Architetture Dati

UniShare

Davide Cozzi
@dlcgold

Indice

Capitolo 1

Introduzione

Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`.

Capitolo 2

Sistemi centralizzati

Definizione 1. *Un **DBMS (DataBase Management System)** è un sistema, ovvero un software, in grado di gestire collezioni di dati che siano:*

- *grandi, ovvero di dimensioni maggiori della memoria centrale dei sistemi di calcolo usati (se ho a che fare con una quantità di dati non così grande e con un uso personale posso affidarmi ad una hashmap piuttosto che ad un db)*
- *persistenti, ovvero con un periodo di vita indipendente dalle singole esecuzioni dei programmi che le utilizzano e per molto tempo*
- *condivise, ovvero usate da diversi applicativi e diversi utenti (fattore che porta anche allo studio del carico di lavoro, workload). L'accesso può essere sia in scrittura che in lettura (ovviamente anche entrambi) a seconda del caso. SI pongono quindi problemi di concorrenza e sicurezza*
- *affidabili, sia resistente dal punto di vista hardware (un guasto non deve farmi perdere i dati) che dal punto di vista della sicurezza informatica. Le transazioni devono essere quindi **atomiche** (o tutto o niente) e **definitive** (che non verranno più dimenticate). Il software può cambiare mentre i dati no*

A livello di architettura per un *sistema centralizzati* si hanno:

- uno o più *storage* per memorizzare i dati, a loro volta su uno o più file del *file system*
- il *DBMS*, il componente software che funge da componente logico

- diverse applicazioni che elaborano i dati provenienti dal db (*lettura*) ed eventualmente scrivono dati sullo stesso (*scrittura*)
- il **DBA (*DataBase Administrator*)** che tramite riga di comando o GUI si occupa di manutenzione, sicurezza, ottimizzazione etc... del DBMS

L'*architettura dati* di un DBMS è definita dall'ente *ANSI/SPARC* e è a tre livelli:

1. diversi **schemi esterni**, porzioni di db messi a disposizione per le varie applicazioni
2. uno **schema logico (o concettuale)**, che fa riferimento al *modello relazionale* dei dati ed è indipendente dalla tecnologia usata. Avendo un unico schema logico si ha un'unica semantica (perlomeno a livello astratto). Si ha unica base di dati, quindi un unico insieme di record interrogati e aggiornati da tutti gli utenti. Non si ha nessuna forma di eterogeneità concettuale
3. uno **schema fisico**, che fa riferimento alla tecnologia usata per implementare le tabelle per salvare i dati. Si ha un'unica rappresentazione fisica dei dati e quindi nessuna distribuzione e nessuna eterogeneità fisica

Un unico schema fisico è collegato ad un unico schema logico.
Inoltre si hanno:

- un **unico linguaggio di interrogazione** e quindi un'unica modalità di accesso ai dati
- un unico sistema di gestione per accesso, aggiornamento e gestione per la transazioni e le interrogazioni
- un'unica modalità di ripristino in caso d'emergenza
- un unico amministratore dei dati
- **nessuna autonomia gestionale**

Per il discorso della persistenza dei dati si ha necessità di una memoria secondaria dove il DBMS salva le strutture dati, studiando un modo efficiente di trasferimento dei dati nel *buffer* in memoria centrale. Il *buffer* è un'area di memoria (o meglio un componente software) nella memoria centrale che

cerca, tramite una logica di "vicinanza", di mettere i dati della memoria secondaria in quella centrale. Si usa il **principio di località**.
A causa degli accessi condivisi al db si hanno problemi di **concorrenza**, avendo accesso multi-utente alla stessa dei dati condivisa, accesso che necessità anche di meccanismi di **autorizzazione**. In merito alla concorrenza si ha che le transazioni sono corrette se **seriali** (ordinate temporalmente) ma questo non è sempre applicabile e quindi si deve stabilire un *controllo della concorrenza*.
Per accedere ai dati di un db si hanno le **query (*interrogazioni*)** che fanno parte del modello logico a cui si interfaccia l'utente. Essendo i dati nelle memorie secondarie bisogna cercare un modo di rendere gli accessi performanti, in primis tramite opportune strutture fisiche in quanto e strutture logiche non sarebbero efficienti in memoria secondaria. Bisogna fare in modo che gli accessi alla memoria secondaria siano il più limitati possibili e quindi bisogna ottimizzare l'esecuzione delle query. Ovviamente una scansione lineare delle tabelle sarebbe troppo dispendiosa con tabelle grosse, ricordando che i file sono ad accesso sequenziale. Inoltre un ipotetico *join* tra tabelle renderebbe ancora più complesso l'accesso, soprattutto se *full-join*.
Per poter garantire tutto ciò che è stato detto l'architettura del DBMS deve essere organizzata in termini di *funzionalità cooperanti*:

- un **query compiler** che prende una query in SQL e la traduce con un compilatore
- un **gestore di interrogazioni e aggiornamenti** che trasforma le query in SQL in algebra relazionale facendo operazioni di ottimizzazione
- un **gestore dei metodi di accesso** per permettere il passaggio tra file e tabelle passando dal **gestore del buffer** e il **gestore della memoria secondaria** dove i dati non sono in forma tabellare ma di file e pagine
- un **DDL compiler**, dove DDL sta per Data Description Language, che si occupa dei comandi del DBA
- un **gestore della concorrenza**, che garantisce il controllo della concorrenza
- un **gestore dell'affidabilità**, che garantisce che un dato non vada perso
- un **gestore delle transazioni**

Gli ultimi quattro entrano in uso specialmente in fase di scrittura.
Tutto deve essere veloce!
In un sistema distribuito la parte di query compiler, gestore delle interrogazioni, gestore delle transazioni e gestore della concorrenza resta invariato mentre il resto cambia drasticamente (in quanto i dati sono distribuiti) dovendo gestire diversamente l'accesso ai dati e la sua sicurezza. Bisogna gestire anche come i vari nodi devono interagire coi dati.

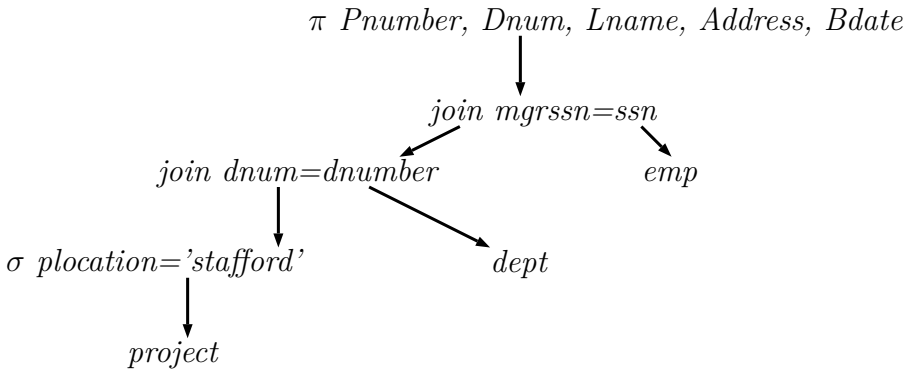
2.1 Ottimizzazione delle query

Ottimizzare le query è tutt'altro che banale.
Il primo step è il **parsing**, che stabilisce se la query è sensata dal punto di vista sintattico e se i vari nomi di tabelle e attributi sono coerenti con lo schema. Per questo ultimo aspetta ci si appoggia al **Data Catalog**, un particolare db che contiene informazioni sui vari database, in primis sui vari nomi delle tabelle e per ciascuna sui nomi di ogni attributo. SI ha quindi una soluzione per gestire i *metadati*. Il parser effettua un'analisi lessicale, per la sintattica e la semantica, usando il dizionario e l a traduzione in algebra relazionale, producendo un **query tree**. Si calcola anche un **query plan logico**, utilizzando regole sintattiche di buon senso, per capire cosa fare prima (per esempio se fare prima una *select* o un *join*) per ottenere il risultato corretto nel minor tempo possibile (prima di fare una *join* magari seleziono prima una sottotabella con i dati potenzialmente utili, togliendo quelli sicuramente inutili... magari quel *join* può anche essere evitato). La query viene quindi rappresentata come un albero dove e foglie corrispondono alle strutture dati logiche, ovvero le tabelle. I nodi interni sono invece le varie **operazioni algebriche (*select, join, proiezione, prodotto cartesiano* e *operazioni insiemistiche*)**.

Esempio 1. *Vediamo una query:*

```
SELECT Pnumber, Dnum, Lname, Address, Bdate
FROM Project P, Dept D, Emp E
Where P.dnum=D.dnumber and E.ssn = D.mgrssn
and P.location = 'Stafford'
```

che produce:



ma l'ottimizzatore va oltre (magari invertendo i where etc...) con un query plan più efficiente che permette di cambiare automaticamente le query in altre più efficienti.

Si ha un db chiamato **Statistics** che contiene statistiche sulla storia delle query nonché altre informazioni sui dati. L'uso di tale db permette di ottimizzare le query.
Solo dopo questo processo si ha la trasformazione delle tabelle logiche in strutture fisiche e metodi di accesso alla memoria e la trasformazione delle operazioni algebriche nelle loro implementazioni sulle strutture fisiche. Per la trasformazione si usano proprietà algebriche e una stima dei costi delle operazioni fondamentali per diversi metodi di accesso (in poche parole le regole della ricerca operativa). L'ottimizzazione ha complessità **esponenziale** e quindi si introducono approssimazioni basate su euristiche, usando un'**alberatura di costi** usando la tecnica del **Branch&Bound**.

2.2 Transazioni

Una **transazione** è l'insieme di istruzioni di accesso in lettura e scrittura ai dati, istruzioni eventualmente inserite in un linguaggio di programmazione. Una transazione gode di proprietà che garantiscono la corretta esecuzione anche in ambito di concorrenza e sicurezza, tanto che sono paradigmatiche

del modello relazionale. Le transazioni iniziano con un **begin-transaction** (a volte finiscono con *end-transaction*, opzionale) e all'interno deve essere eseguito tra:

- **commit work**, per terminare correttamente la lettura e/o scrittura
- **rollback work**, per abortire la transazione

Un **sistema transazionale OLTP (*OnLine Transaction Processing*)** è in grado di definire ed eseguire transazioni per conto di un certo numero di applicazioni concorrenti anche alto.

Esempio 2. *Vediamo un esempio di transazione (esempio di addebito su un conto corrente e accredito su un altro):*

```
start transaction;
update ContoCorrente
  set Saldo = Saldo + 10 where
  NumConto = 12202;
update ContoCorrente
  set Saldo = Saldo - 10 where
  NumConto = 42177;
commit work;
```

oppure, con anche la verifica che ci siano ancora soldi dopo il prelievo (con eventuale aborto):

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

Il controllo può essere fatto a posteriori grazie al rollback che permette di "dimenticare" tutte le operazioni precedenti

Le istruzioni commit work e rollback work possono comparire più volte all'interno del programma ma esattamente una delle due deve essere eseguita. Si ha un **approccio binario**.
Bisogna approfondire quindi le **unità di elaborazione** che hanno le proprietà cosiddette *ACID*:

- Atomicità, ovvero una transazione è un'unità atomica di elaborazione. Non si può lasciare il db in uno "stato intermedio". Un problema prima del commit cancella tutto le operazioni svolte (*UNDO*) e un problema dopo il commit non deve avere conseguenze, se necessario vanno ripetute le operazioni (*REDO*)

- Consistenza, ovvero la transazione rispetta i vincoli di integrità (se lo stato iniziale è corretto lo è anche quello finale). Quindi se ci sono violazioni non devono restare alla fine (nel caso *rollback*)

- Isolamento, ovvero la transazione non risente delle altre transazioni concorrenti. Una transazione non espone i suoi stati intermedi evitando l'*effetto domino* (si evita che il rollback di una transazione vada in cascata con le altre). L'esecuzione concorrente di una collezione di transazioni deve produrre un risultato che si potrebbe ottenere con una esecuzione sequenziale

- Durata (ovvero persistenza), ovvero gli effetti di una transazione andata in commit non vanno persi anche in presenza di guasti (a tal fine si sfrutta il **recovery manager**, che garantisce l'affidabilità, del DBMS)

2.3 Gestore della concorrenza

Il **gestore della concorrenza** permette di eseguire in parallelo più operazioni.
Definiamo **schedule** come una sequenza di esecuzione di un insieme di transazioni. Uno schedule è **seriale** se se una transazione termina prima che la successiva iniziale, altrimenti è **non seriale**. Qualora non sia seriale si potrebbero avere problemi.
Si sfrutta quindi la **proprietà di isolamento** facendo in modo che ogni transazione esegua come se non ci fosse concorrenza: *un insieme di transazioni eseguite concorrentemente produce lo stesso risultato che produrrebbe una (qualsiasi) delle possibili esecuzioni sequenziali delle stesse transazioni allora si ha la proprietà di isolamento.*

Si ha quindi che uno schedule è serializzabile se l'esito della sua esecuzione è lo stesso che si avrebbe con una qualsiasi sequenza seriale delle transazioni contenute.
Si hanno quindi diversi algoritmi per il controllo della concorrenza secondo varie tipologie:

- controllo basato su *conflict equivalence*
- controllo di concorrenza basato su *locks (protocollo 2PL o two phase locking, shared locks e gestione dei deadlock*). Il protocollo 2PL è usato nei DBMS dove per costruzione si hanno schedule serializzabili usando i lock per bloccare l'accesso alla risorse da parte di una transazione fino a che una risorsa non sia rilasciata. Si hanno quindi i concetti di *lock* e *unlock* che garantiscono l'uso esclusivo di una risorsa e l'autorizzazione esclusiva dell'uso di una risorsa viene dato dal gestore delle transazioni. Si hanno delle **tabelle di lock**. Si ha che, in ogni transazione, tutte le richieste di *lock* precedono tutti gli *unlock* (che comunque devono essere fatti dopo l'operazione di *commit*)
- controllo di concorrenza basato su *timestamps*

Capitolo 3

Sistemi distribuiti relazionali

Abbiamo visto nei sistemi centralizzati come ci fosse una sola base dati. In un sistema distribuito abbiamo diversi **basi dati locali**, diverse applicazioni su ogni nodo di elaborazione (dove ogni nodo condivide varie informazioni) con gli utenti che accedono alle varie applicazioni. Questo tipo di architettura prende il nome di **architettura shared nothing**, in quanto i DBMS di ogni singola macchina sono autonome (anche di vendor diversi) ma che lavorano insieme.
Un sistema distribuito permette non solo di avere dati "distribuiti" tra vari nodi ma anche di "duplicarne" alcuni per diversi scopi, coi nodi collegati in rete (addirittura si hanno soluzioni interamente distribuite nel cloud).
Confrontando un db distribuito con un multi-database (ovvero vari database completi da "unificare") notiamo come entrambi abbiano un'alta distribuzione, il primo una bassa eterogeneità (a differenza del secondo, dove nei vari db potrei avere forte differenza di tipologia dei dati contenuti). Si ha anche bassa autonomia nel caso si db distribuiti a differenza del multi-database (dove ogni db è singolarmente autonomo).
Bisogna capire cosa distribuire. Si hanno diverse condizioni (che possono essere presenti simultaneamente):

- le applicazioni, fra loro cooperanti, risiedono su più nodi elaborativi **(elaborazione distribuita**)
- l'archivio informativo è distribuito su più nodi (**base di dati distribuita**)

La distribuzione si dice essere **ortogonale e trasparente** agli altri.
Capire cosa distribuire è una parte consistente dello studio di come costruire un'architettura distribuita (magari frutto di situazioni particolari come la "fusione" di due sistemi a causa di un'acquisizione aziendale etc... dove

diverse logiche applicative e diverse strutture dati possono creare situazioni molto pericolose).
Possiamo classificare i db distribuiti. Si ha innanzitutto che un **DBMS Distribuito Eterogeneo Autonomo** e' in generale una federazione di DBMS che collaborano nel fornire servizi di accesso ai dati con livelli di *trasparenza* definiti (infatti le diversità tra db nei nodi vengono "nascosti" a vari *livelli di trasparenza* per distribuzione, eterogeneità e autonomia). Come abbiamo visto esiste l'esigenza di integrare a posteriori vari db preesistenti (anche a causa di integrazione di nuovi applicativi o nuove cooperazioni di processi) e questa situazione è spinta dallo sviluppo della rete.
Possiamo quindi dividere i livello di federazione su tre categorie tra loro ortogonali (ovvero indipendenti):

- autonomia
- distribuzione
- eterogeneità

L'**autonomia** fa riferimento al grado di indipendenza tra i nodi e si hanno diverse forme:

- **autonomia di progetto**, il livello "massimo" dove ogni nodo ha un proprio modello dei dati e di gestione delle transazioni
- **autonomia di condivisione**, dove ogni nodo sceglie la porzione di dati da condividere ma condividendo con gli altri nodi lo schema comune
- **autonomia di esecuzione**, dove ogni nodo sceglie in che modo eseguire le transazioni

Si hanno quindi:

- **DBMS Strettamente integrati** con nessuna autonomia, con dati logicamente centralizzati, un unico data manager per le transazioni applicative e vari data manager locali che non operano in modo autonomo ma eseguono le direttive centrali
- **DBMS semi-autonomi**, dove ogni data manager è autonomo ma partecipa a transazioni globali, dove una parte dei dati è condivisa e dove sono richieste modifiche architetturali per poter fare parte della federazione

- **DBMS Peer to Peer** completamente autonomi, dove ogni DBMS lavora in completa autonomia ed è inconsapevole dell'esistenza degli altri

Per la **distribuzione** dei dati si hanno 3 livelli classici:

- **distribuzione client/server**, in cui la gestione dei dati è concentrata nei server, mentre i client forniscono l'ambiente applicativo e la presentazione
- **distribuzione Peer to Peer**, in cui non c'è distinzione tra client e server, e tutti i nodi del sistema hanno identiche funzionalita' DBMS
- **nessuna distribuzione**

Le prime due possono anche non essere distinte.
L'**eterogeneità** può invece riguardare vari aspetti:

- **modello dei dati** (relazionale, XML, object oriented (OO), json)
- **linguaggio di query** (diversi dialetti SQL, query by example, linguaggi di interrogazione OO o XML)
- **gestione delle transazione** (protocolli diversi per il gestore della concorrenza o per il recovery)
- **schema concettuale e logico** (concetti rappresentati in uno schema come attributo e in altri come entità)

Quindi si hanno:

- **DBMS distribuito omogeneo (DDBMS)** quando si ha alta distribuzione ma non si hanno autonomia ed eterogeneità (gestiti solitamente dallo stesso vendor)
- **DBMS eterogeneo logicamente integrato (data warehouse)** quando si ha alta eterogeneità ma non si hanno distribuzione e autonomia
- **DBMS distribuiti eterogenei** quando si ha alta eterogeneità e distribuzione ma non autonomia
- **DBMS federati distribuiti** quando si ha alta distribuzione, semi autonomia e non eterogeneità

- **DBMS distribuiti federati eterogenei** quando si ha alta distribuzione ed eterogeneità e semi autonomia
- **multi db MS**, totalmente autonomi ed eventualmente omogenei o eterogenei

Si hanno molti altri sistemi in base alle 3 categorie.

3.1 DDBMS

Parliamo di **DBMS distribuito omogeneo (*DDBMS*)**.
Studiamo uno schema in cui si passa da un sistema centralizzato ad un sistema distribuito.
Si hanno due architetture di riferimento:

- l'**architettura dati**
- l'**architettura funzionale**, ovvero l'insieme di tecnologie a supporto dell'architettura dati

Non avendo eterogeneità mantengo lo stesso schema di un DBMS centralizzato ma distribuisco dati bisogna prendere lo schema centralizzato e aggiungere componenti tra lo schema logico e lo schema fisico. Infatti non si avrà più un solo schema logico e un unisco schema fisico ma tanti schemi logici e fisici locali (ad ogni logico corrisponde un fisico). I vari schemi logici inoltre si interfacciano con uno **schema logico globale**, i vari schemi logici locali non sono quindi altro che delle *viste* dello schema logico globale. Questa organizzazione tra schemi logici locali e schema logico globale è la cosiddetta **organizzazione LAV (*Local As View*)**. In ogni caso il progettista interroga lo schema logico globale e saranno varie tecnologie ad interrogare gli schemi logici locali (si fa una sorta di routing delle query).
Per ciascuna funzione (come query processing, transaction manager etc...) si possono avere vari tipi di gestione:

- centralizzata/gerarchica o distribuita
- con assegnazione statica o dinamica dei ruoli

Lo schema globale viene progettato prima degli schemi locali.
Ovviamente cambia il **processo di progettazione** nel caso dei DDBMS. Normalmente si ha un approccio *top-down* per la progettazione, con:

1. analisi dei requisiti

2. progettazione concettuale
3. progettazione logica
4. progettazione fisica

ma questo tipo di progettazione va cambiato e quindi si introduce una nuova fase e si cambiano le ultime due:

1. analisi dei requisiti
2. progettazione concettuale
3. **progettazione della distribuzione**, per capire dove mettere i dati
4. progettazione logica **locale**, che traduce dallo schema concettuale globale allo schema logico locale solo alcuni concetti
5. progettazione fisica **locale**

Si introduce il concetto di **portabilità**, ovvero la capacita' di eseguire le stesse applicazioni DB su ambienti runtime diversi (anche con SQL diversi e differenti dallo standard). La portabilità è a *compile-time*
Si ha anche il concetto di **interoperabilità** (tra vendors diversi), ovvero la capacita' di eseguire applicazioni che coinvolgono contemporaneamente sistemi diversi ed eterogenei (con zero autonomia). A tal fine sono stati introdotti dei *middleware*, tra cui **ODBC** che si occupa dell'accesso a dati di diversi vendor. ODBC, a livello architetturale, si pone sopra il DBMS e da un'immagine indipendente da ciò che c'è sotto (funziona come una sorta di *driver*), trasformando tutto in una sorta di SQL standard. Si hanno anche dei protocolli, come **X-Open Distributed Transaction Processing (*DTP*)**, che consentono di eseguire delle transazioni secondo una logica diversa. Questo protocollo stabilisce una serie di API che vengono implementate da ogni singolo DBMS per offrire una connettività standard (approccio molto usato per transazioni con vendor diversi). Il protocollo funziona sia se si ha che fare con omogeneità che con eterogeneità.
Si hanno altri approcci:

- **basi dati parallele**, con incremento delle prestazione mediante parallelismo sia di storage devices che di processore (scalabilità orizzontale). Un esempio sono le **basi dati GRID**
- **basi dati replicate** dove si ha la replicazione della stessa informazione su diversi server per motivi di performance. Importanti per i temi della consistenza e della sicurezza

- **Data warehouses**, ovvero DBMS centralizzati, risultato dell'integrazione di fonti eterogenee, dedicati nel dettaglio alla gestione di dati per il supporto alle decisioni. Prevede la *cristallizzazione* dei dati, acquisiti da varie sorgenti, creando un nuovo schema con la memorizzazione dei dati in formato nuovo (solitamente relazionale). Non usa un approccio LAV