l'uno dall'altro (a meno che di incongruenze)*

#### 11.4.1 Event a singolo scatto

Un event è eseguito una sola volta, poi o viene mantenuto o viene distrutto  
Sintassi SQL:

```
ON SCHEDULE AT 'data_ora' + INTERVAL numero DAY | MONTH | YEAR |
SECOND | MINUTE | HOUR
[ON COMPLETION PRESERVE]
```

**Nota 60 :** ON COMPLETION PRESERVE: *una volta eseguito l'event viene buttato via il codice di default, si può evitare ciò tramite questa dicitura*

#### 11.4.2 Scheduler degli event

Gli event sono monitorati (gestione delle attese e delle esecuzioni) da un processo (programma di esecuzione sul DBMS) detto event scheduler. Per funzionare quindi gli event hanno bisogno nel DBMS di un processo attivo che li scheduli (=mandi in esecuzione)/"che controlli gli orologi" **Attivazione dello scheduler:**

```
SET GLOBAL event_scheduler = ON;
```

**Nota 61:** *Settando questa variabile di sistema, parte la schedulazione degli event*

Esempio es esame: Implementare una stored procedure che, ricevute in ingresso due date d1 e d2, e un paziente, restituisca il numero di medici da cui p è stato visitato, per la prima volta, nel lasso di tempo fra d1 e d2.

```
-- modo dichiarativo
DROP PROCEDURE IF EXISTS medici_prima_visita;
DELIMITER $$
CREATE PROCEDURE medici_prima_visita(IN _d1 DATE, IN _d2 DATE,
IN _p CHAR(4), OUT n_ INT)
BEGIN
    DECLARE n_medici INT DEFAULT 0;
    SET n_medici=
    (
        SELECT COUNT(DISTINCT V.Medico)
        FROM Visita V
        WHERE V.Data BETWEEN _d1 AND _d2
        AND V.Paziente = _p
        AND NOT EXISTS(
            SELECT *
            FROM Visita Vprec
            WHERE Vprec.Medico = V.Medico
            AND Vprec.Paziente = V.Paziente
            AND Vprec.Data < V.Data
        )
    );

```

```

        SET n_ = n-medici;
END $$

-- -----
-- modo procedurale (e un po' anche dichiarativo)
DROP PROCEDURE IF EXISTS medici_prima_visita;
DELIMITER $$

CREATE PROCEDURE medici_prima_visita(IN _d1 DATE, IN _d2 DATE,
IN _p CHAR(4), OUT n_ INT)
BEGIN
    DECLARE finito INT DEFAULT 0;
    DECLARE medico CHAR(2);
    DECLARE data_visita DATE;
    DECLARE n_medici INT DEFAULT 0;
    DECLARE visite_cur CURSOR FOR
    SELECT V.medico, V.Data
    FROM Visita V
    WHERE V.Data BETWEEN _d1 AND _d2
    AND V.paziente = _p;

    DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET finito = 1;

    OPEN visite_cur;

    scan: LOOP
        FETCH visite_cur INTO medico, data_visita;
        -- nel fetch estraiamo medico e data del select
        -- e le mettiamo in due variabili locali dette
        -- medico e data_visita
        IF finito = 1 THEN
            LEAVE scan;
        END IF;

        IF NOT EXISTS (SELECT *
                       FROM Visita Vprec
                       WHERE Vprec.Data < data_visita
                         AND Vprec.medico = medico
                         -- secondo me qui manca
                         -- la condizione su p
                         )
        THEN
            SET n_medici = n_medici + 1;
        END IF;
    END LOOP scan;
    CLOSE visite_cur;
-- -----
-- modo solo procedurale (non lo scrivo perche dice che in questo caso
-- e' ridicolo farlo)

```

# Capitolo 12

## Lezione 9

La lezione 8 è esercitazione.

### 12.1 Materialized view

Se non si può attendere che una query termini le sue elaborazioni fornendo il risultato nei suoi tempi (che possono essere anche di uno o più giorni), occorre che questi risultati siano memorizzati, già pronti, da qualche parte nel nostro database in modo che non debba attendere l'esecuzione della query ma basti prelevarle da una tabella già pronta detta *materialized view*. Il problema è che quando calcolo un risultato con l'utilizzo di questa materialized view le tabelle sottostanti (medici visite ecc..) continuano a cambiare: con la M.V. ho solo fatto una foto in un certo momento del tempo → se guardo questi (nella M.V.) dati non appena calcolati saranno uguali a quelli che restituisce la query sui dati grezzi (raw data), se guardo questi dati aspettando del tempo può darsi che il risultato della query sia diverso: nel frattempo il db si è evoluto (nuove visite ecc....).

**Definizione 173 Materialized view:** Una materialized view contiene il risultato (pre-calcolato) di una query □

## Facciamo chiarezza

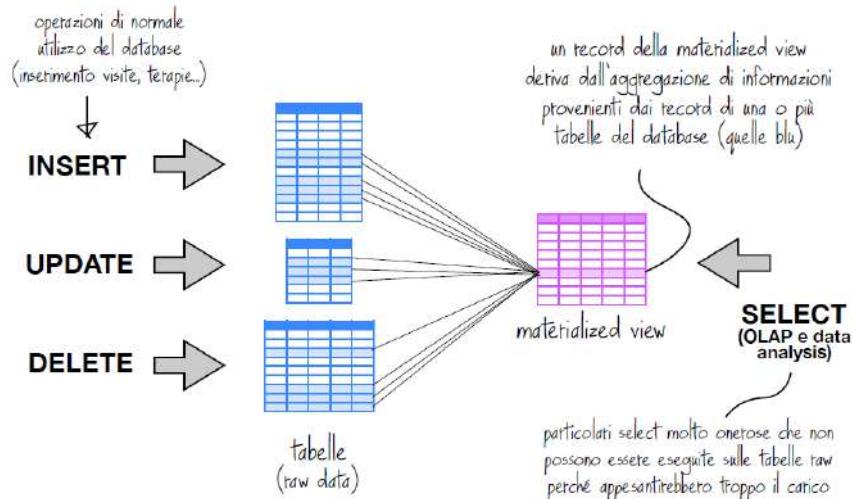


Figura 12.1: se noi facciamo le nostre operazioni olap sui dati blu sono lentissime, meglio farle sui dati rosa

**Nota 62 :** fun fact: *le operazioni si dividono in OLTP (on line transaction processing) e OLAP (on line analytical processing), le prime frequenti e veloci mentre le seconde lente e rare, noi trattiamo le olap mentre le oltp sono tipo le transazioni che possono avvenire da unicredit (transazioni a diritto)*

### 12.1.1 Utilizzi

Si usano se c'è bisogno di una risposta veloce, e la query che restituisce il risultato della materialized view è complessa e lenta nel restituirla.

Si accettano quindi dei compromessi (avere dati non aggiornatissimi) ma che però sono già una buona indicazione su di essi.

Ricapitolando: Pro:

- risultati di query complesse già pronti
- il risultato senza ricalcolarlo
- indici ad hoc per velocizzare l'accesso

contro:

- possibilità ddi informazioni non aggiornate
- maggiore occupazione di memoria
- maggiore carico sul server

### 12.1.2 Refresh

È la politica che permette l'aggiornamento dei dati nella m.v.

Si possono seguire varie strategie per mantenere i dati aggiornati (ad esempio fare il refresh la notte quando il server ha meno carico) fino ad arrivare al costante aggiornamento dei dati tali che siano sincronizzati con "quelli blu", a questo punto non c'è differenza tra i raw data e la m.v.

**Definizione 174 Refresh:** Una materialized view può essere aggiornata secondo due modalità:

- full refresh: aggiorna la materialized view "from scratch" (da zero)
- incremental refresh: aggiornamento della parte non più aggiornata (totale o parziale)

□

#### Politiche di refresh

- **immediate:** tiene la m.v. costantemente sincronizzata (in sync) con i raw data → si fa tramite trigger
- **deferred:** si tengono le modifiche da applicare alla m.v. per poi applicarle in un secondo momento in modo da "non intaccare il normale fluire della vita quotidiana" (esempio bancomat) → si fa tramite event
- **on demand:** se si vuole il risultato "ora" ci vuole una stored procedure che aggiorni su richiesta la m.v. e in quel momento effettui l'aggiornamento → si fa tramite stored procedure

nell'immediate il tipo di refresh non è né incremental né full (essendo sincronizzato), le altre due politiche possono essere sia full refreshed che incremental.

Esempio:

```
-- creazione della materialized View
CREATE TABLE MATERIALIZED_VIEW(
    Paziente CHAR(100) NOT NULL,
    NumVisite INT(11) NOT NULL DEFAULT 0,
    UltimaVisita DATE,
    PRIMARY KEY (Paziente)
) ENGINE = InnoDB DEFAULT CHARSET=latin1;
```

**Nota 63:** la parte di engine... si può omettere perché sono valori di default del nostro dbms

**Immediate refresh (sync):** avviene tramite uno o più trigger:

```

-- immediate refresh (sync)
DELIMITER $$

DROP TRIGGER IF EXISTS immediate_refresh_mv1 $$

CREATE TRIGGER immediate_refresh_mv1
AFTER INSERT ON Visita
FOR EACH ROW
BEGIN
    UPDATE MATERIALIZED_VIEW
    SET NumVisite = NumVisite + 1,
        UltimaVisita = CURRENT_DATE
    WHERE Paziente = NEW.Paziente;
END $$ -- finisco il primo statement quindi uso il delimitatore

DROP TRIGGER IF EXISTS immediate_refresh_mv2 $$

CREATE TRIGGER immediate_refresh_mv2
AFTER INSERT ON Paziente
FOR EACH ROW
BEGIN
    INSERT INTO MATERIALIZED_VIEW(Paziente)
    VALUES (NEW.CodFiscale);
END $$
```

**On demand refresh (full):**

```

-- On demand refresh (full)
DELIMITER $$

DROP TRIGGER IF EXISTS on_demand_refresh_mv $$

CREATE TRIGGER on_demand_refresh_mv
BEGIN
    TRUNCATE MATERIALIZED_VIEW;
    -- truncate = elimina tutto il contenuto
    INSERT INTO MATERIALIZED_VIEW
    SELECT P.CodFiscale,
        IF(V.Paziente IS NULL, 0, COUNT(*)),
        -- se non ha fatto join 0 visite altrimenti count *
        IF(V.Paziente IS NULL, NULL, MAX(V.Data))
        -- se non ha fatto join NULL altrimenti ultima data
    FROM Visita V
    RIGHT OUTER JOIN
        Paziente P ON V.Paziente = P.CodFiscale
    GROUP BY P.CodFiscale;
END $$
```

**Deferred refresh (full):** avviene di solito nei momenti in cui non c'è carico computazionale

```

-- Deferred refresh (full)
DELIMITER $$

DROP EVENT IF EXISTS deferred_refresh_mv $$

CREATE EVENT deferred_refresh_mv
ON SCHEDULE EVERY 1 WEEK
```

```
DO
BEGIN
    CALL on_demand_refresh_mv();
END $$
```

DELIMITER ;

esempio slide

## Esempio

---

Creare una materialized view MV\_RESOCOMTO avente funzione di reporting, contenente, per ogni specializzazione medica della clinica, il numero di visite effettuate, il numero di pazienti visitati, e l'incasso, relativo al mese in corso, realizzato grazie alle visite non mutuate.

Implementare l'on demand refresh, l'immediate refresh e il deferred refresh.

© Francesco Pistoletti — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Creazione della materialized view

---

```
1  DROP TABLE IF EXISTS MV_RESOCOMTO;
2  CREATE TABLE MV_RESOCOMTO (
3      Specializzazione varchar(100) NOT NULL,
4      Visite INT DEFAULT NULL,
5      Pazienti INT DEFAULT NULL,
6      IncassoMese DOUBLE DEFAULT NULL,
7      PRIMARY KEY (`Specializzazione`)
8  );
```

in MySQL una materialized view è  
fisicamente una tabella

© Francesco Pistoletti — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Inizializzazione (build)

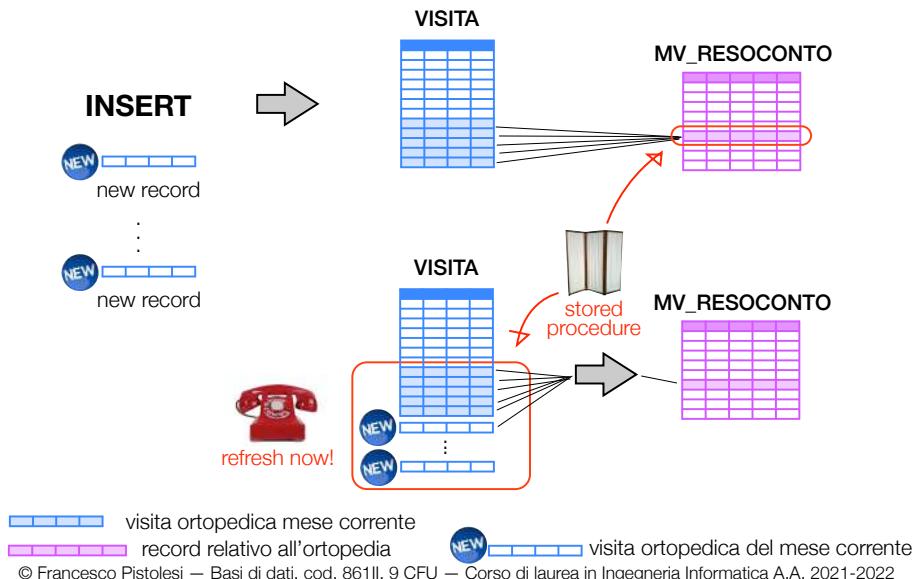
---

```
9   WITH incasso_mese AS
10  (
11      SELECT M.Specializzazione,
12          SUM(M.Parcella) AS Incasso
13     FROM Visita V
14        INNER JOIN
15          Medico M ON V.Medico = M.Matricola
16    WHERE MONTH(V.Data) = MONTH(CURRENT_DATE)
17      AND YEAR(V.Data) = YEAR(CURRENT_DATE)
18      AND V.Mutuata = 0
19    GROUP BY M.Specializzazione
20  )
21  INSERT INTO MV_RESOCOMTO
22  SELECT M.Specializzazione,
23      COUNT(*) AS Visite,
24      COUNT(DISTINCT V.Paziente) AS Pazienti,
25      IM.Incasso AS IncassoMeseCorrente
26  FROM Visita V
27    INNER JOIN
28      Medico M ON V.Medico = M.Matricola
29      NATURAL JOIN
30      incasso_mese IM
31  GROUP BY M.Specializzazione, IM.Incasso;
```

© Francesco Pistoletti — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

buco spiegazione slide da 530-540 mostrato di seguito

## On demand refresh



© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## On demand refresh

```
1 DROP PROCEDURE IF EXISTS full_refresh;
2 DELIMITER $$
3 CREATE PROCEDURE full_refresh()
4 BEGIN
5
6    -- flushing della materialized view
7    TRUNCATE MV_RESOCOMTO;
8
9    -- CTE di supporto
10   WITH incasso_mese AS
11   (
12       SELECT M.Specializzazione,
13             SUM(M.Parcella) AS Incasso
14      FROM Visita V
15         INNER JOIN
16           Medico M ON V.Medico = M.Matricola
17     WHERE MONTH(V.Data) = MONTH(CURRENT_DATE)
18       AND YEAR(V.Data) = YEAR(CURRENT_DATE)
19       AND V.Mutuata = 0
20     GROUP BY M.Specializzazione
21   )
22
23   -- insert statement per il build
24   INSERT INTO MV_RESOCOMTO
25     SELECT M.Specializzazione,
26           COUNT(*) AS Visite,
27           COUNT(DISTINCT V.Paziente)
28             IM.Incasso AS IncassoMeseCorrente
29   FROM Visita V
30     INNER JOIN
31       Medico M ON V.Medico = M.Matricola
32     NATURAL JOIN
33       incasso_mese IM
34   GROUP BY M.Specializzazione, IM.Incasso;
35
36 END $$
37 DELIMITER ;
```

Tramite on demand refresh, di tipo full,  
la materialized view viene svuotata  
e riempita con i dati aggiornati

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Perché si fa il flushing?

tipicamente

È più efficiente gettare il contenuto della materialized view  
e ricalcolarlo **from scratch**



per sfruttare ciò che già è presente nella materialized view si dovrebbero esprimere condizioni articolate sulle tabelle su cui essa è basata, come condizioni temporali, condizioni sui valori degli attributi dei record ecc. Queste condizioni renderebbero più lento l'aggiornamento. Ecco perché il flushing conviene

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

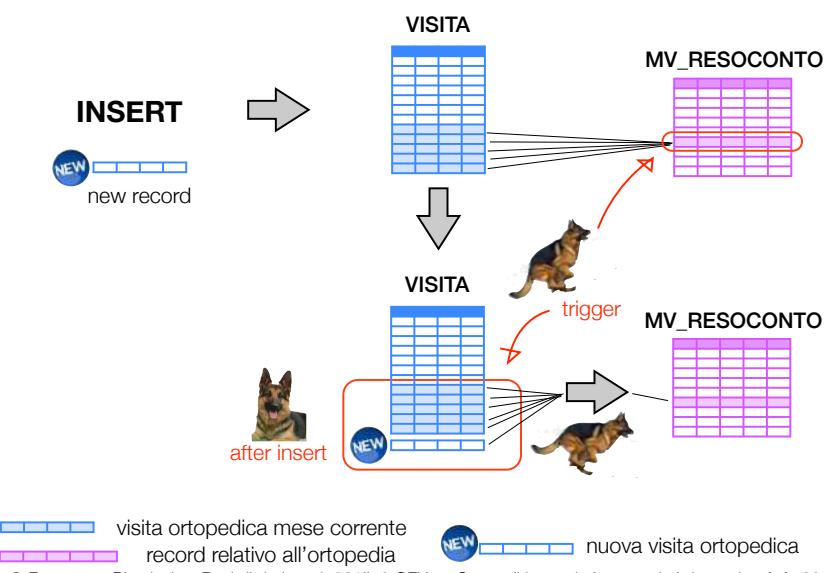
## Immediate refresh

**Aggiorna immediatamente** la materialized view a seguito della modifica delle tabelle (raw data) su cui si basa la materialized view



© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Immediate refresh: funzionamento



© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Immediate refresh (1 di 4)

```
1 DROP TRIGGER IF EXISTS immediate_refresh;
2 DELIMITER $$
3 CREATE TRIGGER immediate_refresh
4 AFTER INSERT ON Visita
5 FOR EACH ROW
6 BEGIN
7
8     DECLARE specializzazione VARCHAR(100) DEFAULT '';
9     DECLARE parcella INTEGER DEFAULT 0;
10    DECLARE visite_prec INTEGER DEFAULT 0;
11    DECLARE mai_visitato INTEGER DEFAULT 0;
12
13    — recupero specializzazione e parcella
14    — del medico di cui è appena stata
15    — inserita la visita (NEW)
16
17    SELECT M.Specializzazione, M.Parcella
18    FROM Medico M
19    WHERE M.Matricola = NEW.Medico
20    INTO specializzazione, parcella
21
22    ;
23    — controllo se il medico ha già
24    — visitato il paziente prima
25    SET visite_prec =
26    (
27        SELECT COUNT(*)
28        FROM Visita V
29        WHERE V.Paziente = NEW.Paziente
30        AND V.Medico = NEW.Medico
31        AND V.Data < CURRENT_DATE
32    );
33
34    IF visite_prec = 0 THEN
35        SET mai_visitato = 1;
36    END IF;
37
38    UPDATE MV_RESOCOMTO
39    SET Visite = Visite + 1,
40        Pazienti = Pazienti + mai_visitato,
41        Incasso = Incasso + parcella
42    WHERE Specializzazione = specializzazione;
43
44 END $$
45 DELIMITER;
```

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Deferred refresh

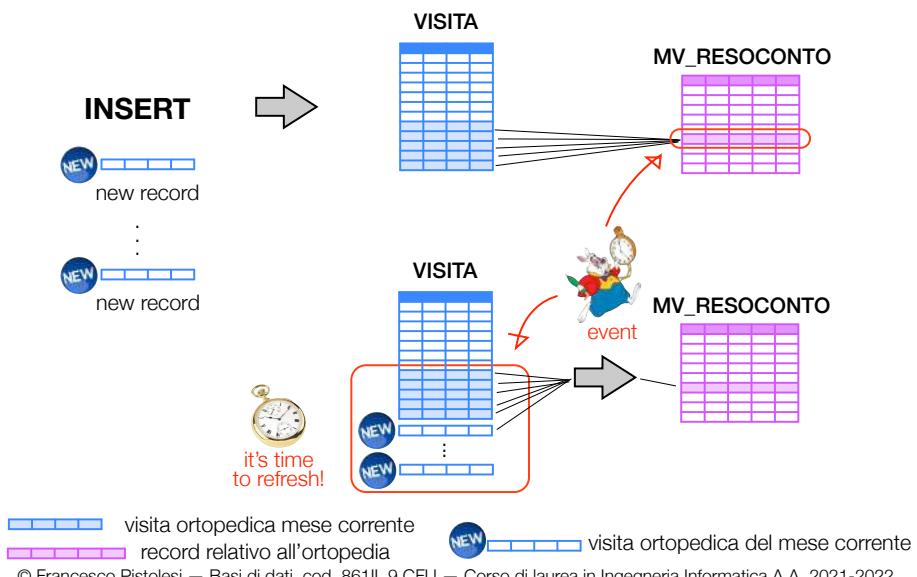
Rinnova il contenuto della materialized view **rispettando una periodicità**

un event aggiorna il contenuto della materialized view in specifici momenti in cui, per esempio, il carico applicativo è basso



© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Deferred refresh: funzionamento



## Deferred refresh

```
1  DROP EVENT IF EXISTS deferred_refresh;
2  DELIMITER $$ 
3  CREATE EVENT deferred_refresh
4  ON SCHEDULE EVERY 1 MONTH
5  STARTS '2021-05-31 23:00:00'
6  DO
7    BEGIN
8      CALL full_refresh();
9    END $$ 
10   DELIMITER $$
```

L'event a partire dal 31 Maggio 2021, chiama la stored procedure `full_refresh()` ogni ultimo giorno del mese. La procedura svuota la materialized view e la riempie con i dati aggiornati

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Incremental refresh

Se invece mantengo le modifiche che ho fatto al database dall'ultimo aggiornamento della m.v. ad ora in un log per aggiornare la m.v. non importa fare tutto da capo: nella m.v. quindi aggiorno solo utilizzando il log senza toccare i raw data.

**Definizione 175 Incremental refresh:** Funzionalità che permette di avere i dati di una materialized view aggiornati fino a un certo istante di tempo (del passato). (fra un refresh e il successivo, le modifiche alle tabelle raw sono mantenute in un log) Si divide in due sottocategorie:

- **complete:** se viene processato tutto il log e quindi al termine la mv torna sincronizzata con i raw data (trasferisco tutti i cambiamenti verificatisi dall'ultimo aggiornamento a ora, richiede quindi più tempo del partial)
- **partial:** trasferire il materiale del log solo in parte nella mv: attendo meno tempo del tempo di processazione dell'intero log processandone solamente un parte: porto avanti la mv in termini di aggiornamento=al termine del refresh la data di ultimo aggiornamento avanza (ma non diventa contemporanea al momento del refresh) e nel log ci rimane roba ancora da processare

Si parla di **rebuild** invece quando si fa il full refresh=ricompilare da zero la mv □

Tramite l'incremental refresh è possibile aggiornare la materialized view sfruttando solo **i dati che essa già contiene** e la **log table**

**Definizione 176 log table:** la tabella di log dove vengono salvati i cambiamenti □

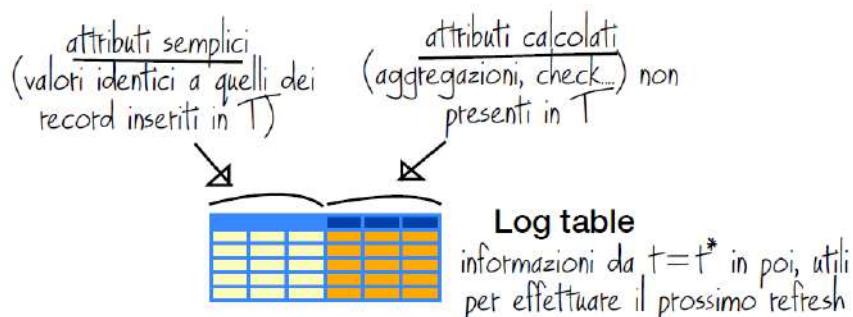


Figura 12.2: la log table deve contenere TUTTI gli attributi semplici e calcolati in modo tale che per aggiornare la mv NON devo accedere ai raw data

Per tutte le operazioni dml (insert, update, delete) si deve verificare una operazione di *push*:

**Definizione 177 push:** propagazione nel nog di una operazione □

**inserimento** l'inserimento nei raw data deve essere propagato nella tabella di log

**cancellazione** la cancellazione nei raw data deve essere propagato nella tabella di log

**aggiornamento** l'update nei raw data deve essere propagato nella tabella di log

**Nota 64:** è bene tenere una log table diversa per ciascuna operazione (*insert, update, delete*)

piccolo reminder: La log table deve contenere le informazioni per far sì che il refresh della materialized view possa essere eseguito usando solo la log table e i dati presenti nella materialized view. L'accesso ai raw data va evitato. Se lo si fa, deve essere solo per operazioni efficienti e veloci, come ad esempio un join sulla chiave primaria.

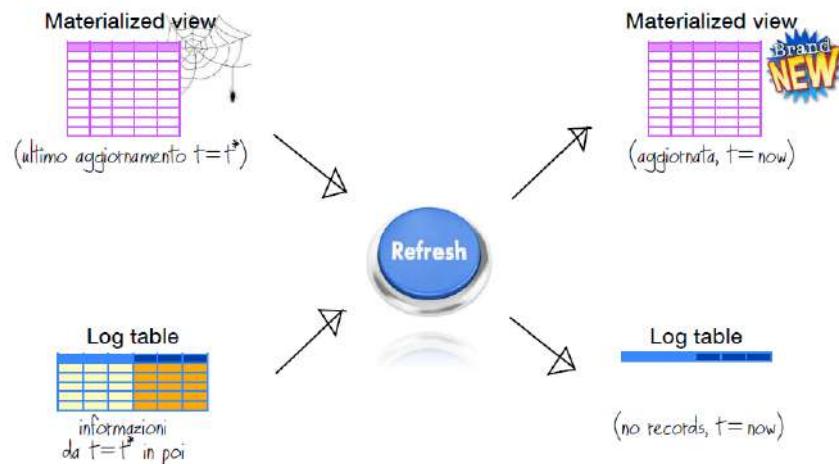


Figura 12.3: esempio complete refresh

| MATERIALIZED VIEW |           |              |
|-------------------|-----------|--------------|
| Paziente          | NumVisite | UltimaVisita |
| aaa               | 2         | 29/4         |
| bbb               | 1         | 2/5          |
| ccc               | 2         | 2/5          |

| LOG_TABLE |            |
|-----------|------------|
| Paziente  | DataVisita |
| aaa       | 3/5        |
| bbb       | 4/5        |
| ddd       | 4/5        |
| aaa       | 8/5        |

esempio (figura sopra) di materialized view e log table dato l'esempio precedente

```
-- creazione log table
CREATE TABLE LOG_TABLE(
    Paziente CHAR(100) NOT NULL ,
    DataVisita DATE
);

DELIMITER $$

DROP TRIGGER IF EXISTS push_1 $$ 
CREATE TRIGGER push_1
AFTER INSERT ON Visita
FOR EACH ROW
BEGIN

    INSERT INTO LOG_TABLE
    VALUES(NEW.Paziente, CURRENT_DATE);

END $$

DROP TRIGGER IF EXISTS push_2 $$ 
CREATE TRIGGER push_2
AFTER INSERT ON Paziente
FOR EACH ROW
BEGIN

    INSERT INTO LOG_TABLE
    VALUES(NEW.CodFiscale, NULL);

END $$
```

Figura 12.4: Trigger di push per la log table dello stesso esempio

**Nota 65:** con la log table posso a tenere aggiornata la mv anche nel caso venga aggiunto un nuovo paziente (guarda secondo trigger)

Partial refresh della mv precedente tramite il log appena descritto

## Partial refresh

```

DROP PROCEDURE IF EXISTS on_demand_refresh_mv $$;
CREATE PROCEDURE on_demand_refresh_mv(_up_to DATE)
BEGIN
    -- aggregazione del log
    WITH aggregated_log AS (
        SELECT LT.Paziente,
               -- con SUM(IF), anziché COUNT(*), scarto le insert di nuovi pazienti
               SUM(IF(LT.DataVisita IS NULL, 0, 1)) AS NuoveVisite,
               MAX(LT.DataVisita) AS NuovaUltima
        FROM LOG_TABLE LT
        WHERE LT.DataVisita <= _up_to OR LT.DataVisita IS NULL
        GROUP BY LT.Paziente
    )

```

LOG\_TABLE

| Paziente | DataVisita |
|----------|------------|
| aaa      | 3/5        |
| bbb      | 4/5        |
| ddd      | 4/5        |
| aaa      | 6/5        |
| eee      | NULL       |



CTE aggregated\_log

| Paziente | NuoveVisite | NuovaUltima |
|----------|-------------|-------------|
| aaa      | 2           | 6/5         |
| bbb      | 1           | 4/5         |
| ddd      | 1           | 4/5         |
| eee      | 0           | NULL        |

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Cosa si vuole fare a questo punto

MATERIALIZED VIEW (MV)

| Paziente | NumVisite | UltimaVisita |
|----------|-----------|--------------|
| aaa      | 2         | 29/4         |
| bbb      | 1         | 2/5          |
| ccc      | 2         | 2/5          |



CTE aggregated\_log (AL)

| Paziente | NuoveVisite | NuovaUltima |
|----------|-------------|-------------|
| aaa      | 2           | 6/5         |
| bbb      | 1           | 4/5         |
| ddd      | 1           | 4/5         |
| eee      | 0           | NULL        |

Result set generato dal join esterno

| MV.Paziente | MV.NumVisite | MV.UltimaVisita | AL.Paziente | AL.NuoveVisite | AL.NuovaUltima |
|-------------|--------------|-----------------|-------------|----------------|----------------|
| aaa         | 2            | 29/4            | aaa         | 2              | 6/5            |
| bbb         | 1            | 2/5             | bbb         | 1              | 4/5            |
| NULL        | NULL         | NULL            | ddd         | 1              | 4/5            |
| NULL        | NULL         | NULL            | eee         | 0              | NULL           |

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Partial refresh

"Replace into" esegue un inserimento  
se non collide sulla chiave primaria, altrimenti  
esegue un update

```
REPLACE INTO MATERIALIZED_VIEW
-- segue immediatamente la CTE aggregated_log
SELECT MV.Paziente,
       IF(MV.Paziente IS NULL,
          AL.NuoveVisite,
          MV.NumVisite + IF(AL.NuovaUltima IS NULL, 0, AL.NuoveVisite)
        ),
       IF(MV.Paziente IS NULL,
          IFNULL(AL.NuovaUltima, NULL),
          AL.NuovaUltima
        )
  FROM MATERIALIZED_VIEW MV
    RIGHT OUTER JOIN
      aggregated_log AL USING(Paziente);
-- se un record di MV fa join, vuol dire che nel log
-- c'è un aggiornamento che coinvolge quel record;
-- se un record di MV non fa join, vuol dire che
-- in MV quel record (paziente) ancora non c'era

-- flushing della parte di log processata
DELETE FROM LOG_TABLE LT
 WHERE LT.DataVisita <= _up_to OR (LT.Paziente IS NULL
                                     AND LT.Paziente IN (SELECT Paziente
                                     FROM MATERIALIZED_VIEW));
END $$
```

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

Figura 12.5: nell'ultimo if LT.DataVisita IS NULL...\*

558-575 esercizio svolto passo passo

# Capitolo 13

## lezione 10

**Definizione 178 Data analyticts:** Funzioni/funzionalita (già presenti) che rendono molto semplice andare a eseguire alcuni compiti che richiedono un'analisi dei dati che sarebbe possibile solo tramite la scrittura di molto codice □

Queste funzionalità permettono di cercare dei pattern nascosti nei dati

**Definizione 179 Pattern:** sono delle caratteristiche che si possono trovare nei dati grazie a delle elaborazioni abbastanza ... □

Il concetto di funzionalità analytics è il concetto di funzionalità finestra (windows functions)

**Definizione 180 Windows function:** Affiancano a ogni record r un valore ottenuto da un'operazione (conteggio, media, rank...) eseguita su un insieme di record logicamente connessi a r □

**Nota 66:** *l'operazione non per forza deve aggregare i record connessi logicamente*

Esistono quindi 2 tipi di analytics:

- aggregate (sum, avg, ...)
- non-aggregate (lead/lag, rank, first\_value, ...)

## Esempi di query con window functions



© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

**Esempio 3 aggregazioni spinte:** Considerate le visite del 2016, indicare il numero di visite effettuate da ogni medico e il totale complessivo di tali visite

```

1 SELECT V.Medico,
2      COUNT(*)
3 FROM Visita V
4 WHERE YEAR(V.Data) = 2016
5 GROUP BY V.Medico WITH ROLLUP;
  
```

| Medico | COUNT(*) |
|--------|----------|
| 001    | 4        |
| 002    | 4        |
| 004    | 5        |
| 013    | 5        |
| 014    | 3        |
| [Null] | 21       |

Figura 13.1: Modificatore WITH ROLLUP

**Nota 67:** *with rollup aggiunge al result set il super-aggregate record (quello indicato dalla freccia) che corrisponde alla somma dei record precedenti.*

*NB: quando si usa non si può usare ORDER BY.*

**Esempio 4 windows functions es 1:** Scrivere una query che indichi, per ogni cardiologo, la matricola, la parcella, e la parcella media della sua specializzazione

**Nota 68:** *ogni record del result set è un record della tabella Medico, tutti i record relativi ai cardiologi devono essere mantenuti, tuttavia, occorre anche applicare un operatore di aggregazione*

## Primo step

Scrivere una query che indichi, per ogni cardiologo, la matricola, la parcella, e la parcella media della sua specializzazione

```
1 SELECT AVG(M.Parcella) ← fa squash ed elimina i record originali!
2 FROM Medico M
3 WHERE M.Specializzazione = 'Cardiologia';
```

| Matricola | Cognome        | Nome        | Specializzazione | Parcella | Città |
|-----------|----------------|-------------|------------------|----------|-------|
| 014       | Indachi        | Loredana    | Cardiologia      | 191      | Pisa  |
| 015       | Ciani          | Gualtiero   | Cardiologia      | 244      | Pisa  |
| 016       | Verdolini      | Maria Paola | Cardiologia      | 233      | Pisa  |
| 017       | Amaranti       | Adevane     | Cardiologia      | 275      | Pisa  |
| 018       | Terra di Siena | Bruciata    | Cardiologia      | 275      | Siena |



Avg(M.Parcella)  
243.6000



per calcolare la parcella media, si perdono i record dei medici

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

Figura 13.2: intraprendendo questa strada risolutiva dovremmo ricorrere all'uso di cte, il che diventa lungo

Introduciamo quindi:

**Definizione 181 Clausola OVER:** permette di definire quelle righe che sono imparentate a ogni riga r di prima (discorso del logicamente connessi). Nel nostro caso

Applica una funzione di tipo aggregate o non-aggregate a un insieme di record, detto partition/partizione, associati a un record di un result set

**Nota 69:** *partition != group*

Le funzioni di tipo aggregate le abbiamo già viste (sum, avg, max...) mentre le non-aggregate usano la partition ottenendo un valore scalare che però non è il risultato di un operatore di aggregazione dei record che la compongono: non fanno lo squash e quindi tutti i record della tabella rimangono disponibili (a meno di eventuali where)

## Soluzione

Scrivere una query che indichi, per ogni cardiologo, la matricola, la parcella, e la parcella media della sua specializzazione

```
1 SELECT
2     M.Matricola,
3     M.Parcella,
4     AVG(M.Parcella) OVER()
5 FROM
6     Medico M
7 WHERE
8     M.Specializzazione = 'Cardiologia';
```



| Matricola | Parcella | AVG(M.Parcella) |
|-----------|----------|-----------------|
| 014       | 180      | 230.0000        |
| 015       | 230      | 230.0000        |
| 016       | 220      | 230.0000        |
| 017       | 260      | 230.0000        |
| 018       | 260      | 230.0000        |

**OVER()** se over non ha contenuto, la partition è sempre la stessa per ogni record, e coincide con il result set della query senza la parte arancio

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

Figura 13.3: il terzo attributo del risultato è la media delle parcelle calcolato su tutta la tabella, nota: non si fa squash :)

**Definizione 182 Partition:** È l'insieme di record ai quali si applicano le funzioni di aggregazione e non, e cambia dipendentemente dal record processato (NB: Può essere definita anche su attributi non proiettati) □

**Esempio 5 con def. della partition:** Scrivere una query che indichi, per ogni medico, la matricola, la specializzazione, la parcella, e la parcella media della sua specializzazione.

(sottolineatura = collegamento logico di ogni riga con il suo "contorno" che definisce la partition) □

## OVER con PARTITION BY

Scrivere una query che indichi, per ogni medico, la matricola, la specializzazione, la parcella, e la parcella media della sua specializzazione

```
1 SELECT
2     M.Matricola,
3     M.Specializzazione,
4     M.Parcella,
5     AVG(M.Parcella) OVER(
6         PARTITION BY
7             M.Specializzazione
8     )
9 FROM
10    Medico M;
```

avg() è applicato partizionando i record del result set della query per specializzazione; ad ogni record del result set si associa poi il valore avg() relativo alla corrispondente specializzazione medica

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

Figura 13.4: partition by non fa squash

## Aggregate functions utilizzabili con OVER

| Name             | Description                                      |
|------------------|--------------------------------------------------|
| AVG()            | Return the average value of the argument         |
| BIT_AND()        | Return bitwise AND                               |
| BIT_OR()         | Return bitwise OR                                |
| BIT_XOR()        | Return bitwise XOR                               |
| COUNT()          | Return a count of the number of rows returned    |
| COUNT(DISTINCT)  | Return the count of a number of different values |
| GROUP_CONCAT()   | Return a concatenated string                     |
| JSON_ARRAYAGG()  | Return result set as a single JSON array         |
| JSON_OBJECTAGG() | Return result set as a single JSON object        |
| MAX()            | Return the maximum value                         |
| MIN()            | Return the minimum value                         |
| STD()            | Return the population standard deviation         |
| STDDEV()         | Return the population standard deviation         |
| STDDEV_POP()     | Return the population standard deviation         |
| STDDEV_SAMP()    | Return the sample standard deviation             |
| SUM()            | Return the sum                                   |
| VAR_POP()        | Return the population standard variance          |
| VAR_SAMP()       | Return the sample variance                       |
| VARIANCE()       | Return the population standard variance          |

→ già viste

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Non-aggregate functions utilizzabili con OVER

| Name           | Description                                                     |
|----------------|-----------------------------------------------------------------|
| CUME_DIST()    | Cumulative distribution value                                   |
| DENSE_RANK()   | Rank of current row within its partition, without gaps          |
| FIRST_VALUE()  | Value of argument from first row of window frame                |
| LAG()          | Value of argument from row lagging current row within partition |
| LAST_VALUE()   | Value of argument from last row of window frame                 |
| LEAD()         | Value of argument from row leading current row within partition |
| NTH_VALUE()    | Value of argument from N-th row of window frame                 |
| NTILE()        | Bucket number of current row within its partition.              |
| PERCENT_RANK() | Percentage rank value                                           |
| RANK()         | Rank of current row within its partition, with gaps             |
| ROW_NUMBER()   | Number of current row within its partition                      |

■ usano l'**intera partition**

■ lavorano **su frame**, cioè sottoinsiemi della partition (vedere slide 53 e seguenti).

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

**Definizione 183 windows functions non-aggregate:** Associano a ciascun record di un result set un valore ottenuto dalla partition senza fonderne i record in un'informazione riepilogativa



## Funzione ROW\_NUMBER

Assegnare un **numero** a ogni medico nella sua specializzazione

```

1 SELECT
2   M.Matricola,
3   M.Cognome,
4   M.Specializzazione,
5   ROW_NUMBER() OVER(PARTITION BY
6     M.Specializzazione)
7 FROM
8   Medico M;

```

row\_number() non fa squash,  
non te serve aggregate!



Un istante della processione:  
la current row t (in rosso) e la  
current partition (rosa).

| Matricola | Cognome        | Specializzazione     | ROW_NUMBER() |
|-----------|----------------|----------------------|--------------|
| 014       | Indachi        | Cardiologia          | 1            |
| 015       | Ciani          | Cardiologia          | 2            |
| 016       | Verdolini      | Cardiologia          | 3            |
| 017       | Amaranti       | Cardiologia          | 4            |
| 018       | Terra di Siena | Cardiologia          | 5            |
| 005       | Neri           | Gastroenterologia    | 1            |
| 013       | Blu            | Gastroenterologia    | 2            |
| 001       | Rossi          | Otorinolaringoiatria | 1            |
| 002       | Verdi          | Otorinolaringoiatria | 2            |
| 003       | Gialli         | Otorinolaringoiatria | 3            |
| 004       | Turchesi       | Otorinolaringoiatria | 4            |

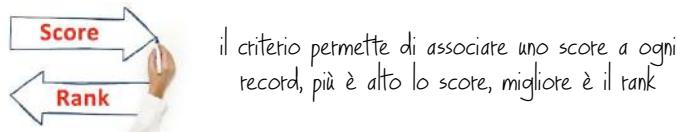
© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Funzione RANK

---

è spesso un attributo su cui si fa un sort

Serve per **stilare una classifica** dipendentemente da **un criterio**



© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

**Esempio 6 di rank:** Effettuare una classifica della convenienza dei medici dipendentemente dalla loro parcella. Restituire matricola, cognome, specializzazione, parola e posizione in classifica. (parcelle basse hanno maggior convenienza, quindi rank migliori) □

## Soluzione

---

Effettuare una classifica della convenienza dei medici dipendentemente dalla loro parcella. Restituire matricola, cognome, specializzazione, parcella e posizione in classifica.

```
1 SELECT
2   M.Matricola,
3   M.Cognome,
4   M.Specializzazione,
5   M.Parcella,
6   RANK() OVER(
7     ORDER BY M.Parcella
8   )
9 FROM
10  Medico M;
```

per ogni row (current row), la partition è ottenuta calcolando il result set della query (righe da 1 a 5 e da 9 a 10), ordinandolo in modo crescente su Parcella, e selezionando infine le row dall'inizio fino alla current row

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Risultato

| Matricola | Cognome        | Specializzazione     | Parcella | RANK() |
|-----------|----------------|----------------------|----------|--------|
| 012       | Grigi          | Medicina generale    | 50       | 1      |
| 001       | Rossi          | Otorinolaringoiatria | 100      | 2      |
| 019       | Rossi          | Oculistica           | 100      | 2      |
| 013       | Blu            | Gastroenterologia    | 110      | 4      |
| 020       | Acquamarina    | Oculistica           | 110      | 4      |
| 008       | Gialli         | Nefrologia           | 120      | 6      |
| 005       | Neri           | Gastroenterologia    | 130      | 7      |
| 002       | Verdi          | Otorinolaringoiatria | 150      | 8      |
| 011       | Marroni        | Psichiatria          | 150      | 8      |
| 006       | Bianchi        | Ortopedia            | 160      | 10     |
| 009       | Arancioni      | Nefrologia           | 170      | 11     |
| 007       | Rosi           | Ortopedia            | 180      | 12     |
| 014       | Indachi        | Cardiologia          | 180      | 12     |
| 003       | Gialli         | Otorinolaringoiatria | 200      | 14     |
| 010       | Celesti        | Neurologia           | 200      | 14     |
| 016       | Verdolini      | Cardiologia          | 220      | 16     |
| 015       | Ciani          | Cardiologia          | 230      | 17     |
| 004       | Turchesi       | Otorinolaringoiatria | 250      | 18     |
| 017       | Amaranti       | Cardiologia          | 260      | 19     |
| 018       | Terra di Siena | Cardiologia          | 260      | 19     |

LOOK  
AT THIS

la posizione 3 non esiste,  
come si vede, in caso di pari  
merito, la funzione rank() salta  
un numero di posizioni della  
classifica pari al numero di ex  
aequo trovati

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

Figura 13.5: Si parla quindi di *rank con gap*, se non ci fosse il salto si parrebbe di *dense rank*

**Esempio 7 su rank n2:** Effettuare una classifica dei medici di ogni specializzazione dipendentemente dalla loro parcella, partendo dalla più alta. Restituire matricola, cognome, specializzazione, parcella e posizione in classifica.

□

## Soluzione

Effettuare una classifica dei medici **di ogni specializzazione** dipendentemente dalla loro parcella, partendo dalla più alta. Restituire matricola, cognome, specializzazione, parcella e posizione nella classifica.

```
1 SELECT          occorre partizionare in questo caso, la
2   M.Matricola, partition è composta dai soli medici della
3   M.Cognome,    specializzazione della current row
4   M.Specializzazione,
5   RANK() OVERC  ↓
6           PARTITION BY M.Specializzazione
7           ORDER BY M.Parcella DESC
8   )
9 FROM
10 Medico M;
```

ordinamento decrescente dei medici in ogni specializzazione, dipendentemente dalla parcella

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

Figura 13.6: (uso del desc)

## Sintassi alternativa con WINDOW

```
1 SELECT
2   M.Matricola,
3   M.Cognome,
4   M.Specializzazione,
5   RANK() OVERC
6   PARTITION BY M.Specializzazione
7   ORDER BY M.Parcella DESC
8   )
9 FROM
10 Medico M;
```

utile per non ripetere la definizione della partition quando si proiettano più attributi basati su di essa

```
1 SELECT
2   M.Matricola,
3   M.Cognome,
4   M.Specializzazione,
5   RANK() OVER w
6   FROM
7   Medico M
8   WINDOW w AS (PARTITION BY M.Specializzazione
9   ORDER BY M.Parcella DESC);
```



© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

**Definizione 184 DENSE\_RANK:** Classifica i record come la funzione rank, ma in caso di ex aequo, la row che segue i record a pari merito assume rank immediatamente successivo (il dense rank è anche detto “rank senza gap”)

□

### Risultato di DENSE\_RANK su partition



| Matricola | Cognome        | Specializzazione     | DENSE_RANK() |
|-----------|----------------|----------------------|--------------|
| 017       | Amaranti       | Cardiologia          | 1            |
| 018       | Terra di Siena | Cardiologia          | 1            |
| 015       | Ciani          | Cardiologia          | 2            |
| 016       | Verdolini      | Cardiologia          | 3            |
| 014       | Indachi        | Cardiologia          | 4            |
| 005       | Neri           | Gastroenterologia    | 1            |
| 013       | Blu            | Gastroenterologia    | 2            |
| 012       | Grigi          | Medicina generale    | 1            |
| 009       | Arancioni      | Nefrologia           | 1            |
| 008       | Gialli         | Nefrologia           | 2            |
| 010       | Celesti        | Neurologia           | 1            |
| 020       | Acquamarina    | Oculistica           | 1            |
| 019       | Rossi          | Oculistica           | 2            |
| 007       | Rosi           | Ortopedia            | 1            |
| 006       | Bianchi        | Ortopedia            | 2            |
| 004       | Turchesi       | Otorinolaringoiatria | 1            |
| 003       | Gialli         | Otorinolaringoiatria | 2            |
| 002       | Verdi          | Otorinolaringoiatria | 3            |
| 001       | Rossi          | Otorinolaringoiatria | 4            |
| 011       | Marroni        | Psichiatria          | 1            |

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

### Rank multipli

**Esempio 8:** Stilare una classifica dei medici in base al numero di visite effettuate. Restituire cognome, specializzazione, numero di viste effettuate, posizione nella classifica generale, e posizione nella classifica per specializzazione.

□

## Soluzione

```
1 WITH visite AS
2 (
3   SELECT V.Medico,
4         M.Cognome,
5         M.Specializzazione,
6         COUNT(*) AS Visite
7   FROM Visita V
8     INNER JOIN Medico M ON V.Medico = M.Matricola
9   GROUP BY V.Medico
10 )
11   SELECT VV.Cognome,
12         VV.Specializzazione,
13         VV.Visite,
14         RANK() OVER(ORDER BY VV.Visite DESC) AS GlobalRank, 1
15         RANK() OVER(PARTITION BY VV.Specializzazione 2
16                           ORDER BY VV.Visite DESC) AS SpecRank
17   FROM visite VV;
```

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

**Definizione 185 Lead & Lag:** Permettono di andare avanti e indietro nella partition: Ricavano il valore di un attributo di una row posta k posizioni prima (lag) o dopo (lead) la current row □

In particolare:

**Definizione 186 LAG:** Permette di affiancare a ogni record i il valore di un record j posizionato k posizioni prima del record i all'interno della partition associata a i. Fa un balzo indietro nella partition di k rows, dove il valore di k deve essere passato come secondo argomento della funzione, mentre il primo argomento è il nome dell'attributo del record j da restituire in uscita □

**Esempio 9:** Considerare le visite otorinolaringoiatriche dal 2010 al 2019, restituire, per ciascuna, matricola del medico, codice fiscale del paziente, data, e data della visita precedente del paziente con un medico della stessa specializzazione. □

## Soluzione con LAG

```

1  SELECT          significa che k=1, quindi si va indietro di una sola row
2    V.Medico,      e si affianca alla current row (i) la data della visita j
3    V.Paziente,
4    V.`Data`,
5    LAG(V.`Data`, 1) OVER C posita nella posizione (i-k)
6    PARTITION BY V.Paziente
7
8        ORDER BY V.`Data` )
9
10   FROM
11     Visita V
12   INNER JOIN
13     Medico M ON V.Medico = M.Matricola
14 WHERE
15   M.Specializzazione = 'Otorinolaringoiatria'
16   AND YEAR(V.`Data`) BETWEEN 2010 AND 2019;

```

© Francesco Pistoletti — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Risultato finale di LAG

| Medico | Paziente | Data       | LAG(V.`Data`, 1) OVER (PARTITION BY V.Paziente) |
|--------|----------|------------|-------------------------------------------------|
| 003    | aaa1     | 2012-05-23 | [Null]                                          |
| 001    | aaa1     | 2012-12-01 | 2012-05-23                                      |
| 001    | aaa1     | 2013-03-01 | 2012-12-01                                      |
| 004    | bbc4     | 2010-10-03 | [Null]                                          |
| 004    | bbel1    | 2010-05-30 | [Null]                                          |
| 003    | bbe1     | 2011-11-27 | 2010-05-30                                      |
| 001    | ccc2     | 2012-07-27 | [Null]                                          |
| 002    | ddd6     | 2010-10-16 | [Null]                                          |
| 004    | eev9     | 2010-10-16 | [Null]                                          |
| 003    | eev9     | 2011-01-23 | 2010-10-16                                      |
| 002    | eev9     | 2012-02-16 | 2011-01-23                                      |
| 004    | erv4     | 2012-06-07 | [Null]                                          |
| 004    | fqa8     | 2012-09-03 | [Null]                                          |
| 002    | fqa8     | 2013-01-26 | 2012-09-03                                      |
| 001    | fqa8     | 2013-02-01 | 2013-01-26                                      |
| 002    | fqa8     | 2010-11-16 | [Null]                                          |
| 001    | gwp5     | 2010-01-19 | [Null]                                          |
| 004    | hh01     | 2011-01-29 | [Null]                                          |
| 003    | hh01     | 2011-08-31 | 2011-01-29                                      |
| 002    | hh01     | 2012-02-15 | 2011-08-31                                      |

© Francesco Pistoletti — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

**Definizione 187 LEAD:** Permette di affiancare a ogni record i il valore di un record j posizionato k posizioni dopo il record i all'interno della partition associata a i. Fa un balzo in avanti nella partition di k rows, dove il valore di k deve essere passato come secondo argomento della funzione, mentre il primo argomento è il nome dell'attributo del record j da restituire in uscita □

**Esempio 10:** Considerare le visite otorinolaringoiche dal 2010 al 2019, restituire, per ciascuna, matricola del medico, codice fiscale del paziente, data, e data della visita successiva del paziente con un medico della stessa specializzazione □

## Soluzione

---

```
1 SELECT
2     V.Medico,
3     V.Paziente,
4     V.`Data`,
5     LEAD(V.`Data`, 1) OVER (
6         PARTITION BY V.Paziente
7             ORDER BY V.`Data`
8     )
9
10 FROM
11     Visita V
12     INNER JOIN
13         Medico M ON V.Medico = M.Matricola
14 WHERE
15     M.Specializzazione = 'Otorinolaringoiatria'
16     AND YEAR(V.`Data`) BETWEEN 2010 AND 2019;
```

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

**Definizione 188 Frame:** È un sottoinsieme della partition dipendente dalla current row, definito dal programmatore per formulare alcune query in modo rapido (Contiene N row precedenti ed M row successive alla current row, all'interno della partition. N ed M dipendono dal contesto) □

## Soluzione

---

```
1 SELECT
2     V.Medico,
3     V.Paziente,
4     V.`Data`,
5     LEAD(V.`Data`, 1) OVER (
6         PARTITION BY V.Paziente
7             ORDER BY V.`Data`
8     )
9
10 FROM
11     Visita V
12     INNER JOIN
13         Medico M ON V.Medico = M.Matricola
14 WHERE
15     M.Specializzazione = 'Otorinolaringoiatria'
16     AND YEAR(V.`Data`) BETWEEN 2010 AND 2019;
```

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

**Definizione 189 windows function su frame:** Processano un result set e calcolano valori (aggregati e non) sulla base di record “adiacenti” alla current row. (il contorno della current row è definito riducendo la partition) □

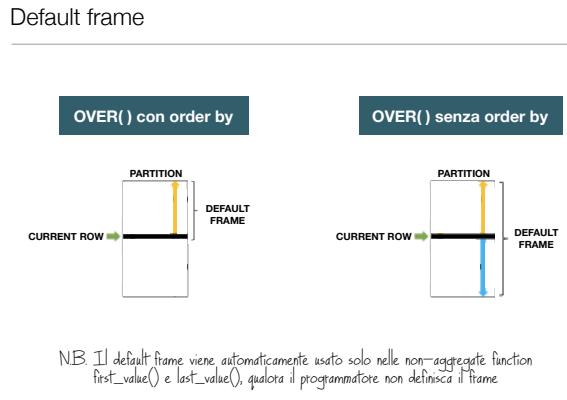
**Definizione 190 Aggregate functions su frame:** se usate su frame le aggregate functions calcolano il risultato usando solo i record del frame, se il programmatore non definisce il frame usano tutta la partition □

**Definizione 191 Non-aggregate functions che lavorano SOLO su frame:** sono:

- FIRST\_VALUE

- LAST\_VALUE

usano i record del frame che se non viene definito dal programmatore (= se N e M non sono definite) è impostato automaticamente a un default frame (il quale cambia dipendentemente dal contenuto di over()) □



© Francesco Pistoletti — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

Figura 13.7: In assenza di specifica di frame, se over() contiene order by, le window functions non aggregate che lavorano su frame considerano i record dall'inizio della partition fino alla current row. Se over() non contiene order by, tali funzioni considerano tutta la partition

**Definizione 192 funzione FIRST\_VALUE: .** □

**Esempio 11:** Date le visite cardiologiche dei pazienti ‘aaa1’, ‘bbc4’ e ‘ccc2’ nel triennio 2012-2014, restituire, per ciascuna, matricola del medico, codice fiscale del paziente, data, e data della prima visita effettuata dal paziente con quel medico □

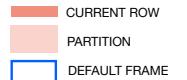
## Soluzione

```

1 SELECT V.Medico,
2     V.Paziente,
3     V.`Data`,
4     FIRST_VALUE(V.`Data`) OVER w
5 FROM Visita V
6 WHERE V.Paziente IN ('aaa1','bbc4','ccc2')
7     AND YEAR(V.`Data`) BETWEEN 2012 AND 2014
8 ->WINDOW w AS (PARTITION BY V.Medico, V.Paziente
9      ORDER BY V.`Data`)

```

Non importa specificare il frame,  
il default frame va bene perché tanto ci  
interessa il primo valore della partition!



| Medico | Paziente | Data       | FIRST_VALUE(Data) | Medico | Paziente | Data       | FIRST_VALUE(Data) | Medico | Paziente | Data       | FIRST_VALUE(Data) |
|--------|----------|------------|-------------------|--------|----------|------------|-------------------|--------|----------|------------|-------------------|
| 001    | aaa1     | 2012-12-01 | 2012-12-01        | 001    | aaa1     | 2012-12-01 | 2012-12-01        | 001    | aaa1     | 2012-12-01 | 2012-12-01        |
| 001    | aaa1     | 2013-01-23 | 2012-12-01        | 001    | aaa1     | 2013-12-11 | 2012-12-01        | 001    | aaa1     | 2013-12-11 | 2012-12-01        |
| 001    | aaa1     | 2013-12-11 | 2012-12-01        | 001    | aaa1     | 2014-01-23 | 2012-12-01        | 001    | aaa1     | 2014-01-23 | 2012-12-01        |
| 001    | ccc2     | 2012-07-27 | 2012-07-27        | 001    | ccc2     | 2012-07-27 | 2012-07-27        | 001    | ccc2     | 2012-07-27 | 2012-07-27        |
| 007    | bbc4     | 2012-09-25 | 2012-09-25        | 007    | bbc4     | 2012-09-25 | 2012-09-25        | 007    | bbc4     | 2012-09-25 | 2012-09-25        |
| 007    | bbc4     | 2014-05-20 | 2012-09-25        | 007    | bbc4     | 2014-05-20 | 2012-09-25        | 007    | bbc4     | 2014-05-20 | 2012-09-25        |
| 007    | ccc2     | 2012-01-04 | 2012-01-04        | 007    | ccc2     | 2012-01-04 | 2012-01-04        | 007    | ccc2     | 2012-01-04 | 2012-01-04        |
| 010    | bbc4     | 2012-01-23 | 2012-01-23        | 010    | bbc4     | 2012-01-23 | 2012-01-23        | 010    | bbc4     | 2012-01-23 | 2012-01-23        |
| 010    | ccc2     | 2012-06-27 | 2012-06-27        | 010    | ccc2     | 2012-06-27 | 2012-06-27        | 010    | ccc2     | 2012-06-27 | 2012-06-27        |
| 011    | bbc4     | 2012-02-23 | 2012-02-23        | 011    | bbc4     | 2012-02-23 | 2012-02-23        | 011    | bbc4     | 2012-02-23 | 2012-02-23        |

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

### Definizione 193 funzione LAST\_VALUE: .



**Esempio 12:** Considerate le visite cardiologiche dei pazienti ‘aaa1’, ‘bbc4’ e ‘ccc2’, nel triennio 2012-2014 e restituirne matricola del medico, codice del paziente, data e data dell’ultima visita del paziente con quel medico nel triennio.



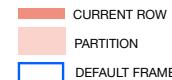
### LAST\_VALUE con definizione del frame

```

1 SELECT V.Medico,
2     V.Paziente,
3     V.`Data`,
4     LAST_VALUE(V.`Data`) OVER w
5 FROM Visita V
6 WHERE V.Paziente IN ('aaa1','bbc4','ccc2')
7     AND YEAR(V.`Data`) BETWEEN 2012 AND 2014
8 ->WINDOW w AS (PARTITION BY V.Medico, V.Paziente
9      ORDER BY V.`Data`
10     ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING)

```

il frame va dalla current row  
fino alla fine della partition



Addesso la data dell’ultima visita di aaa1  
con 001 (cioè 2014-01-23) viene  
affiancata a ogni visita che li coinvolge.  
Tutto torna anche per le altre copie  
paziente-medico :-)

| Medico | Paziente | Data       | (LAST_VALUE(Data)) |
|--------|----------|------------|--------------------|
| 001    | aaa1     | 2012-12-01 | 2014-01-23         |
| 001    | aaa1     | 2013-03-01 | 2014-01-23         |
| 001    | aaa1     | 2013-12-11 | 2014-01-23         |
| 001    | aaa1     | 2014-01-23 | 2014-01-23         |
| 001    | ccc2     | 2012-07-27 | 2012-07-27         |
| 007    | bbc4     | 2012-09-25 | 2014-05-20         |
| 007    | bbc4     | 2014-05-20 | 2014-05-20         |
| 007    | ccc2     | 2012-01-04 | 2012-01-04         |
| 010    | bbc4     | 2012-01-25 | 2012-01-25         |
| 010    | ccc2     | 2012-06-27 | 2012-06-27         |
| 011    | bbc4     | 2012-02-23 | 2012-02-23         |

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

**Esempio 13 moving average (media mobile):** Scrivere una funzione analytics efficiente che, per ogni terapia conclusa del paziente ‘ttw2’, proietti

il farmaco, la durata e la durata media rispetto alla terapia precedente e successiva con lo stesso farmaco □

## Soluzione

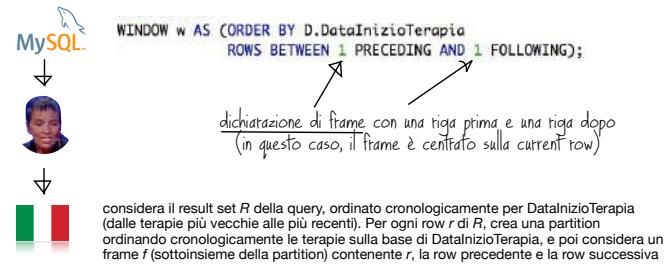
```

1 WITH durate AS
2   (
3     SELECT T.Farmaco,
4            T.DataInizioTerapia,
5            DATEDIFF(
6              T.DataFineTerapia,
7              T.DataInizioTerapia
8            ) AS Durata
9   FROM Terapia T
10  WHERE T.Paziente = 'ttw2'
11    AND T.DataFineTerapia IS NOT NULL
12  )
13 SELECT D.Farmaco,
14       D.Durata,
15       D.DataInizioTerapia,
16       AVG(D.Durata) OVER w
17  FROM durate D
18  WINDOW w AS (ORDER BY D.DataInizioTerapia
19                ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING);

```

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

## Traduzione dal MySQL[LESE] by Olga...



| Paz. | Farmaco | DataInizioTerapia | DataFineTerapia | Materiale |
|------|---------|-------------------|-----------------|-----------|
| 001  | slq6    | 2003-06-03        | 0               |           |
| 002  | slq6    | 2004-04-05        | 0               |           |
| 003  | slq6    | 2005-01-16        | 0               |           |
| 004  | slq6    | 2006-01-16        | 0               |           |
| 005  | slq6    | 2009-12-03        | 0               |           |
| 006  | slq6    | 2012-07-21        | 0               |           |
| 007  | slq6    | 2007-02-23        | 0               |           |
| 008  | slq6    | 2005-02-22        | 0               |           |
| 009  | slq6    | 2008-01-11        | 0               |           |
| 010  | slq6    | 2006-01-24        | 0               |           |
| 011  | slq6    | 2009-07-18        | 0               |           |

© Francesco Pistolesi — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

**Esempio 14:** Considerate le visite ortopediche di ogni paziente, scrivere una query analytics che restituisca codice fiscale del paziente, matricola del medico, la sua parcella, il numero di visite ortopediche effettuate fino a quel momento, e la spesa sostenuta dal paziente per tali visite □

## Soluzione

```
1 SELECT V.Paziente,
2      V.Medico,
3      M.Parcella,
4      COUNT(*) OVER w,
5      SUM(M.Parcella) OVER w
6 FROM
7      Visita V
8      INNER JOIN
9      Medico M ON V.Medico = M.Matricola
10 WHERE
11      M.Specializzazione = 'Ortopedia'
12 WINDOW w AS (
13      PARTITION BY V.Paziente
14      ORDER BY V.`Data`
15      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
16 ); 

Il frame considera le row fino alla current row, andando all'indietro illimitatamente, finché ce ne sono


```

© Francesco Pistoletti — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

**Esempio 15 variante con frame temporale:** Considerate le visite ortopediche di ogni paziente, scrivere una query analytics che restituisca codice fiscale del paziente, matricola del medico, la sua parcella, il numero di visite ortopediche effettuate nei sei mesi precedenti, e la spesa sostenuta dal paziente per tali visite □

## Soluzione con dichiarazione di range

```
1 SELECT V.Paziente,
2      V.Medico,
3      M.Parcella,
4      COUNT(*) OVER w - 1,
5      SUM(M.Parcella) OVER w
6 FROM
7      Visita V
8      INNER JOIN
9      Medico M ON V.Medico = M.Matricola
10 WHERE
11      M.Specializzazione = 'Ortopedia'
12 WINDOW w AS (
13      PARTITION BY V.Paziente
14      ORDER BY V.`Data`  

15      RANGE BETWEEN
16      INTERVAL 6 MONTH PRECEDING
17      AND CURRENT ROW
18 ); 

con la keyword 'range' si possono dichiarare frame basati su intervalli di valori numerici, date o timestamp


```

© Francesco Pistoletti — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

Figura 13.8: il range lo definisce temporalmente di 6 mesi precedenti perché l'ordinamento è fatto su V.Data

Se voglio escludere il mese attuale?

```
1 SELECT V.Paziente,
2      V.Medico,
3      M.Parcella,
4      COUNT(*) OVER w - 1,
5      SUM(M.Parcella) OVER w
6 FROM
7      Visita V
8      INNER JOIN
9      Medico M ON V.Medico = M.Matricola
10 WHERE
11      M.Specializzazione = 'Ortopedia'
12 WINDOW w AS (
13     PARTITION BY V.Paziente
14     ORDER BY V.`Data`
15     RANGE BETWEEN
16     INTERVAL 6 MONTH PRECEDING
17     AND INTERVAL 1 MONTH PRECEDING
18 );
```

© Francesco Pistoletti — Basi di dati, cod. 861II, 9 CFU — Corso di laurea in Ingegneria Informatica A.A. 2021-2022

Figura 13.9: il range lo definisce temporalmente di 6 mesi precedenti perché l'ordinamento è fatto su V.Data

**Nota 70 :** IMPORTANTE: le funzioni che lavorano su frame hanno bisogno della specifica tramite rows o tramite range altrimenti usano un frame di default che dipende dalla presenza o meno di order by (con order by infatti si va a lavorare sul frame invece che sulla partition completa)

**Nota 71:** Se si usano frame dichiarati su range, si limita la partition solo usando lassi temporali, non si possono usare anche limitazioni della partition basate sul numero di rows preceding e following