Basi di dati

Giacomo Fantoni
Telegram: @GiacomoFantoni
Github: https://github.com/giacThePhantom/BasiDati

18 settembre 2020

Indice

Capitolo 1

Introduzione

Un database è una collezione di dati contenente entità e relazioni riguardanti un'applicazione. Un database management system o DBMS è un software utilizzato per assistere nel mantenimento e utilizzo di larghe collezioni di dati.

1.1 Gestire dati

Esistono varie tipologie di DBMS ma ci si concentrerà sui relational database systems o RDBMSs che sono quelli più utilizzati.

1.2 File systems e DBMS

Il salvataggio di dati in un file system porta a problematiche di memoria in quanto la dimensione delle memorie principali è limitata, non è possibile indirizzare tutti i files. Si rende necessario scrivere programmi specifici per estrarre i dati dai files, proteggere i dati da cambi concorrenti che potrebbero creare inconsistenze. Si devono scrivere applicazioni ad-hoc in modo che dati possano essere ripristinati ad uno stato consistente se il sistema fallisce e creare un sistema per implementare politiche di sicurezza come l'user control. Un DBMS viene creato con lo scopo di ovviare a questi problemi.

1.2.1 Vantaggi di un DBMS

- Data indipendence: i programmi applicativi non sono dipendenti da dettagli della rappresentazione e salvataggio dei dati, pertanto il DBMS fornisce un'astrazione che li nasconde.
- Efficient data access: un DBMS utilizza molte tecniche sofisticate per salvare e recuperare dati efficientemente.
- Data integrity and security: se viene fatto l'accesso ai dati attraverso un DBMS questo può forzare constraints di integrità e un access control che stabilisce che dati sono visibili a quali utenti.
- Data administration: facilita il processo di gestione dei dati affidandolo a utenti esperti.

- Concurrent access and crash recovery: un DBMS organizza gli accessi concorrenti ai dati in modo che un utente può considerare i dati come se fosse fatto l'accesso ad essi da un utente alla volta. Inoltre protegge gli utenti dagli effetti del fallimento del sistema.
- Reduced application development time: fornendo importanti funzioni comuni a varie applicazioni unitamente all'interfaccia di alto livello permette la creazione più veloce e robusta di applicazioni.

Esistono ragioni per non utilizzare un DBMS: nel caso di applicazioni specifiche con constraints realtime o con poche ben definite operazioni critiche o nel caso un'applicazione debba gestire dati in modo non supportato dal linguaggio di query.

1.3 Descrivere e salvare dati in un DBMS

Un data model è una collezione di descrizioni di dati di alto livello che nasconde molti dettagli. Un DBMS permette di descrivere i dati da salvare utilizzando tale modello. Il modello più utilizzato è il relational data model. Un'ulteriore astrazione rispetto al data model è il semantic data model ma un DBMS non è costruito introno a quello. Un esempio di semantic data model è l'entity reationship (ER) model.

1.3.1 Livelli di astrazione

La descrizione di un database è creata da uno schema per ognuno dei tre livelli di astrazione: conceptual, physical e external. Un data definition language DLL è utilizzato per definire gli schemi esterni e concettuali come SQL. Le informazioni riguardanti gli schema sono salvati nei system catalogs.

1.3.1.1 Conceptual schema

Il conceptual schema o schema logico descrive i dati salvati nei termini del data model.

1.3.1.2 Physical schema

Il physical schema specifica dettagli di salvataggio, ovvero riassume come il data model utilizzato è fisicamente salvato nella memoria.

1.3.1.3 External schema

Gli external schema, solitamente espressi nel data model permettono di autorizzare e personalizzare gli accessi ai dati, può esistere un'external schema per ogni gruppo di utenti e consiste in una o più viste e relazioni dal conceptual schema. Le viste sono relazioni create a partire da specifiche degli utenti finali.

1.3.2 Data indipendence

Si intende con data indipendence il fatto che i programmi applicativi sono isolati dai cambi nel modo in cui i dati sono strutturati e salvati, è creata dal conceptual e external schemas. Esistono due livelli di data indipendence: logica rispetto ai cambi nelle relazioni e physical rispetto ai cambi della struttura di salvataggio dei dati su disco.

1.4 Query

La facilità con cui le informazioni possono essere ottenute da un database determina il suo valore. Per ricavare informazioni si utilizzano le queries, strutture create attraverso il linguaggio di query. Un esempio di linguaggio di query è il relational calculus basato su logica matematica. Un DBMS permette di creare, modificare e richiedere data attraverso un data manipulation language o DML che insieme al DDL forma il data sublanguage quando unito a un linguaggio host.

1.5 Transaction management

Per ordinare le richieste di vari utenti in modo da evitare inconsistenze da accessi concorrenti e proteggere gli utenti da fallimenti di sistema si devono gestire in maniera accurata le transazioni, ovvero ogni esecuzione di un programma utilizzatore in un DBMS. Transazioni parziali non sono ammesse.

1.5.1 Esecuzione concorrente di transazioni

Un'importante funzione del DBMS è organizzare gli accessi ai dati in modo che ogni utente possa ignorare il fatto che altri possano star accedendo ai dati concorrentemente. Un locking protocol è un insieme di regole che ogni transazione deve seguire per permettere che nonostante diverse transazioni siano sovrapposte il loro risultato netto sia lo stesso che se fossero eseguite serialmente. Un lock è un meccanismo utilizzato per controllare l'accesso agli oggetti del DBMS, ne esistono di due tipi: shared locks su un oggetto possono essere mantenuti da due differenti transazioni allo stesso tempo, mentre un exclusive lock su un oggetto garantisce che nessun'altra transazione mantenga un lock sullo stesso oggetto.

1.5.2 Transazioni incomplete e crash di sistema

Un DBMS deve garantire che cambi svolti da transazioni incomplete siano rimossi. Per farlo mantiene un log di tutte le scritture sul database: ogni azione di scrittura deve essere scritta su disco prima che il cambio corrispondente sia effettuato. Questa proprietà è chiamata Write-ahead log o WAL. Per far questo un DBMS deve poter spostare una pagina da memoria a disco. Il tempo di recupero da crash può essere ridotto forzando delle informazioni su disco periodicamente e creando un checkpoint.

1.6 Struttura di un DBMS

Un DBMS accetta comandi SQL generati da una varietà di interfacce utente, produce piani di query evaluation li esegue e ritorna la risposta. Quando un utente presenta una query entra in gioco un query optimizer che usa le informazioni riguardo a come i dati sono salvati per produrre un piano di esecuzione efficiente. Un piano di esecuzione è un albero di operatori relazionali. Il codice che implementa questi operatori si trova sopra il livello di file e dei metodi di accesso che supporta file che sono collezioni di pagine o collezioni di records. Gli heap files o file di pagine non ordinate e indici sono supportati. Questo livello si trova sopra il buffer manager che porta pagine da disco a memoria principale in necessità di risposte per richieste di lettura. Il livello più basso è il disk space manager. Esistono un transaction manager, un lock manager e un recovery manager.

Capitolo 2

The entity relationship model

Il modello relazioni entità viene utilizzato per descrivere i dati coinvolti in un'operazione del mondo reale attraverso oggetti e le rispettive relazioni ed è utilizzato nella fase di database design concettuale.

2.1 Database design e ER diagram

Il processo di database design può essere diviso in sei fasi, ma l'ER diagram viene utilizzato solo nelle prime tre:

2.1.0.1 Requirement analysis

In questa fase viene analizzato che tipo di dati vanno salvati nel database, che applicazione vada costruita al di sopra di esso e quali dati sono maggiormente soggetti a cambiamenti e pertanto influenti sulle performance. È solitamente un processo informale in cui si discute con gli utenti e in cui viene analizzato l'ambiente, analisi di documentazione di applicazioni da modificare o cambiare.

2.1.0.2 Conceptual database design

Le informazioni raccolte nella fase precedente sono utilizzate per costruire un modello di alto livello dei dati che saranno salvati nel database con i constraints conosciuti per quanto riguarda i dati. In questa fase viene solitamente creato l'ER diagram in modo da crearne una semplice descrizione di come sia sviluppatori che utenti lo rappresentano. Permette sia una facile discussione con gli utenti che una rappresentazione abbastanza precisa da rendere la traduzione in un modello dati comprensibile al database utilizzato.

2.1.0.3 Logical database design

In questa fase viene scelto un DBMS per implementare il conceptual design, pertanto l'ER schema viene tradotto in un relational schema. Il risultato è un logical schema nel relational model.

2.1.0.4 Schema refinement

In questa fase vengono analizzate le relazioni nello schema in modo da trovare eventuali problematiche e raffinarlo. Questa fase è guidata dalla teoria della normalizzazione delle relazioni.

2.1.0.5 Physical database design

In questa fase si considerano tipiche cariche di lavoro per il database e lo si raffina in modo che raggiunga i criteri di performance.

2.1.0.6 Application and security design

Si deve identificare entità e processi coinvolti nell'applicazione, descrivere il loro ruolo nell'applicazione e garantire che i dati siano accessibili solo a chi ne abbia il privilegio.

2.2 Entità, attributi e entity set

Si definisce un'entità un oggetto del mondo reale distinguibile dagli altri. È spesso utile identificare una collezione di simili entità, chiamata entity set. Questi ultimi non possono essere divisi. Un entità viene descritta da una serie di attributi e tutte le entità nello stesso entity set hanno gli stessi attributi. La scelta degli attributi caratterizza il livello di dettaglio con cui si vuole rappresentare un entità. Per ogni attributo si rende necessario identificare un dominio di possibili valori. Inoltre ad ogni entity set va associata una key, un set di attributi minimale che identifica univocamente l'entità. Essendo che ci potrebbe essere più di una candidate key ne si designa una a primary key. Un entity set è rappresentato da un rettangolo mentre gli attributi da un ovale. La primary key è sottolineata.

2.3 Relazioni e relationship set

Una relazione è un'associazione tra due o più entità. Analogamente alle entità più relazioni simili possono essere raccolte in relationship sets, che possono essere pensati come insiemi di ennuple: $\{(e_1, \ldots, e_n) | e_1 \in E_1, \ldots, e_n \in E_n\}$. Ogni ennupla denota una relazione che coinvolge n entità da e_1 a e_n, dove e_i sta nell'entity set E_n. Una relazione può possedere degli attributi descrittivi in modo da registrare informazioni riguardanti la relazione ma non le entità. Si definisce istanza di un relationship set un relationship set, ovvero come lo stato del relationship set in un qualche istante. Gli entity set che partecipano in una relazione non devono essere necessariamente distinti. Una relazione può infatti coinvolgere entità appartenenti allo stesso entity set. In questo caso le due entità svolgono diversi ruoli che deve essere identificato da due role identificator su ogni lato della relazione in modo da costituire un identificatore unico per la relazione.

2.4 Capacità addizionali dell'ER model

2.4.1 Key constraints

I key constraints si riferiscono alla numerazione delle relazioni, che si possono dividere in tre categorie principali:

- Many to many: le entità di entrambi gli entity set si possono relazionare in qualsiasi numero con le entità dell'altro entity set.
- Many to one: le entità di un entity set si possono relazionare con al più un'entità dell'altro, mentre queste ultime in qualsiasi numero con le prime.

- One to one: le entità di entrambi gli entity set si possono relazionare con al più un'entità dell'altro entity set.

2.4.1.1 Key constraints per relazioni ternarie

La convenzione sopra descritta può essere estesa a relazioni tra tre o più entità: se un'entità E possiede un key constraints nella relazione R ogni istanza di E appare al più una volta in ogni istanza della relazione R.

2.4.2 Participation constraints

Indipendenti dai key constraints indicano quando la relazione sia totale o parziale. Nel primo caso ogni istanza dell'entità deve apparire nell'istanza della relazione.

2.4.3 Weak entities

Non è sempre vero che ogni entità possieda una key. In questo caso si determinano delle weak entities, ovvero entità considerate unicamente in relazione con l'entità con cui sono in relazione. Si identificano con una combinazione tra loro attributi e la key della seconda entità. Permettono di salvare dati dell'entità unicamente quando è in relazione con un'altra chamata identifying owner. Per poter essere create si necessita che la owner entity e la weak entity siano in una one to many relationship set (ogni owner con una o più weak entities, ma ogni weak entity ha un singolo owner). Questo relationship set è detto identifiyng relationship set della weak entity. La weak entity deve avere una total partecipation con l'identifiyng relationship set.

2.4.4 Class hierarchy

In alcuni casi risulta naturale classificare le entità di un entity set in sottoclassi. In questo modo una sottoclasse eredita tutti gli attributi e quando richiesta la classe padre vengono restituite anche le classi figli. La class hierarchy può essere considerata in due modi:

- Ogni super classe è specializzata in sottoclassi. La specializzazione è il processo di identificare un sottoinsieme di un entity set (la classe padre) che condivide delle caratteristiche. Tipicamente è creata prima la superclasse e poi le sottoclassi con gli attributi specifici.
- Le sottoclassi sono generalizzate nella superclasse. In questo modus operandi vengono identificate delle caratteristiche comuni di una collezione di classi che vengono poi raccolte in una superclasse.

Si possono specificare due tipologie di constraints per le gerarchie: di overlap determina se due sottoclassi possono contenere la stessa entità, di covering determina se le entità della sottoclasse includono tutte le entità della superclasse. Esistono due motivi principali per identificare sottoclassi:

- Aggiungere attributi descrittivi che hanno senso solo per la sottoclasse.
- Per identificare un entity set che partecipa in una relazione.

2.4.5 Aggregation

In alcuni casi si rende necessario modellare una relazione tra una collezione di entità e relazioni. Per fare questo si usa l'aggregazione che ci permette di indicare che un relationship set partecipa in un altro relationship set in modo che il primo sia considerato come un'identità. L'uso di un aggregazione invece di una relazione ternaria va usato in caso di attributi alle relazioni o a particolari constraints sulla relazione.

2.5 Conceptual design con ER model

2.5.1 Entità o attributo

Durante il processo di design può non essere chiaro se un attributo debba essere considerato come un'entità a sè stante, questo va reso necessario quando:

- Vanno salvati più attributi per entità.
- La struttura dell'attributo è complessa e la si vuole catturare nel database.
- Un attributo rappresenta un set di valori.

2.5.2 Entità o relazione

Una relazione con troppi attributi porta a una perdita di efficienza di spazio, pertanto conviene trasformarla in alcuni casi in un'entità.

2.5.3 Binary or ternary relationship

Una relazione ternaria deve essere trasformata da una serie di relazioni binarie quando un'entità non può essere in relazione con più di un'altra, la stessa entità deve essere sempre in relazione con l'altra, è una weak entity. Il viceversa è necessario quando le due relazioni sono strettamente correlate, ovvero se avviene una deve avvenire anche l'altra e non si può rappresentare un attributo di una delle relazioni con chiarezza.

2.5.4 Relazione ternaria o aggragazione

La scelta tra relazione ternaria o aggregazione è per principalmente determinata dall'esistenza di una relazione con un relationship set, ma anche da certi constraints che si vogliono imporre: come ad esempio un record da salvare della relazione con l'aggregazione o quando una relazione è many to one.

Capitolo 3

Il modello relazionale

Il modello relazionale considera il database come una collezione di una o più relazioni dove ogni relazione è una tabella. Una relazione consiste di uno relation schema e un relation instance: l'istanza è una tabella e lo schema descrive le teste di colonna per le tabelle. Lo schema specifica il nome della relazione, il nome di ogni campo o colonna o attributo e il dominio di ogni campo. Un dominio è riferito dal suo nome e possiede un iniseme di valori associati. Uno schema di una relazione è nella forma $< Tablename > (att_1 : \ dom_1, \ldots, att_n : \ dom_n)$. Un'istanza di una relazione è un insieme di tuple dette records in cui ogni tupla ha lo stesso numero di campi dello schema relazionale. L'istanza può essere pensata come la tabella in cui ogni tupla è una riga e tutte le righe hanno lo stesso numero di campi. Ogni relazione è definita da un insieme di tuple uniche, ovvero tutte diverse. I campi sono convenzionalmente numerati. Uno schema relazionale specifica il dominio di ogni attributo nell'istanza. Questi constraint di dominio specificano che i valori che appaiono nella colonna devono essere presi da quelli di tale dominio. Considerando lo schema superiore $\{< att_1 : \ d_1, \ldots, att_n : \ d_n > \| d_1 \in Dom_1, \ldots, d_n \in Dom_n\}$. Le istanze di relazioni sono instanze che soddisfano i constraints di dominio nel relational schema. Il grado o arity di una relazione è il numero di campi, mentre la sua caridnalità è il numero di tuple in un'istanza. Un database relazionale è un insieme di relazioni di nome diverso. Un relational database schema è una collezione di schemi per le relazioni del database. Si dice istanza di un database relazionale una collezione di istanze di relazioni.

3.1 Creare e modificare relazioni utilizzando SQL

il linguaggio SQL utilizza la parola table per denotare relazioni. Il sottoinsieme di SQL per la creazione, eliminazione e modifica di tabelle è detto DDL. Per creare nuove tabelle si utilizza il comando *CREATE TABLE <nome tabella>(att$_1$ dom$_1$, ..., att$_n$ dom$_n$)*. Le tuple sono inserite attraverso il comando *INSERT INTO <caratterizzazione tabella> VALUES <tupla>* su può omettere la lista dei campi in INTO, ma è buona pratica includerla in modo da chiarire univocamente l'ordine. Si possono eliminare tuple attraverso il comando *DELETE FROM <caratterizzazione tabella> WHERE <caratteristica delle tuple da eliminare>*. Si possono modificare i valori di una tupla esistente attraverso il comando *UPDATE <caratterizzazione tabella> SET <elementi da modificare> WHERE <caratteristica delle tuple da eliminare>*. La clausola WHERE è applicata prima di SET e se il valore della colonna è utilizzato per modificarlo nella parte destra dell'uguaglianza viene utilizzato il valore vecchio.

3.2 Integrity constraints sulle relazioni

In integrity constaint è una condizione specificata su uno schema di database che restringe i dati che possono essere salvati su un'istanza del database. Se tale istanza li soddisfa tutti è detta legale. Un DBMS forza gli integrity constraints nel senso che permette di salvare nel database solo istanze legali. Gli integrity constraints sono specificati e forzati a tempi diversi: sono specificati durante la definizione dello schema, mentre quando un applicazione di database gira il DBMS controlla le violazioni e vieta cambi ai dati che violerebbero gli IC. In alcuni casi potrebbe compensare i cambiamenti ai dati in modo che si evitano violazioni. Un esempio di IC sono i domain constraints.

3.2.1 Key constraints

Un key constraint è una proposizione secondo cui un sottoinsieme minimo di campi di una relazione è un identificatore unico per la relazione. Questo insieme è detto candidate key per la relazione. Una candidate key pertanto determina che due tuple distinte in un'istanza legale non possono avere valori identici in tutti i campi della chiave e nessun sottoinsieme dell'insieme che forma la candidate key è un identificatore unico per la chiave. Un insieme che contenga oltre alla chiave ulteriori campi è detto superkey. Ogni relazione è garantita di avere una chiave e se non diversamente specificato tutti i campi la formano. Una relazione può avere diverse candidate key e di tutte queste si può identificare una primary key e intuituvamente una tupla può essere riferita da ogni parte del database salvando i valori dei campi della sua primary key.

3.2.1.1 Specificare key constraints in SQL

In SQL per dichiarare che un sottoinsieme costituisce una chiave si utilizza l'UNIQUE constraint. Al più una di queste candidate keys può essere dichiarata come primary key attravero PRIMARY KEY constraint. *CREATE TABLE <nome tabella>(att$_1$ dom$_1$, ..., att$_n$ dom$_n$, UNIQUE(<candidate key>), CONSTRAINT <nome constraints> PRIMARY KEY(<attributi>))*. CONSTRAINT permette di nominare un constraint in modo che quando venga violato venga ritornato il nome così specificato.

3.2.2 Foreign key constraints

Può capitare che l'informazione salvata in una relazione sia collegata a informazioni legate in altre relazioni. Se una delle relazioni è modificata, l'altra deve essere controllata e cambiata per mantenere i dati consistenti. Un IC che riguarda entrambe le relazioni deve essere specificata in modo da permettere questi controlli. In questo caso si rende necessario un foreign key constraint. Un foreign key constraints è un campo di una relazione che si riferisce ad un'altra tabella e deve essere identica per numero di campi e tipi ad essa. Una foreign key si può riferire alla stessa relazione. Si introduce il valore **null** di un campo che vuol dire che il valore nel campo è sconosciuto o non applicabile. Sono possibili nelle foreign key ma non nelle primary key.

3.2.2.1 Specificare foreign key constraints in SQL

CREATE TABLE <nome tabella>(att$_1$ dom$_1$, ..., att$_n$ dom$_n$, FOREIGN KEY (<nome campi>) REFERENCES <nome tabella>) che obbliga ai campi della foreign key di apparire uguali nella tabella referenziata.

3.3 Constraints generali

Si può aumentare la granularità dei constraints attraverso table constraints e assertions: le prime sono associate ad una tabella e controllate quando è modificata, mentre le assertions agiscono su più tabelle.

3.4 Forzare integrity constraints

Se i constraints su primary key e UNIQUE sono di facile soluzione, SQL crea diverse soluzioni per gestire la violazione dei foreign key constraints:

- Se è inserita una riga con una foreign key che non esiste nella tabella referenziata l'INSERT viene respinto.
- Se una riga della tabella referenziata è eliminata o aggiornata si può:
  - Eliminare tutte le righe della tabella che la referenzia con lo stesso valore. *CASCADE*.
  - Non permettere l'eliminazione se una riga la referenzia. *NO ACTION*.
  - Settare ogni valore che si riferiva a tale riga ad un valore di default. *SET DEFAULT*, dove il valore di default è specificato nella definizione del campo.
  - Se non è parte della primary key della tabella referenziata si può settare a *null*. *SET NULL*

SQL permette in caso di eliminazione o aggiornamento di scegliere una delle quattro opzioni specificandole come parte della dichiarazione della foreign key attraverso le parole chiave scritte sopra precedute da *ON DELETE* o *ON UPDATE*.

3.4.1 Transazioni e constraints

Di default i constraints sono controllati alla fine di ogni SQL statement che potrebbe generare una violazione. Questo controllo può essere troppo rigido e pertanto si possono separare i constraints in *DEFERRED* o *IMMEDIATE*: il primo tipo viene controllato al tempo di commit.

3.5 Querying dati relazionali

Una relational database query è una domanda riguardante i dati e la risposta è una nuova relazione che li contiene. SQL è il linguaggio di query più utilizzato. Si possono visualizzare i dati attraverso il comando *SELECT <nome campi ∗ per tutti> FROM <nome tabella> WHERE <condizioni su tabella>*. Si possono combinare informazioni presenti su tabelle diverse.

3.6 Trasformazioni da ER a relational

Il modello ER è un modo conveniente per rappresentare un design di database di alto livello che poi viene tradotto nello schema relazionale per poi essere implementato.

3.6.1 Da entity sets a tables

Un entity set è mappato in una relazione in modo che ogni attributo di esso diventi un attributo della tabella.

3.6.2 Da relationship sets senza constraints a tables

Un relationship set è mappato a una relazione. Si consideri un relationship set senza chiave e constraints di partecipazione. Per rappresentarlo si deve essere capaci di identificare ogni entità partecipante e dare valori agli attributi descrittivi, pertanto gli attributi di una relazione contengono:

- Gli attributi primary key di ogni entity set partecipante come foreign key.
- Gli attributi descrittivi dell relationship set.

Gli attributi non descrittivi formano una superkey.

3.6.3 Da relationship sets con key constraints a tables

Se un relationship set coinvolge n entity set e qualche m di essi sono collegati attraverso frecce la chiave di ciascuno di essi costituisce una chiave per la relazione dove il relationship set è mappato. Un ulteriore metodo è rappresentato da includere informazioni della relazione nelle tavole corrispondenti agli entity set con la freccia, in particolare se un relationship set coinvolge n entity set e qualcuno di essi è linkato attraverso una freccia la relazione corrispondente a ognuno di questi set può essere aumentata includendo la foreign key delle altre e gli attributi della relazione. Queste foreign key possono avere null value e pertanto dello spazio può essere sprecato, risparmiando sulla difficoltà delle queries.

3.6.4 Da relationship set con partecipation constraints a tables

Questo viene implementato attraverso le assertions: si deve garantire che una foreign key appaia con non null values.

3.6.5 Da weak entities a tables

Un weak entity set partecipa sempre in una relaizone one-to-many relazione binaria e possiede un key constraint e total partetipation. Vengono pertanto tradotte utilizzando una foreign key della entity da cui si riferiscono. Si deve comunque prendere in considerazione che possiede solo una chiave parziale e che quando un'entità padrona viene eliminata deve essere eliminata anche la weak entity. La primary key è pertanto una combinazione della foreign key e un altro attributo e l'eliminazione deve avvenire *ON DELETE CASCADE*.

3.6.6 Tradurre gerarchie di classi

Esistono due approcci per generare gerarchie di classi:

- Mappare ogni elemento della classe padre nella classe figlio attraverso la foreign key della superclasse che svolge il ruolo di chiave primaria e per referenziare tutti gli attributi del padre. Un eliminazione del padre deve essere fatto a cascata sul figlio.
- Creare una relazione che possiede i propri attributi e tutti gli attributi della classe padre.

Il primo approccio è generalmente applicabile. Il secondo unicamente se la superclasse non contiene elementi che non sono presenti nelle classi figlie in quanto non esiste.

3.6.7 Tradurre aggregazioni

Quando si vuole fare una relazione con un'aggregazione si crea una relazione con la chiave dell'altro elemento, la chiave della relazione dell'aggregazione e gli attributi descrittivi. Se la relazione centrale dell'aggregazione non possiede attributi può essere ottenuta attraverso la coppia delle chiavi degli entity set nell'aggregazione come foreign key.

3.7 Viste

Si dice vista una tabella le cui righe non sono salvate esplicitamente nel database ma sono computate quando rese necessarie da una view definition. Si creano come *CREATE VIEW* con un select. Possono essere utilizzate come una tabella base. Quando vengono utilizzate la view definition viene prima computata e poi portata come valore d'ingresso.

3.7.1 Data indipendence, security

Il meccanismo delle viste permette il supporto per la data indipendece logica nel relational model: può essere utilizzato per definire relazioni che mascherano cambi nello schema concettuale. Inoltre permettono di definire viste che danno a gruppi di utenti solo le informazioni che hanno il privilegio di vedere.

3.7.2 Aggiornamenti su viste

Si permette di aggiornare viste solo se sono definite su una singola tabella base. La vista è chiamata updatable view.

3.8 Eliminare e alterare tabelle e viste

Tabelle e fiste possono essere rimosse attraverso *DROP TABLE* o alterate attraverso *ALTER TABLE*.

Capitolo 4

Relational Algebra

Un linguaggio di query permette di recuperare dati da un database. Il modello relazionale supporta linguaggi semplici ma potenti con forti fondamenti formali basati sulla logica che permettono molte ottimizzazioni. Questi linguaggi non sono Touring completi e non sono utilizzati per calcoli complessi, ma gestiscono in maniera ottima accessi a grandi gruppi di dati. Due linguaggi di query matematici sono alla base delle implementazioni: la relational algebra più operazionale, utile per rappresentare piani di esecuzione e il relational calculus che permette agli utenti di descrivere cosa vogliono e non come computarlo e non è operazionale ma dichiarativo. Una query è applicata alle istanze delle relazioni. Gli schema delle relazioni di input e di output sono fissati, determinati dalla definizione dei costrutti del linguaggio di query. La notazione dei campi di una tabella può essere posizionale in modo da facilitare la formalizzazione o nominativa in modo da facilitare la lettura.

4.1 Operazioni

- Selezione σ seleziona un sottoinsieme di righe di una relazione.
- Proiezione π elimina colonne non volute dalla tabella.
- Prodotto cartesiano $\times$ permette di combinare due relazioni.
- Differenza di insiemi $-$ le tuple nella relazione 1 ma non nella relazione 2.
- Unione $\cup$, tuple nella relazione 1 e nella relazione 2.

Essendo che ogni operazione ritorna una relazione possono essere composte. Questo dice che l'algebra è chiusa.

4.1.1 Proiezione

Elimina tutti gli attributi che non esistono nella lista di proiezione, lo schema risultato è una tabella che contiene esattamente gli attributi elencati con gli stessi nomi della tabela in input. Questa operazione deve eliminare i duplicati, cosa che non succede nei sistemi reali a meno che non venga specificato.

4.1.2 Selezione

La selezione seleziona le righe che soddisfano la condizione di selezione. Non ci sono duplicati nel risultato e lo schema rimane identico allo schema della tabella di input.

4.1.3 Unione, intersezione e differenza

Queste operazioni prendono come input due tabelle compatibili rispetto all'unione: stesso numero di campi e campi corrispondenti dello stesso tipo. Lo schema di risultato è quello analogo alle due tabelle.

4.1.4 Prodotto cartesiano

Ogni riga della prima tabella è accoppiata con una della seconda, lo schema risultato ha tutti i campi delle tabelle con i nomi ereditati se possibile, altrimenti si rende necessaria un'operazione di rinominazione: $\rho(C(i \to name_1, \ldots, j \to name_n), R_1 \times R_2)$.

4.1.5 Joins

4.1.5.1 Condition join

$R_1 \bowtie_C R_2 = \sigma_C(R_1 \times R_2)$, lo schema di risultato è lo stesso del prodotto cartesiano ma contiene meno tuple. A volte chiamato theta-join.

4.1.5.2 Equi-join

Un caso speciale del conition join dove la condizione contiene solo uguaglianze.

4.1.5.3 Natural join

Un equi-join su tutti i campi in comune.

4.1.6 Divisione

Non supportata come operazione primitiva ma utile. Si definisce come $A/B = \{\langle x\rangle | \exists \langle x, y\rangle \in A \forall y \in B\}$, ovvero se l'insieme di y valori associati con un valore x in A contiene tutte i valori y in B allora $x \in A/B$. In generale x e y possono essere qualsiasi lista di campi e y è la lista di campi in B e $x \cup y$ è la lista di campi in A.

4.1.6.1 Implementare una divisione

Si computino tutte le tuple che non sono squalificate da qualche valore y in B. Un valore x è squalificato se unendoci un valore y da B si ottiene una tupla che non sta in A. Questi valori si ottengono attraverso $\pi_x((\pi_x(A) \times B) - A)$ e $A/B : \pi_x(A) -$ tuple squalificate.

Capitolo 5

SQL

Una query basica in SQL ha la forma: *SELECT [DISTINCT] target-list FROM relation-list WHERE qualification*, dove relation-list è una lista di nomi di relazioni, possibilmente seguiti da una range-variable dopo ogni nome. La target-list è la lista di attributi delle relazioni in relation-list e qualification una comparazione tra attributi o tra attributo e costante con gli operatori $<$, $>$, $\leq$, $\geq$, $=$, $\neq$. DISTINCT è una parola chiave opzionale che indica se il risultato dovrà contenere duplicati.

5.1 Semantica

La semantica di una query SQL è definita secondo questa strategia: si computa il prodotto cartesiano della relation-list, si eliminano le tuple che non rispettano la qualification e se DISTINCT è specificato si eliminano i duplicati.Le range-variables sono necessarie unicamente se la tabella compare due volte in FROM, ma è buona pratica usarle sempre. I campi nel risultato si possono rinominare in SELECT attraverso *AS* o $=$, mentre *LIKE* permette di effettuare dello string matching, dove _ indica qualsiasi carattere e % per 0 o più istanze di un carattere arbitrario.

5.2 Operazioni

5.2.1 Unione

Può essere utilizzata per computare il risultato di ogni insieme di tuple compatibili per l'unione attraverso la parola chiave *UNION* o *EXCEPT* che computa il compatibile.

5.2.2 Intersezione

Viene utilizzata per computare l'intersezione di tuple compatibili per l'unione attraverso la parola chiave *INTERSECT*.

5.3 Nested query

Ogni clausola *WHERE* può contenere a sua volta una query per fare paragoni con parole chiave come *IN* o *EXISTS* che verificano l'esistenza del primo elemento nella query successiva, *UNIQUE* che verifica se è unico o operazione booleana con *ANY, ALL, IN*.

5.3.1 Divisioni in SQL

SELECT S.x FROM S WHERE NOT EXISTS (SELECT B.y FROM B WHERE NOT EXISTS (SELECT A.x FROM A WHERE A.y=B.y AND S.x=A.x)), o *SELECT S.x FROM S WHERE NOT EXISTS ((SELECT B.y FROM B EXCEPT (SELECT A.x FROM A WHERE S.x=B.x))*

5.4 Operatori aggregati

Sono un'estensione dall'algebra relazionale, permettono di svolgere operazioni su una singola colonna, sono: *COUNT, AVG, MAX, MIN, SUM.*

5.5 Grouping

SELECT [DISTINCT] target-list FROM relation-list WHERE qualification GROUP BY grouping-list HAVING group-qualification, dove la target list deve avere una lista attributi sottoinsieme della grouping-list. Ogni tupla di risposta corrisponde a un gruppo e gli attributi hanno un singolo valore per gruppo. Dopo aver eseguito il prodotto cartesiano della relation list si eliminano le tuple che non rispettano la qualificazione e quelle rimanenti sono partizionate in gruppi secondo i valori degli attributi nella grouping list, successivamente è applicata la group-qualification per eliminare le tuple che non la rispettano, tali espressioni devono avere un valore singolo per gruppo.

5.6 Null Values

Valori di campi nelle tuple possono essere sconosciuti o non specificati, pertanto per queste situazioni si introduce il valore *null*. La presenza di questo valore rende molto difficile garantire la verità delle condizioni *WHERE*. Si rendono possibili o necessarie nuove operazioni come per esempio gli outer joins.

5.7 Integrity constraints

Gli IC descrivono delle condizioni che ogni istanza legale di una relazione deve soddisfare, pertanto inserts, updates, deletes che li violano non vengono considerati. Gli IC possono essere di dominio, di primary key, di foreign key e generali. Quelli di dominio sono sempre forzati.

5.7.1 General constraints

Utili quando constraints più generali di quelli riguardanti unicamente le chiavi sono richiesti. Le queries possono essere utilizzate per creare questi constraints e possono essere nominati attraverso *Constraint <nome constraint> CHECK <descrizione constraint>*.

5.7.2 Constraints riguardanti più relazioni

Se un constraint riguarda più relazioni viene chiamato asserzione e creato attraverso la query *CREATE ASSERTION* in modo che non dipenda dall'effettiva popolazione di nessuna di quelle query.

5.8 Triggers

I triggers sono procedure che cominciano in maniera automatica se un certo evento accade all'interno del DBMS, sono formati da tre parti: l'evento che attiva il trigger, la condizione che verifica se l'azione del trigger debba essere applicata e l'azione che il trigger compie.

Capitolo 6

Normal form

Detta anche Boyce-Codd normal form.

6.1 Functional dependencies

Una dipendenza funzionale è un tipo generale di constraint sulle relazioni.

6.1.0.1 Sintassi

Sia $R(A_1, \ldots, A_n, B_1, \ldots, B_m, C_1, \ldots, C_k)$ uno schema relazionale, si definisce una dipendenza funzionale come:

$$A_1, \ldots, A_n \to B_1, \ldots, B_m$$

6.1.0.2 Semantica

Ogni qual volta due tuple t_1 e t_2 in un'istanza di R $A_1, \ldots, A_n$ determinano gli attributi $B_1, \ldots, B_m$. Se questo è dimostrato per ogni istanza della relazione si dice che R soddisfa la dipendenza funzionale.

6.2 Chiavi

Sia $A_1, \ldots, A_n$ degli attributi di R, questi sono una chiave se determinano funzionalmente tutti gli attributi di R e nessun sottoinsieme di $A_1, dots, A_n$ determina funzionalmente tutti gli attributi di R.

6.3 Decomposizione

L'idea è di decomporre la relazione in due per evitare i problemi che nascono da dati ridondanti. La ricomposizione avviene attraverso un join. Data la relazione $R(A, B, C)$ in cui vale la FD $A \to B$ se $R = \pi_{A,B}(R) \bowtie \pi_{A,C}(R)$ si dice che la decomposizione è lossless se non si perdono informazioni (il numero di tuple generato dall'operazione è uguale a quello di R).

6.3.1 Non vengono perse tuple

6.3.1.1 Enunciato

Per ogni relazione $R(A, B, C)$, $R \subseteq \pi_{A,B}(R) \bowtie \pi_{A,C}(R)$.

6.3.1.2 Dimostrazione

Sia $(a, b, c) \in R$, allora $(a, b) \in \pi_{A,B}(R)$ e $(a, c) \in \pi_{A,C}(R)$. Facendo un'operazione di join su queste tuple si ottiene $(a, b, c) \in \pi_{A,B}(R) \bowtie \pi_{A,C}(R)$.

6.3.2 Decomposizioni lossless e dipendenze funzionali

6.3.2.1 Enunciato

Se $A \to B$ in R allora $\pi_{A,B}(R) \bowtie \pi_{A,C}(R) \subseteq R$.

6.3.2.2 Dimostrazione

Sia $(a, b, c) \in \pi_{A,B}(R) \bowtie \pi_{A,C}(R)$, allora $(a, b) \in \pi_{A,B}(R)$ e $(a, c) \in \pi_{A,C}(R)$, pertanto esistono c' e b' tali che $(a, b, c') \in R$ e $(a, b', c) \in R$ ma $(a, b, c) \in R$ e $(a, b', c) \in R$ implica che $b = b'$, pertanto $(a, b, c) = (a, b', c) \in R$.

6.4 BCNF Boyce-Codd Normal form

6.4.1 Definizione

Una relazione è in BCNF se e solo se esiste una dipendenza funzionale non banale tale che $A_1A_2 \cdots A_n \to B_1B_2 \cdots B_m$ per R allora $\{A_1, \ldots, A_n\}$ è una chiave o superchiave per R.

6.4.2 Decomposizioni in BNFC

SI applica ripetutamente la tecnica di decomposizione in modo che sia lossless controllando per dipendenze funzionali non banali la cui parte sinistra non sia una superchiave e rompere lo schema in due fino ad ottenere un insieme di schema in BCNF.

6.4.2.1 Algoritmo di decomposizione

Un algoritmo di decomposizione in BCNF prende in input una relazione R_0 con un insieme di dipendenze funzionali F_0 e ritorna una decomposizione di R_0 in una collezione di relazioni ognuna delle quali sono in BCNF. Applicando l'algoritmo ricorsivamente cominciando con $R = R_0$ e $S = S_0$:

- Controllare se R è in BCNF, se sì ritornare $\{R\}$.
- Se esistono delle violazioni BCNF, considerandone una $X \to Y$ si scelga $R_1 = X \cup Y$ e $R_2 = \{att | att \in R \wedge att \notin Y)\}$. Si trovino gli insiemi di dipendenze funzionali che valgono su R_1 e R_2.
- Si decompongano ricorsivamenteR_1 e R_2 e si ritorni l'unione di tali decomposizioni.

6.5 Regole di inferenza per le dipendenze funzionali

- Dipendenze funzionali banali: se $\{B_1, \ldots, B_m\} \subseteq \{A_1, \ldots, A_n\}$ allora $A_1 \cdots A_n \to B_1 \cdots B_m$.
- Augmentation: se $A_1 \cdots A_n \to B_1 \cdots B_m$ allora $A_1 \cdots A_n C_1 \cdots C_k \to B_1 \cdots B_m$ per qualsiasi $\{C_1, \ldots, C_k\}$.
- Transitività: se $A_1 \cdots A_n \to B_1 \cdots B_m$ e $B_1 \cdots B_m \to C_1 \cdots C_k$ allora $A_1 \cdots A_n \to C_1 \cdots C_k$.

6.5.1 Inferenza per le dipendenze funzionali

Dato un insieme F di dipendenze funzionali si vuole apprendere se $X \to Y$ è una conseguenza di X. Questo si ottiene data X trovando la chiusura X^+ di X, ovvero l'insieme di tutti gli attributi che potrebbero essere sul lato destro di X: $X^+ = \{A | X \to A \text{segue da } F\}$.

6.5.1.1 Computare la chiusura

L'algoritmo prende in input un insieme di dipendenze funzionali F e un insieme $\{A_1, \ldots, A_n\}$ di attributi. Si assuma che ogni dipendenza funzionale abbia solo un attributo sulla destra. Alla fine ritorna la chiusura $\{A_1, \ldots, A_n\}^+$.

- Sia $X = \{A_1, \ldots, A_n\}$, alla terminazione X sarà la chiusura.
- Si trovi una dipendenza funzionale nella forma $B_1 \cdots B_m \to C$ tale che $\{B_1, \ldots, B_m\} \in X$ e $C \notin X$.
- Se non esiste tale F termina, altrimenti aggiungi C a X e ritorna al secondo passo.

Essendoci solo un numero finito di attributi l'algoritmo termina. Alla fine per vedere se $A_1 \cdots A_n \to B$ segue da F si controlla se B è in $\{A_1, \ldots, A_n\}^+$-

6.5.1.2 Dimostrazione dell'algoritmo di chiusura

Sia un insieme fissato di dipendenza funzionali F e S l'insieme degli attributi e X l'insieme computato dall'algoritmo e Y la chiusura. Si deve dimostrare che $X = Y$. Si dimostra prima che $X \subseteq Y$. La prova si fa di induzione sul numero di passi dell'algoritmo. Nel caso base tutti gli attributi di X sono già in S. Si assuma che $S \to A$ per ogni $A \in X$. Se si applica un algoritmo utilizzando una dipendenza funzionale la cui parte sinistra è contenuta in X si può utilizzare la transitività per mostrare che tutti i nuovi attributi sono determinati da X. Si dimostri ora che $Y \subseteq X$. Si supponga che C sia nella chiusura ma non in X. Sia lo schema di R $(A_1 \cdots A_n B_1 \cdots B_m)$ dove $X = \{A_1, \ldots, A_n\}$. Se C è in Y e non in X deve avere valori diversi in tuple diverse. Per induzione si dimostra che ad ogni passo dell'algoritmo i nuovi attributi aggiunti a X devono avere lo stesso valore nelle tuple e si genera una contraddizione.

6.5.2 Calcolo della BCNF

Essendo ora in grado di trovare tutte le dipendenze funzionali implicate dall'insieme si possono trovare tutte le violazioni BCNF e decomporre la relazione se necessario. Si devono trovare le dipendenze funzionali che valgono nelle nuove relazioni studiando le proiezioni delle dipendenze funzionali. Dato R L è un sottoinsieme degli attributi di R e le dipendenze funzionali sono F. Si devono trovare le dipendenze funzionali che valgono su $R_1 = \pi_L(R)$. Formalmente si devono trovare

le dipendenze funzionali seguono da F e coinvolgono unicamente attributi di L. Si deve trovare una base, un insieme minimo di dipendenze funzionali che implicano tutte le dipendenze funzionali che valgono su R_1.

6.5.2.1 Algoritmo

Sia l'input R, L, F come sopra e l'output l'insieme delle dipendenze funzionali che valgono in $\pi_L(R)$.

- Sia T l'insieme vuoto.
- Per ogni $X \subseteq L$ si computi X^+. Questo potrebbe contenere attributi non in L.
- Si aggiungano a T tutte le dipendenze funzionali della forma $X \to A$ quando $A \in X^+ \cap L$.
- Si minimizzi l'insieme ripetendo ogni volta possibile:
  - Se ci sono dipendenze funzionali in T che segue dalle altre eliminala.
  - Se $YZ \to B$ è in T e $Z \to B$ segue da F la si sostituisca con $Z \to B$

Capitolo 7

Storage and indexing

I dati possono essere salvati su disco, in cui considerare pagine randomiche ha un costo fisso ma leggere pagine in sequenza è più efficente o tapes in cui i dati sono recuperati solo sequenzialmente. Si chiama file organization un metodo di ordinare i file di records su memoria esterna. Un record id è sufficiente per localizzarlo fisicamente e gli indici sono strutture dati che permettono di trovare i record id dati i valori nei campi di index search key. Un buffer manager prende pagine dalla memoria esterna al main memory buffer pool. I layer di file e indici fanno chiamate al buffer manager. Esistono molte alternative, scelte in base alle necessità:

- Heap file: ottima quando un tipico accesso è un file scan che recupera tutti i record.
- Sorted files: ottima quando i record devono essere recuperati in un certo ordine o secondo un certo range.
- Indexes: strutture dati organizzate secondo alberi o hashing.

7.1 Indexes

Un indice su un file rende più veloce le selezioni sul search key fields per l'indice. Ogni sottoinsieme di campi di una relazione può essere un search key per l-indice della relazione. Non è una chiave della relazione. Un indice contiene una collezione di data entries e supporta un efficiente recupero di tutte le data entries $k*$ con una key value k. Dato $k*$ si trovano record con la chiave k in al più in un'operazione di disco I/O.

7.1.1 B+ Tree Indexes

Le pagine foglia contengono data entries e sono concatenate, le pagine non foglia contengono unicamente index entries e sono utilizzate unicamente per direzionare la ricerca. Gli insert o delete richiedono di cercare dati foglia e poi cambiarne i valori. I cambi necessari possono riguardare anche livelli più alti dell'albero.

7.1.2 Hash based indexes

Sono buoni per le selezioni d'uguaglianza. Gli indici sono una collezione di buckets, formati da primary pages e un numero arbitrario di overflow pages. Si rende necessaria una hasing function

h. Il risultato di $h(r)$ è il bucket cui r appartiene h controlla i search key fields di r. Per trovare il bucket di r si prendono gli ultimi *global depht* numeri di bit di $h(r)$. Per l'insert, se il bucket è pieno lo si deve splittare allocando una nuova page e redistribuendo. Se necessario si deve anche raddoppiare la directory: per determinare il raddoppio si devono confrontare global depth e local depth.

7.1.3 Alternative di data entry k∗ nell'indice

In una data entry $k*$ si possono salvare data record con valore di chiave k o riduzioni di data records con valore di search key pari a k o liste di riduzioni di data records con chiave di ricerva k. La scelta per alternative data entries è ortogonale alla tecnica di indexing utilizzata per localizzare data entries con un dato valore di chiave k. Tipicamente gli indici contengono informazioni ausiliarie che direzionano le ricerche verso le data entries desiderate.

7.1.3.1 Salvare data records con valore chiave k

Se viene utilizzata questa alternativa la struttura di indice è un file di organizzazione per data records. Al massimo un indice di una data collezione di data records può utilizzare questa alternativa. Se vengono salvati numerosi data records il numero di pagine che contiene data entrie è alto e implica che la dimensione delle informazioni ausiliare dell'indice è alta.

7.1.3.2 Salvare riduzioni di data records o liste di riduzioni

Le data entries sono tipicamente molto minori rispetto a data records, pertanto funziona meglio con data records grandi, specialmente se le chiavi sono piccole. La terza alternativa è più compatta della seconda, ma porta a data entries di grandezza variabile anche se le chiavi di ricerca sono di lunghezza fissa.

7.2 Classificazione degli indici

Se una search key contiene una primary key è chiamata un primary index, altrimenti un secondario. Se contiene una candidate key è detto unique.

7.2.1 Clustered e unclustered

Se l'ordine dei data records è lo stesso o simile ordine delle data entries si chiama l'indice clustered, implicato dalla prima alternativa per le data entries. Un file può essere clustered su al più una search key. Il costo di recupero dei dati attraverso indici dipende fortemente da questa caratteristica.

7.2.1.1 Indici clustered

Per creare un indice clustered su data entries salvate in un Heap file si deve ordinare l'heap file lasciando spazio libero nelle pages per inserts futuri e pagine di overflow potrebbero rendersi necessarie in futuro. L'indice è clustered se ogni data entries con lo stesso valore di search key si trova sequenzialmente alle altre.

7.3 Scelta degli indici

Per creare degli indici ottimi si considerino le queries più importanti e il piano migliore utilizzando gli indici correnti. Si guardi se esiste un piano migliore con indici addizionali e se è così lo si implementi. Prima di creare un indice si deve considerare l'impatto sugli updates nella workload: si deve fare un trade-off in quanto gli indici velocizzano le queries ma rallentano le operazioni di update e richiedono spazio su disco.

7.3.1 Linee guida sulla selezione degli indici

- Gli attributi nelle clausole WHERE sono candidati per le chiavi di indice. Una condizione di match esatto suggerisce indici hash e query di range suggeriscono indici ad albero (il clustering aiuta molto in questo ambito).
- Search key multi attributo dovrebbero essere considerate quando una clausola WHERE contiene mutliple condizioni. L'ordine degli attributi è importante per le range queries. Tali indici possono qualche volta rendere possibili strategie index-only.
- Si provi a scegliere indici che diano benefici a più queries possibile. Essendo che un unico indice può essere clustrerizzato per relazione lo si scelga in base alle queries più importanti da cui ne trarrebbero beneficio.

7.4 Indici con searh keys composte

Una chiave di ricerca composta è una chiave di ricerca su una combinazione di campi. Le data entries nell'indice sono ordinate in modo da supportare range queries.

Capitolo 8

Teoria del costo

8.1 Definizioni

8.1.1 Page

Ogni volta che a un hard disk viene richiesto di leggere o scrivere informazioni questo lo fa operando su unità di lunghezza specifica conosciute come pages, denotate con P.

8.1.2 Dimensione di un record

I record di una relation hanno tutti la stessa dimensione, che indica quanti byte questo occupa quando viene salvato nel disco. Se non è data la dimensione di un record si ottiene dai tipi degli attributi che lo compongono. È denotata con t_R.

8.1.3 Pages di una relazione

I dati di ogni relazione sono salvati su disco. Su ogni page contenente dati di una relazione non sono permessi record di un'altra relazione. Il database cerca pertanto di riempire la pagina con più record possibili provenienti dalla stessa relazione. Si indica con P_R il numero di pagine che la relazione occupa.

8.1.4 Cardinalità di una relazione

La cardinalità di una relazione è il numero di record che essa contiene: $|R|$.

8.1.5 Cardinalità di un attributo

Si intende con cardinalità di un attributo il numero di valori distinti che questo assume nei record della relazione. Per un attributo A della relazione R si indica con $|R.A|$. Le chiavi primarie hanno la stessa cardinalità della relazione, mentre per un attributo generico A si avrà $|R| \geq |R.A|$.

8.1.6 Record per page

Dato che i record contenuti in una page provengono dalla stessa relazione e un record non può essere diviso tra più page, il numero di record di una relazione R in una page di dimensione P sarà $\lfloor \frac{P}{t_R} \rfloor$.

8.1.7 Dimensioni di una relazione

La dimensione di una relazione è lo spazio che occupa su disco. Essendoci la possibilità che si avanzi spazio nelle pages, la dimensione effettiva può essere superiore di quella reale: $\lceil \frac{|R|}{\lfloor \frac{P}{t_R} \rfloor} \rceil$.

8.1.8 Costo

Operazioni che coinvolgono lettura o scrittura su disco sono dette di I/O, quelle fatte solo in memoria sono dette in-memory. Essendo le prime molto più costose sono quelle che determinano il costo di un'operazione generica. Si assume per definizione che il costo per leggere o scrivere è pari a 1.

8.2 Operazioni

8.2.1 Scan

Un'operazione di scan è una lettura sequenziale della relazione. Il costo è il costo di lettura di tutte le pagine occupate dalla relazione, pertanto $Costo\ of\ scan = \#pages\ in\ relation$. Affinchè il database possa trovare e leggere un record in una relazione normalmente richiede di effettuare uno scan.

8.2.2 Sorting

Se si deve ordinare una relazione basandosi sul valore di un attributo specifico ci sono due algoritmi specifici: uno dei due assume che il database possiede solo tre buffer nella sua memoria. Se N è il numero di pagine occupate dalla relazione il costo del sorting è $2N(|\log_2 N| + 1)$. Tipicamente il database possiede più di tre buffer pertanto si può utilizzare un algoritmo ottimizzato. Se ci sono B buffer a disposizione il costo di sorting è $2N(|\log_{B-1}\lceil \frac{N}{B} \rceil| + 1)$.

8.2.3 Indici

Gli indici sono strutture ausiliarie che sono costruite basate sul valore di un attributo specifico. Aiutano a identificare la posizione del record che possiede un valore nell'attributo che soddisfa delle condizioni.

8.2.3.1 Indice Hash

Aiuta a trovare records che soddisfano condizioni di uguaglianza sull'attributo indicizzato.

8.2.3.2 Indice B+ tree

È un indice che aiuta a trovare records che soddisfano condizioni di uguaglianza o di range.

8.2.4 Indici composti

È possibile creare indici che dipendono da più di un attributo. Sono chiamati indici composti. Nel caso degli alberi B+ l'ordine con cui questi attributi sono definiti gioca un ruolo: i dati sono indicizzati prima basandosi sul primo campo e successivamente sul secondo, poi sul terzo. Pertanto un albero B+ può essere utilizzato quando si cerca per attributi sull'indice composto e quelli al suo inizio ma non alla sua fine.

8.2.5 Aggiornare indici

Ogni volta che una relazione è aggiornata anche l'indice deve esserlo e questo può influire sul tempo di esecuzione della query, pertanto si deve limitare il numero di indici.

8.2.6 Lookup di indici

L'operazione di utilizzare l'indice per identificare dove i records che soddisfano una condizione sono posizionati è chiamata lookup di indice. L'output è un insieme di puntatori alle pages che contengono i records. Gli indici occupano anche spazio. In particolare il costo medio di un indice hash è di 1.2, mentre per un albero B+ è di $\log_s |R.A|$ dove s è il numero massimo di figli di un nodo dell'albero. Il costo di lookup è denotato come L.

8.2.7 Il fattore di selettività

Una questione importante è sapere quanti qualifiyng records sono recuperati data una condizione. Data una condizione c e una relazione R si denota con R_c i records di R che soddisfano c. Questi dati sono mantenuti nel database come statistiche. Si può assumere che i valori abbiano un'uguale distribuzione, pertanto se si ha una relazione R con un attributo A, il numero di qualifiyng records che soddisfano la condizione $A = x$ sono $|R_{A=c}| = \frac{|R|}{|R.A|}$. Si definisce il fattore di selettività la percentuale del numero totale di records che si aspetta siano recuperati da una condizione, pertanto per il fattore f, $|R_c| = f \cdot |R|$.

8.2.8 Clustered/Unclustered

Un indice può essere clustered o unclustered. Il primo vuol dire che i records che soddisfano una condizione sono salvati sequenzialmente. In caso di un indice unclustered il lookup ritorna un page-pointer per ogni qualifying record sul disco, mentre nel caso di un indice clusterizzato un puntatore è ritornato per le pagine che contengono il qualifying record. Per una condizione c su una relazione R il numero di page pointers ritornato da un lookup di un indice clusterizzato è: $\#page\ occupate\ dalle\ tuple = \frac{\#tuple\ che\ soddisfano\ la\ condizione}{\#tuple\ di\ R\ che\ stanno\ in\ una\ page} = \frac{|R_c|}{\lfloor \frac{P}{t_r} \rfloor}$.

8.2.9 Costo di recupero dei qualifying records

Recuperare qualifying records dopo un lookup dell'indice ha come costo la lettura delle pagine per cui il lookup ha ritornato un puntatore. Se l'indice è unclusterizzato il costo è $L + |R_c|$, se è clusterizzato il costo è $L + \frac{|R_c|}{\lfloor \frac{P}{t_r} \rfloor}$.

8.2.10 Costo di un operatore di update

Il costo di un update è il costo del lookup dell'indice più il costo di lettura della pagina più il costo di scrittura della pagina: $Lookup + \#pages\ che\ contengono\ il\ record\ da\ cambiare \cdot (1 + 1)$.

8.2.11 Join

Il join è l'operatore più utilizzato pertanto esistono diversi algoritmi per implementarlo efficientemente. Siano date due relazioni R e S.

8.2.11.1 Nested loops join

È il metodo che può essere sempre fatto in quanto non necessita di nessuna struttura ausiliaria. Legge la prima relazione e per ogni pagina della prima legge l'intera seconda. Pertanto $R \bowtie S$ avrà un costo di $P_R + P_S \cdot P_R$. È sempre meglio cominciare dalla relazione più piccola.

8.2.11.2 Sort merge join

Si ordina ogni relazione e poi si svolge il join attraverso un passaggio verso ogni relazione, pertanto il costo è $costo\ per\ ordinare\ R + costo\ per\ ordinare\ S + P_R + P_S$.

8.2.11.3 Hash join

Assumendo di avere abbastanza memoria per salvare una struttura hash si può costruirla e utilizzarla per fare il join. Ha costo $3 \cdot (P_R + P_S)$.

8.2.11.4 Index nested loops join

Utilizzato quando c'è un indice che può essere sfruttato. Si legge una relazione e poi per ogni record si usa l'indice dell'altra per selezionare i matching records. L'indice della seconda relazione deve essere sull'attributo di join. Il costo sarà $costo\ per\ leggere\ R + \#recordsInR \cdot costo\ per\ recuperare\ matching\ records\ in\ S = P_R + |R| \cdot (Index\ Lookup\ cost + cost\ of\ retrieving\ qualifying\ records)$.

8.2.12 Salvare i risultati di una query

I risultati di una query possono essere risalvati nel database. Il costo di questa operazione è il numero di pages in cui si rende necessario scrivere, ovvero il numero di pagine che il record occupa. $\#pages\ for\ the\ tuples = \lceil \frac{\#tuples}{\#tuples\ per\ page} \rceil = \frac{\#tuples}{\lfloor \frac{P}{t} \rfloor}$.

8.2.13 Esecuzione e query plans

Quando una query viene passata al database per l'esecuzione questo la trasforma in un query plan, un albero di operazioni in algebra relazionale che implementano la query. Possono esistere diversi query plans per la stessa query. Dato un query plan si rende necessario computare il costo totale, ovvero il costo di ogni operatore del nodo. Per ogni nodo si rende necessario sapere il costo, il numero di records che genera. Il database possiede una componente speciale detta ottimizzazione il cui ruolo è valutare le queries che riceve e identificare il piano di esecuzione più efficace.

8.2.14 Operatori on the fly

Si definiscono operatori on the fly gli operatori la cui operazione può essere svolta mentre i records arrivano in quanto non si necessita di aspettare per l'arrivo di tutti i risultati. Il loro costo è zero in quanto vengono svolti in memoria.

8.2.15 Operatori index-only

Un'operazione che non richiede acceso ai records ma può trovare dall'indice ciò che è richiesto.

Capitolo 9

Transazioni

In applicazioni reali le operazioni possono essere richieste in maniera concorrente e due operazioni che riguardano le stesse tuple eseguite allo stesso tempo possono portare a risultati errati. Si definiscono pertanto le transazioni, un insieme di operazioni che devono essere svolte insieme. Due transazioni devono comportarsi come se fossero eseguite serialmente ma alcune loro parti possono sovrapporsi. Maggiore la sovrapposizione, migliori le prestazioni del sistema. Una sequenza di operazioni è serializzabile se è uguale a una esecuzione seriale delle transazioni. Un ulteriore problema nasce quando due operazioni devono per forza essere svolte atomicamente: se occorre un crash tra la prima e la seconda il database viene lasciato in uno stato incoerente. Pertanto si raccolgono operazioni sul database in transazioni, collezioni di diverse operazioni sul database che devono essere eseguite atomicamente: o si svolgono tutte o nessuna. L'SQL di default richiede che le transazioni siano eseguite in modo serializzabile: una transazione è eseguita nella sua interezza prima dell'altra.

9.1 Testare la serializzabilità

Sia una schedule un'esecuzione di transazioni con operazioni sovrapposte. Si determina un'invariante, una condizione che se vera prima dell'esecuzione delle transazioni deve rimanere vera anche al loro termine.

9.1.1 Testing

Per testare la serializzabilità si devono ignorare i dettagli e testare se la schedule sarà serializzabile in ogni caso. Si analizzano solo la sequenza di letture e scritture. Il principio generale è di creare un algoritmo che non dia falsi positivi. Falsi negativi non creano inconsistenze ma unicamente una degradazione delle prestazione. Si indichi:

- $r_T(X)$ la transazione T che legge elementi dal database X.
- $w_T(X)$ la transazione T che scrive elementi del database X.
- Se le transazioni sono indicizzate rispetto a i si scrive $w_i(X)$ invece di $w_{T_i}(X)$.
- Si indica come azione un'espressione nella forma $r_i(X)$ o $w_i(X)$.
- Una transazione T_i è una sequenza di azioni indicizzate a i.

- Una schedule S è un insieme di transizioni T, è una sequenza di azioni in cui per ogni transazione T_i in T le azioni di T_i appaiono in S nello stesso ordine in cui appaiono nella definizione di T_i.
- Si dice che S è sovrapposta sulle azioni delle transazioni che la compongono.

9.1.2 Serializzabilità del conflitto

Data una schedule S si esamino le azioni consecutive da differenti transazioni T_i e T_j.

- $r_i(X)$ e $r_j(X)$ non sono mai in conflitto in quanto nessuna di esse cambia il valore degli elementi del database.
- $r_i(X)$ e $w_j(Y)$ con $X \neq Y$ non sono in conflitto: se T_j scrive Y prima che T_i legga X il valore di X non cambia. Anche nel caso contrario, la lettura di X svolta da T_i non ha effetti su T_j.
- $w_i(X)$ e $r_j(Y)$, $X \neq Y$ non è in conflitto.
- $w_i(X)$ e $w_j(Y)$, $X \neq Y$ non è in conflitto.

Se due azioni non sono in conflitto si possono invertire tra di loro senza cambiare il risultato.

- Due azioni della stessa transizione (ad esempio $r_i(X)$ e $w_i(Y)$) sono sempre in conflitto: non si può cambiare l'ordine di azioni in una transizione singola.
- $w_i(X)$ e $w_j(X)$ sono in conflitto in quanto il valore di X è quello computato da T_j. Invertendo l'ordine si ottiene X con il valore computato da T_i.
- $r_i(X)$ e $w_j(X)$ sono in conflitto: invertendo l'ordine il valore di X letto da T_i sarà quello scritto da T_j che potrebbe essere differente dal valore precedente di X.
- $w_i(X)$ e $r_j(X)$ sono in conflitto per ragioni simili.

Si nota come due azioni da transazioni diverse possono essere invertite a meno che involvono lo stesso elemento del database e almeno una è una scrittura. Data una schedule si fa il massimo numero possibile di inversioni necessarie con lo scopo di trasformare una schedule in una seriale. Se si può fare la schedule originale è serializzabile. Due schedules sono conflict-equivalent se ognuna di esse può essere trasformata nell'altra attraverso una sequenza di inversioni senza conflitto di azioni adiacenti. Una schedule si dice conflict-serializable se è conflict-equivalent a una schedule seriale. Conflict-serializability implica serializzabilità. L'inverso è falso ma conflict-serializability è solitamente sufficiente.

9.1.3 Testare conflict-serializability

Data una schedule S che coinvolge due transazioni T_1 e T_2 T_1 ha precedenza su T_2 ($T_1 <_S T_2$) se ci sono azioni A_1 di T_1 e A_2 di T_2 tali che:

- A_1 si trova prima di A_2 in S.
- Sia A_1 che A_2 coinvolgono lo stesso elemento del database.
- Almeno una di A_1 e A_2 è un'azione di scrittura.

Si noti come queste sono esattamente le condizioni per cui non si può invertire l'ordine di A_1 e A_2, pertanto A_1 appare prima di A_2 in ogni schedule che è conflict-equivalent ad S, pertanto ogni schedule seriale conflict-equivalent deve avere T_1 prima di T_2.

9.1.3.1 Grafo delle precedenze

Si dice grafo delle precedenze un grafo orientato i cui nodi sono le transazioni di S e un arco $(T_i, T_j) \in E$ se e solo se $T_i <_S T_j$.

9.1.3.2 Algoritmo

Si costruisca il grafo delle precedenze, si provi se possiede cicli, se li possiede S non è serializzabile, altrimenti qualsiasi ordine topologico dei nodi è un ordine seriale conflict-equivalent.

9.1.3.3 Correttezza

Si dimostri che se esiste un ciclo la schedule non è serializzabile: si supponga che esiste un ciclo di lunghezza n, allora le azioni di una transazione devono precedere quelle della transazione stessa e si raggiunge pertanto una contraddizione. Si dimostra ora che se la schedule è serializzabile allora non esiste un ciclo: si provi per induzione sul numero di transizioni: se il grafo n non ha cicli si possono riordinare le schedules actions utilizzando inversioni legali in modo che la schedule diventi seriale. Se $n = 1$ la schedule deve essere seriale. Si assuma la tesi vera per tutti i valori minori di n e lo si provi per n. La schedule consiste di azioni di transazioni $T_1, \ldots, T_n$ con un grafo aciclico S, se il grafo è aciclico si deve avere un nodo i senza archi in entrata, ovvero nessun nodo lo precede nell'ordinamento pertanto non ci sono altre azioni che coinvolgono una transazione T_j tali che precedono delle azioni di T_i e sono in conflitto con tali azioni. Si possono muovere tutte le azioni di T_i all'inizio della schedule.

9.2 Aspetti pratici

Non si può utilizzare l'algoritmo in quanto si dovrebbe aspettare che tutte le transazioni siano finite e si necessiterebbe di mantenere i dati per ogni transazione. L'algoritmo inoltre è inefficiente. Algoritmi come timestamps permettono di controllare ogni transazione quando finisce e mantengono informazioni limitate su ogni transazione. Quando si scopre un problema si deve cancellare la transazione e disfare ogni cambio fatto al database e portare questo cambio verso tutte le transazioni che nel frattempo hanno modificato il database. Questo funziona solo se questo tipo di problemi sono non frequenti, altrimenti si fa in modo che le transazioni possano procedere unicamente se è possibile garantire la serializzabilità.

9.3 Locks

I locks sono mantenuti su elementi del database per evitare un comportamento non serializzabile. Intuitivamente una transazione che ottiene locks sull'elemento del database vi accede per impedire ad altre transazioni di accedervi allo stesso tempo. Questo garantisce la serializzabilità.

9.3.1 Oggetti

L'algoritmo accede ad oggetti che sono relazioni, tuple, parti di relazione, disk blocks. Più piccoli sono gli oggetti, più ne possono essere eseguiti in parallelo ma il costo di tenere conto di cosa viene lockato può essere proibitivo. Si assume che gli oggetti siano tuple. Esiste uno scheduler che locka le tabelle. Tale scheduler riceve richieste dalle transazioni e permette loro di essere eseguite o le blocca fino a quando è sicuro farlo. Idealmente lo scheduler inoltra una richiesta ogniqualvolta

la sua esecuzione non può portare a uno stato inconsistente nel database. Un locking scheduler dovrebbe garantire conflict-serializability. Ora le transazioni devono richiedere e rilasciare locks oltre che leggere e scrivere oggetti del database. Si devono soddisfare le proprietà di consistenza delle transazioni: una transazione può leggere o scrivere un elemento solo se precedentemente le è stato garantito un lock su quell'elemento e non è stato ancora rilasciato; se una transazione locka un elemento deve in un secondo momento rilasciarlo. La legalità delle schedules: i locks devono avere il loro significato inteso: un lock dello stesso elemento può essere posseduto unicamente da una transazione.

9.3.1.1 Notazione

- Sia $l_i(X)$: T_i richiede un lock su un elemento X del database.
- Si nota che è compito dello scheduler decidere se assecondare la richiesta o no.
- $u_i(X)$: T_i rilascia il suo lock su X.
- Consistenza: ogniqualvolta T contiene un'azione $r_i(X)$ o $w_i(X)$ deve esserci un'azione precedente $l_i(X)$ e la prima occorrenza di $u_i(X)$ deve trovarsi dopo l'azione.
- Legalità delle schedules: se una schedule contiene un'azione $l_i(X)$ seguita da $l_j(X)$ deve esiste un'azione $u_i(X)$ tra le due.

9.3.2 Locking scheduler

Il compito del locking scheduler è di garantire richieste come $l_i(A)$ se e solo se la richiesta risulta in una schedule legale. Se la richiesta non è garantita la transizione è ritardata. La transazione aspetta fino a che lo scheduler garantisce la richiesta più tardi. Lo scheduler utilizza una tabella dei lock che dice per ogni elemento del database quale transazione o transizioni possiedono il lock per l'elemento.

9.3.2.1 Two-phase locking

Il two-phase locking (2PL): in ogni transizione tutte le azioni di lock precedono tutte le azioni di unlock in modo da garantire la serializzabilità. Nella prima fase di una transazione i lock sono ottenuti, nella seconda fase sono rilasciati. Una transazione che rispetta le condizioni 2PL è chiamata una transazione two-phase-locked.

9.3.2.1.1 Dimostrazione

Si assuma che S sia legale in una schedule 2PL. Si mostra come convertire S in una conflict-equivalent schedule seriale. Si provi per induzione su n, il numero di transazioni in S. Si usano le operaizoni di lock e unlock per la prova.

- $n = 1$: S è già una schedule seriale.
- Si assuma che il risultato valga per schedules con transazioni $n-1$: sia T_i la transazione con la prima azione di unlock nella schedule, $u_i(X)$. Si mostra che si possono spostare tutte le azioni di T_i all'inizio della schedule senza conflitti. Si consideri qualche azione T_i, $w_i(Y)$, affinchè possa essere preceduta in S da qualche azione con conflitto come $w_j(Y)$ j deve lockare l'item e i deve successivamente lockarla. Ma per definizione $u_i(X)$ è la prima operazione di unlock, pertanto deve essere prima di $u_j(Y)$, ma allora $u_i(X)$ appare prima di $l_i(Y)$ contrariamente al protocollo 2PL e si deve pertanto riscrivere S come $T_i(T_{n-1} \cdots T_0)$.

Con questo algoritmo sono possibili deadlocks: tutte le transazioni stanno aspettando per la liberazione di un lock che non avverrà mai.