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.

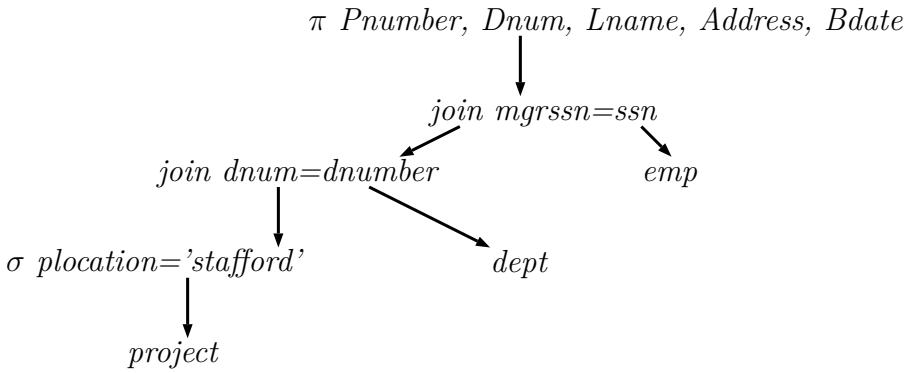
2.1 Ottimizzazione delle query

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** è 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

Autonomia

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

Distribuzione

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 funzionalità DBMS
- **nessuna distribuzione**

Le prime due possono anche non essere distinte.

Eterogeneità

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 vari tipi di DBMS:

- **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 capacità 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 capacità 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

3.1.1 Caratteristiche dei DDBMS

Si hanno vari tipi di architetture DDBMS:

- **shared-everything**, ad esempio *SMP server*, dove il db management system e il disco sono in un unico nodo
- **shared-disk**, ad esempio *Oracle RAC*, dove diversi db management systems agiscono su una stessa **SAN (*Storage Area Network*)**, ovvero un'architettura dati di puro storage (con tanti dischi in raid). I vari db accedono ai dati secondo una certa regolazione. Viene distribuito il carico sui db ma si hanno problemi di concorrenza e hanno grandi problemi di scalabilità e costo economico
- **shared-nothing**, sempre più usati, dove ogni db management system ha il suo disco. È molto scalabile e, a patto di gestire la complessità, posso aggiungere nodi in modo illimitato (**scalabilità orizzontale**). Si presta molto all'ambiente cloud. Sono *architetture federate*.

Vediamo quindi le proprietà generali di un DDBMS (facendo esplicito riferimento alle architetture *shared-nothing* per la loro scalabilità):

- **località**, secondo il *principio di località*, che garantisce un aumento di performances (nonché di sicurezza) tenendo i dati si trovano "vicino" alle applicazioni che li utilizzano più frequentemente

- **modularità**, permettendo di scalare orizzontalmente e permettendo modifiche a dati ed applicazioni a basso costo
- **resistenza ai guasti**
- **prestazioni ed efficienza**

Concentrandoci sulla **località** si ha che la partizione dei dati corrisponde spesso ad una partizione naturale delle applicazioni e degli utenti. I dati risiedono più vicino a dove vengono più usati ma possono comunque essere raggiunti anche da lontano (*globalmente*). Si cerca inoltre sempre di più di spostare i dati verso le applicazioni (paradigma ribaltato nel caso di *big data*).
In merito alla **modularità** si nota come la distribuzione dinamica dei dati si adatta meglio alle esigenze delle applicazioni (magari spostando solo sottotabelle verso alcuni nodi etc. . ., sia in modo trasparente rispetto all'utente che altrimenti).
Parlando di **resistenza ai guasti** si ha una maggior fragilità a causa delle unità che aumentano di numero ma si ha **ridondanza** e quindi maggiore resistenza ai guasti di dati e applicazioni ridondate (*fail soft*).
Discorso più interessante è da farsi sulle **prestazioni**. Ogni nodo in un sistema shared-nothing gestisce db di dimensioni ridotte. Inoltre ogni nodo può essere ottimizzato ad hoc ed è più semplice gestire e ottimizzare applicazioni locali. Si ha inoltre distribuzione del carico totale e parallelismo tra transazioni locali che fanno parte di una stessa transazione distribuita (anche se questo aspetta obbliga soluzioni di coordinamento e appesantisce il carico sulla rete, che rischia di diventare un "collo di bottiglia").
I DDBMS hanno ovviamente **funzionalità** specifiche.
Ogni server ha buona capacità di gestire transazioni indipendentemente, anche se le interazione distribuita tra server rappresentata un carico supplementare. Per le interrogazioni si ha che le query arrivano dalle applicazioni e i risultati dai server mentre per le transazioni le richieste transazionali arrivano dalle applicazioni ma sono richiesti **dati di controllo** per il coordinamento.
' La gestione della rete deve essere ottimizzata e serve uno studio sulla distribuzione locale dei dati.
Ricapitolando si hanno le seguenti funzionalità specifiche:

- **trasmissione** di query, transizioni, frammenti di db e dati di controllo tra i nodi
- **frammentazione, replicazione e trasparenza** (secondo vari livelli), fattori legati alla natura distribuita dei dati

- un **query processor** e un **query plan** per la previsione di una strategia globale accanto a strategie per le query locali. Si gestisce il passaggio tra schema logico globale e quelli locali. Chi esegue la query lo fa senza pensare alla frammentazione dei dati
- **controllo di concorrenza** tramite algoritmi distribuiti, fondamentale per gli accessi *in scrittura*
- **strategie di recovery** e **gestione dei guasti**, sia in merito alla rete che all'hardware stesso

3.1.2 Frammentazione e replicazione

Si definisce **frammentazione** come la possibilità di allocare porzioni (*chunk*) diverse del db su nodi diversi.
Si definisce **replicazione** come la possibilità di allocare stesse porzioni del db su nodi diversi.
Si definisce **trasparenza** come la possibilità per l'applicazione di accedere ai dati senza sapere dove sono allocati (serve qualcosa che instradi le query).

frammentazione

Esistono due tipi di frammentazione:

1. **frammentazione orizzontale**, che prevede di prendere una tabella e frammentare in base alle righe (le prime n da una parte, le seconde m dall'altra etc...). Si mantiene quindi inalterato lo schema in quanto ottengo solamente delle tabelle più piccole in quanto pezzi. Per spezzare uso una *select* (per la **selezione**) che selezioni ogni volta un certo "blocco" di tabella
2. **frammentazione verticale**, che consente di ridurre la dimensionalità della tabelle spezzandola in base alle colonne. In ogni nuova tabella però la prima colonna deve essere uguale alla prima della tabella originale (ovvero dove si ha la chiave primaria), questo per garantire che si possa ricomporre la tabella (e lo schema) originale (con operazioni di *join*, o meglio un *natural join*) e garantire la trasparenza. Anche in questo caso uso una *select* (per la **proiezione**) che selezioni ogni volta un certo numero di colonne da mettere nella nuova tabella

Bisogna quindi garantire:

- **completezza**, ovvero ogni record della relazione R di partenza deve poter essere ritrovato in almeno uno dei frammenti
- **ricostruibilità**, ovvero la relazione R di partenza deve poter essere ricostruita senza perdita di informazione a partire dai frammenti
- **disgiunzione**, ovvero ogni record della relazione R deve essere rappresentato in uno solo dei frammenti
- **replicazione**, l'opposto della disgiunzione

Quindi possiamo definire meglio le proprietà dei due tipi di frammentazione per la relazione R, frammentata in diversi R_i:

1. **orizzontale**:
   - $schema(R_i) = schema(R)$, $\forall i$
   - ogni R_i contiene un sottoinsieme dei record di R
   - è definita da una proiezione su una condizione ci: $\sigma_{ci}(R)$
   - garantisce la completezza, infatti $R_1 \cup R_2 \cup \dots R_n = R$
   - l'unione garantisce la ricostruibilità
2. **verticale**:
   - $schema(R) = L = (A_1, \dots, A_m)$ e $schema(R_i) = L_i = (A_{i1}, \dots, A_{ik})$
   - garantisce la completezza, infatti $L_1 \cup L_2 \cup \dots L_n = L$, dove i vari L_i sono i frammenti verticali ed L è la tabella originale
   - si garantisce la ricostruibilità in quanto $L_i \cap L_j \supseteq chiave\ primaria(R)$, $\forall i \neq j$ (ovvero ogni frammento deve contenere la chiave primaria)

Replicazione

Approfondiamo ora la **replicazione**. Si hanno diversi aspetti positivi per l'accesso *in lettura*, come il miglioramento delle prestazioni in quanto consente la coesistenza di applicazioni con requisiti operazionali diversi sugli stessi dati e aumenta la *località dei dati* usati da ogni applicazioni. Nel momento in cui si ha l'accesso *in scrittura* si hanno però diversi aspetti negativi. Si hanno diverse complicazioni architetturali, tra cui la gestione della transazioni e

l'updates di copie multiple, che devono essere tutte aggiornate. Inoltre bisogna studiare dal punto di vista progettuale cosa replicare, quanto replicare (ovvero capire quante copie mantenere), dove allocare le copie e le politiche per gestirle.
In merito all'allocazione studiamo anche gli **schemi di allocazione**. Ogni frammento può essere allocato su un nodo diverso. Lo schema globale quindi è solo *virtuale* (in quanto non materializzato in un solo nodo) e lo **schema di allocazione** definisce il *mapping* tra un frammento e un nodo. Si ha quindi una tabella, un **catalogo**, che ci da informazioni sul partizionamento, associando ogni frammento al nodo in cui è allocato.

Trasparenza

Con la **trasparenza** si ha la separazione della semantica di alto livello dalle modalità di frammentazione e allocazione. Si separa quindi la *logica applicativa* dalla *logica dei dati* ma per farlo serve uno strato software che gestisca la traduzione dallo schema unico ai sottoschemi, comportando un aumento di complessità del sistema e una perdita di prestazioni (problemi che si riducono con un *mapping* integrato del DDBMS).
Le applicazioni (transazioni, interrogazioni) non devono essere modificate a seguito di cambiamenti nella definizione e organizzazione dei dati e si hanno due tipi di trasparenza, che si applicano agli schemi ANSI-SPARC nel modello distribuito (schema logico globale e schemi logici/fisici locali):

1. **trasparenza logica (o indipendenza logica)**, ovvero in dipendenza dell'applicazione da modifiche dello schema logico. Un'applicazione che usa un frammento non viene modificata se vengono modificati altri frammenti

2. **trasparenza fisica (o indipendenza fisica)**, ovvero in dipendenza dell'applicazione da modifiche dello schema fisico

Frammentazione e allocazione sono tra lo schema logico globale e ogni schema logico locale.
Si hanno quindi tre livelli di trasparenza:

- **trasparenza di frammentazione**, che permette di ignorare l'esistenza dei frammenti ed è lo scenario migliore per la programmazione applicativa con un'applicazione scritta in SQL standard. Il sistema si occupa di convertire query globali in locali e relazioni in sotto-relazioni. La scomposizione delle query per ogni sotto-relazione è detta **query rewriting**

- **trasparenza di replicazione/allocazione**, dove l'applicazione è consapevole dei frammenti ma non dei nodi in cui si trovano. In questo caso la query è già spezzata in quanto si sa di avere a che fare con un sistema frammentato
- **trasparenza di linguaggio**, dove l'applicazione specifica sia i frammenti che i nodi, nodi che possono offrire interfacce che non sono SQL standard. Tuttavia l'applicazione sarà scritta in SQL standard a prescindere dai linguaggi locali dei nodi. Le query vengono quindi tradotte ottimizzatone di query. *Questo è il livello di trasparenza più basso*

3.2 Query distribuite

Analizzeremo prevalentemente DDBMS distribuiti *shared-nothing* e, in seguito, *architetture di replica* (con un *replication server* atto a gestire al replica). Le query sono ovviamente le operazioni più importanti. Possono essere di sola *lettura* (tramite operazioni come la *select*) o anche di *scrittura*. Le due tipologie di operazioni vengono gestite in modo molto differente (la lettura sincrona non è un problema, se non hardware risolvibile con una distribuzione del carico, a differenza della scrittura sincrona). Le operazioni devono essere eseguite **velocemente**.

3.2.1 Query in lettura

Esistono, in un sistema relazionale, una serie di attività che convertono la query in SQL in algebra relazionale e solo dopo si ha la distribuzione. L'utente, ignaro dello schema distribuito, interroga lo schema logico globale e il DDBMS decompone la query secondo una localizzazione specifica in base ai singoli frammenti (ovvero deve distribuire la query in modo sensato). Si ha anche un'ottimizzazione globale della query prima della distribuzione in modo che anche la distribuzione stessa sia ottimizzabile correttamente, infatti il gestore delle interrogazioni manda ai singoli nodi i giusti frammenti di query che verranno ottimizzati localmente. Ho quindi nel complesso 4 fasi che compongono il **query processor**:

1. **query decomposition**
2. **data localization**
3. **global query optimization**
4. **local optimization**

Query decomposition

La *query decomposition* opera sullo schema logico globale non tenendo conto della distribuzione. In questo caso si hanno tecniche di ottimizzazione algebrica (usando quindi l'algebra relazionale indipendentemente dalla distribuzione) analoghe a quelle usati in sistemi centralizzati e si ha come output un **query tree** non ottimizzato rispetto ai **costi di comunicazione**. Il costo di comunicazione riguarda il costo di uso della **rete** e dipende da vari fattori. Il costo di comunicazione è il vero "collo di bottiglia" in sistemi distribuiti.

Data localization

La *data localization* considera la frammentazione delle tabelle e la distribuzione, capendo ad esempio dove effettuare le *select* etc.... Si procede quindi all'ottimizzazione delle operazioni rispetto alla frammentazione, tramite **tecniche di riduzione**. Viene quindi prodotta una query efficiente per la frammentazione ma non ottimizzata. Supponiamo per esempio di avere una tabella su 3 nodi (distribuita tramite frammentazione orizzontalmente) e che di base la query faccia la richiesta a tutti e tre (facendo l'unione dei risultati). Usando la tecnica di riduzione, qualora, per esempio, effettivamente sia necessaria solo in un nodo, si avrà che la query sarà distribuita unicamente nel nodo corretto.

Global query optimization

La *global query optimization* si basa sulle statistiche sui frammenti per effettuare l'ottimizzazione. Viene arricchito il **query tree**, creato con gli operatori dell'algebra relazionale, tramite gli **operatori di comunicazione** (ovvero *send* e *receive*), che vengono effettuati tra nodi. Alcune query, dopo aver tenuto conto dei tempi di comunicazione, potranno essere eseguite in parallelo (grazie all'indipendenza data dallo *shared-nothing*). In questo caso le decisione più rilevanti riguardano le operazioni di *join* (che è uno degli operatori più complicati) e,in particolare (come vedremo più avanti), l'ordine tra i *join* n-ari e la scelta tra *join* e *semijoin* (un operatore particolare per i sistemi distribuiti). L'operatore di *join* infatti "ingrandisce" i dati "fondendo" tabelle, che magari sono frammentate in più tabelle su vari nodi. L'uso di *send* e *receive* permette la comunicazione dei dati tra i nodi, anche se questo rischia di diventare troppo esoso in termini di prestazioni. Si hanno quindi degli **algoritmi di calcolo del costo adattivi** e si deve studiare la rete e i suoi ritardi, che dipendono dalla *topologia* della rete stessa e dal carico applicativo. Si ha quindi una fase di **ottimizzazione a runtime**, dove si riadatta il **query plan**. Per riadattarlo si fa in primis monitoring sull'esecuzione della

query, si procede adattando il modello di costo (eventualmente con software automatici) e, eventualmente, riadattando la query se si calcola uno scarto di costo troppo elevato. È il DBA che stabilisce delle soglie temporali entro le quali ottenere una risposta. Per questo conta la trasparenza, in quanto non è l'applicazione che deve interessarsi di questo aspetto. Si introduce un nuovo *layer* di complessità.
In alcuni casi è impossibile ad avere un DDBMS che si occupi di questo tipo di ottimizzazioni.
La distribuzione non è predicibile a priori. Vediamo un semplice esempio:

Esempio 3. *Si supponga di avere il seguente schema:*

- *Employee (eno, ename, title), di cardinalità 400*
- *AssiGN(eno, projectno, resp, dur), di cardinalità 1000 e dove* resp *rappresentata il tipo di responsabilità*

Si ha la seguente query: trovare i nomi dei dipendenti che sono anche manager di progetti:

```
SELECT ename
FROM Employee E JOIN AssiGN A on E.eno=A.eno
WHERE resp=''manager''
```

Che, in algebra relazionale già abbastanza ottimizzata ipotizzando che ci siano pochi manager, sarebbe:

$$\pi_{ename}(EMP >\!<_{eno} (\sigma_{resp=''manager''}(ASG)))$$

Supponiamo di avere poi 5 nodi uguali, il quinto per il risultato e i primi 4 frammentati orizzontalmente secondo questo schema di divisione per nodo:

1. $ASG1 = \sigma_{eno} \leq' E3'(ASG)$
2. $ASG2 = \sigma_{eno} >' E3'(ASG)$
3. $EMP1 = \sigma_{eno} \leq' E3'(EMP)$
4. $EMP2 = \sigma_{eno} >' E3'(EMP)$

Vediamo quindi una prima esecuzione:

- *chiedo al nodo 1 i manager e sposto i risultati di* $\sigma_{resp=''manager''}(ASG1)$ *sul nodo 3 (dove sono descritti in modo più completo) come* $(ASG'1)$*. Il risultato di* $EMP1 >\!<_{eno} (ASG1)$*, calcolato sul nodo 4, lo porto sul nodo 5* $EMP'1$

- *chiedo al nodo 2 i manager e sposto i risultati di* $\sigma_{resp=''manager''}(ASG'2)$ *sul nodo 4 (dove sono descritti in modo più completo) come* $(ASG'2)$*. Il risultato di* $EMP2 >_{eno} (ASG'2)$*, calcolato sul nodo 4, lo porto sul nodo 5 come* $EMP'2$
- *sul nodo 5 il risultato sarà* $EMP'1 \cup EMP'2$

Vediamo una seconda soluzione:

- *contemporaneamente chiedo al nodo 1 e la nodo 3 di mandare al nodo 5 tutti i manager. Sempre contemporaneamente a queste due operazioni chiedo al nodo 2 e al nodo 4 di mandare al nodo 5 tutte le informazioni. I tempi saranno basati sul più lento dei quattro nodi, che determinerà il tempo massimo dell'operazione (che sono circa calcolabili a priori tramite la tabella delle statistiche)*
- *nel nodo 5 calcolo il risultato:*

$$(EMP1 \cup EMP2) >_{eno} \sigma_{resp=''manager''}(ASG1 \cup ASG2)$$

I tempi di trasporto detteranno quale soluzione tra le due è la più performante ma, viste le cardinalità esigue di dati, probabilmente vince la seconda (dove si ha un solo spostamento globale). Questa seconda strategia costringe a pensare a particolari strutture di accesso secondarie dette ***indici****, che permettono interrogazioni efficaci ma che* ***non possono essere "portati"*** *in sistemi distribuiti. Quindi nel nodo 5 non ho gli* ***indici*** *e quindi devo fare l'intero* ***prodotto cartesiano*** *per il join (che però in questo caso ha un tempo trascurabile, grazie alla bassa cardinalità dei dati, rispetto ai costi di trasferimento, generalmente non trascurabili rispetto ai costi delle operazioni interne ad un nodo).*

Riprendendo l'esempio definiamo:

- **costo di messaggio** come il costo fisso di spedizione o ricezione di un messaggio (detto *setup*)
- **costo di trasmissione** come il costo, fisso rispetto alla topologia, di trasmissione dati
- **costo di comunicazione** come la somma tra il costo di messaggio, moltiplicato per il numero di messaggi, più il costo di trasmissione, moltiplicato per il numero di *bytes* trasmessi
- **costo totale** come la somma dei costi delle operazioni (I/O e CPU) più i costi di comunicazione (*comunicazione*)

- **response time** come la somma dei costi qualora si tenga conto del *parallelismo delle trasmissioni*, quindi come la somma tra il costo di messaggio, moltiplicato per il numero di messaggi comunicati in modo sequenziale, più il costo di trasmissione, moltiplicato per il numero di *bytes* trasmessi in modo sequenziale. In questo conto volendo posso usare dei **pesi** basati sulla cardinalità delle unità da trasferire e tenere conto del massimo tempo di risposta che si ottiene

Si ha quindi che:

- nelle **grandi reti geografiche** i costi di *comunicazione* sono molto maggiori del costo di *I/O*, circa di 10 volte
- nelle **reti locali** i costi di *comunicazione* e *I/O* sono paragonabili, grazie alle reti *gigabit* in locale

Tendenzialmente il costo di comunicazione è ancora il **fattore critico** ma sempre meno.
Bisogna scegliere cosa **minimizzare**:

- il *response time*, aumentando il parallelismo che però può portare ad un aumento del *costo totale*, con un maggior numero di trasmissione e un maggior processing locale. Nell'esempio 3 potrebbe sembrare la seconda soluzione, che effettivamente parallelizza di più ma non minimizza i costi di risposta
- il *costo totale*, senza tener conto del parallelismo utilizzando meglio le risorse e aumentando il *throughput* ma peggiorando così il *response time*. Nell'esempio 3 è la prima soluzione

Join e Semijoin

Il **join** presenta il problema di portare alla perdita dell'**indice**. Bisogna quindi studiare come effettuare l'operazione tra due tabelle su due nodi diversi. Una prima operazione è data dell'operazione di *semijoin*.

Definizione 2. *Definiamo, in algebra relazionale, l'operazione semijoin, tra due tabelle R e S, sull'attributo A, come:*

$$R \ semijoin_A \ S \equiv \pi_{R^*}(R \ join_A \ A)$$

dove R^* è l'insieme degli attributi di R.
In altre parole scelgo esplicitamente di tenere solo gli attributi di R dopo il

semijoin.
Quindi con R $semijoin_A$ S ho la proiezione sugli attributi di R operazione di join e quindi ho che il semijoin non è ***commutativo****.*
Dalla seconda tabella porto solo la serie di attributi che mi servono esplicitamente ($\pi_A(S)$) riducendo il carico di lavoro.
Alla fine il nostro R' ***con i risultati del semijoin sarà trasportato nel nodo di*** S

Prese due tabelle allocate su nodi differenti, il *join* tra di esse può quindi essere calcolato tramite operazioni di *semijoin*, valgono infatti le seguenti equivalenze (che portano a diverse strategie a seconda della stima dei costi):

- $R \ join_\theta \ S \iff (R \ semijoin_\theta \ S) \ join_\theta \ S$
- $R \ join_\theta \ S \iff R \ join_\theta \ (S \ semijoin_\theta \ R)$
- $R \ join_\theta \ S \iff (R \ semijoin_\theta \ S) \ join_\theta \ (A \ semijoin_\theta \ R)$

In tutti i casi si riduce lo spostamento dei dati.
L'uso del *semijoin* è conveniente sse il costo del suo calcolo e del trasferimento del risultato è inferiore al costo del trasferimento dell'intera relazione e del costo dell'intero *join* (e questo dipende dal numero di attributi coinvolti).
Avere più di un join complica la situazione, anche solo per la scelta dell'ordine in cui eseguirli.

Local optimization

La *local optimization* si occupa dell'ottimizzazione degli schemi locali. Ogni nodo riceve una *fragment query* e la ottimizza, con tecniche analoghe ai sistemi centralizzati, in modo completamente indipendente. Si hanno comunque operazioni di ottimizzazione locale a priori sul fatto che il *global query optimization* punti a ridurre i costi di comunicazione (nel caso di un DDBMS in rete geografica) o ad aumentare il parallelismo (in caso di DDBMS in rete locale).
In ogni caso nella progettazione di sistemi di gestione dati distribuiti bisogna tener conto di:

- tipologie di query distribuite
- stime o statistiche sullo storico query distribuite già eseguite (fatto periodicamente dal DDBMS)
- topologia della rete
- carico aspettato e workload previsto

3.2.2 Query in scrittura e controllo di concorrenza