

Appunti di Architetture Dati

A cura di:
Francesco Refolli
Matricola 865955

Anno Accademico 2023-2024

Chapter 1

Introduzione

Il corso approfondisce le architetture dati nel senso di DBMS:

- Progettare l'architettura dati piu' adatta valutando degli specifici use case
- Comprendere i meccanismi di gestione moderni dei DBMS
- I Modelli di rappresentazione dei dati alternativi ai modelli relazionali
- DBMS principali e i loro principi

In particolare durante il corso si approfondiranno i seguenti use case, che sono articolati in parti nel documento presente:

- **Ambito bancario:** si vuole realizzare una architettura dati per consentire le normali operazioni bancarie come i bonifici
- **Machine learning:** si vuole realizzare una pipeline per la costruzione dei dataset di training per addestrare modelli di Machine/deep learning e le relative pipeline per l'applicazione dei modelli addestrati sui dati in esecuzione
- **Modelli e Architetture non relazionali:** si vuole gestire dati che non sono facilmente modellabili nel modello relazionale o che non possono essere gestiti dai sistemi distribuiti relazionali
- **Cloud:** Si vuole utilizzare sistemi di gestione dei dati in cloud
- **AI generativa per la gestione dei dati:** si vuole capire come utilizzare l'AI generativa nei sistemi di gestione dei dati

L'esame e' uno scritto con Domande a risposte aperte sulla teoria/esercizi di progettazione di architetture o sistemi di gestione dati/esercizi sui linguaggi di interrogazioni NoSQL.

In alternativa è possibile realizzare un elaborato che approfondisca un particolare tema / use cases mostrato a lezione, da concordare con il docente.

Part I

Use Case Bancario

Chapter 2

Use Case Distribuito Relazionale

2.1 Introduzione

L'architettura dati deve essere in grado di gestire le operazioni bancarie, quindi transazioni e bonifici tra piu' banche.

Il workload e' misto, si avranno milioni di letture e scritture ogni giorno, di carattere semplice. Il volume di dati e' grande e il sistema e' eterogeneo. Infatti non solo le banche possono usare sistemi diversi, ma quando avvengono acquisizioni e fusioni spesso e volentieri accade che i sistemi delle sottobanche siano mantenuti e messi in parallelo.

Le metriche di qualita' sono il numero di operazioni che e' in grado di soddisfare e la latenza di esse.

Un Sistema Centralizzato va bene? Necessariamente no, non solo perche' le banche sono piu' di una e potenzialmente distanti tra loro ma anche perche' il volume di operazioni e' tale da richiedere un sistema distribuito.

Sistemi Distribuiti Nella progettazione di un sistema distribuito il problema principale e' rappresentato dal capire cosa e come distribuire le risorse e l'elaborazione. Nella progettazione di architetture dati ci si pongono gli stessi questi. Infatti l'elaborazione puo' essere su piu' nodi cosi' come lo storage dei dati (DDBMS). La distribuzione in un livello e' comunque ortogonale e trasparente agli altri livelli (Layered Architecture).

Particolari sono i casi di merger & acquisition perche' richiedono un problema tecnologico ulteriore: capire come unire logiche applicative e sistemi di archiviazione.

2.2 Un Esempio

Si consideri la base di dati di una azienda con progetti e vendite. Questa ha piu' sedi: Milano, Piacenza, Bologna.



Figure 2.1:

Una base di dati centralizzata avr  dei nodi elaborativi si' anche a Piacenza e a Bologna, ma l'archiviazione risieder  solo a Milano. Questo introduce diversi problemi, primo di tutti la rete: infatti una congestione della rete puo' rendere impossibile alle altre due sedi di acquisire i dati necessari all'elaborazione.

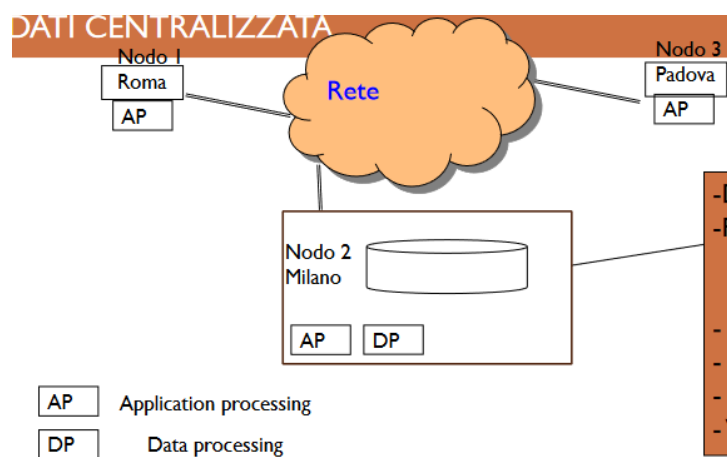


Figure 2.2:

Quindi e' utile dividere il database in frammenti per poter avere i dati vicini il piu' possibile al luogo di utilizzo. Dopo di che possiamo anche immaginare di mantenere delle copie di alcuni frammenti anche su altri nodi se sono necessari all'elaborazione oppure per mantenere un po' di ridondanza.

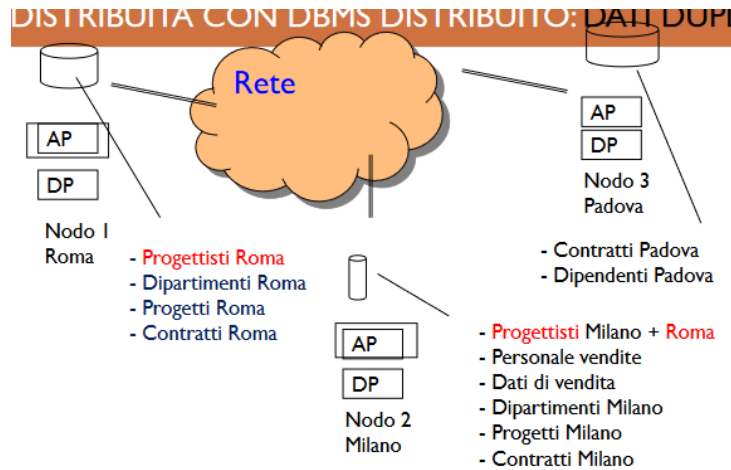


Figure 2.3:

2.3 Modelli Architetture per DBMS

Un DBMS Distribuito Eterogeneo Autonomo è una federazione di DBMS che collaborano nel fornire servizi di accesso ai dati con livelli di trasparenza definiti.

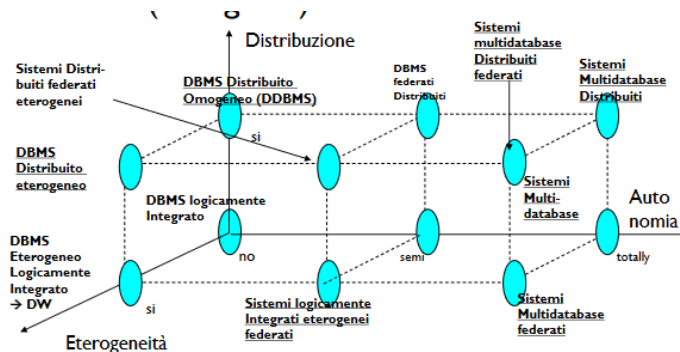


Figure 2.4:

Per trasparenza si intende la proprietà di nascondere le diversità tra basi di dati nei nodi del sistema riguardo alla distribuzione, all'eterogeneità e all'autonomia dei nodi.

2.3.1 Autonomia

Per autonomia si intende il grado di indipendenza dei nodi che può assumere varie forme:

- **Di progetto:** ogni nodo adotta un proprio modello dei dati e sistema di gestione delle transazioni
- **Di condivisione:** ogni nodo sceglie la porzione di dati che intende condividere con altri nodi
- **Di esecuzione:** ogni nodo decide in che modo eseguire le transazioni che gli vengono sottoposte

In DBMS integrati i dati sono logicamente centralizzati e abbiamo un unico data manager che esegue le transazioni applicative ed orchestra i data manager locali.

Si parla di DBMS semi autonomi quando ogni data manager e' autonomo ma partecipa a transazioni globali, una parte dei dati e' condivisa e si richiedono modifiche architettruali per formare una federazione.

Invece esistono DBMS totalmente autonomi dove ognuno lavora in completa autonomia ed e' inconsapevole dell'esistenza di altri nodi.

2.3.2 Distribuzione

La Distribuzione fa riferimento alla distribuzione dei dati, dove possiamo avere architetture client/server con i dati nel server o distribuzioni peer-to-peer in cui non c'e' distinzione tra client/server e tutti i nodi del sistema sono DBMS equipollenti.

2.3.3 Eterogeneita'

Questa riguarda o il modello dei dati (nominalmente relazionale o no), il linguaggio di query, il sistema delle transazioni o lo schema concettuale/logico della base di dati.

2.4 Progettazione di un DBMS Autonomo (DBMSA)

Noi mettiamo il focus sullo schema "Shared Nothing", ovvero i nodi sono completamente slegati da legami infrastrutturali come possono essere invece lo schema Oracle RAC o lo Shared-Everything.

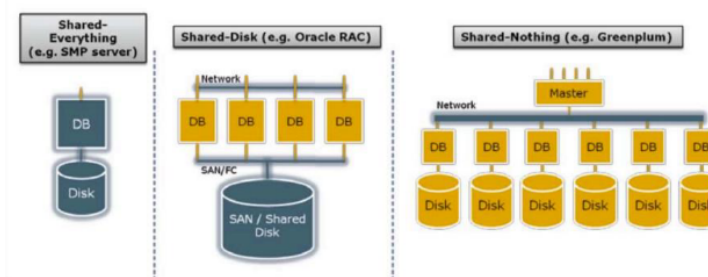


Figure 2.5: Architettura Funzionale di un DBMS

Per ciascuna funzione di un DBMS nei suoi nodi possiamo avere una gestione centralizzata/gerarchica o distribuita con assegnazione statica o dinamica dei ruoli (importante per la ripresa da shock).

Nella progettazione di un DBMS Distribuito si introduce la fase di progettazione della distribuzione successiva alla progettazione concettuale e le progettazioni logico/fisica diventano locali ai nodi.

Per quanto riguarda Portabilita' e Interoperabilita' dei DBMS DEA queste sono facilitate dalla standardizzazione di protocolli e linguaggi ma comunque possono avere dei problemi (si pensi ai diversi dialetti adottati da alcuni DBMS ...).

2.4.1 Vantaggi

Disporre di nodi di DBMS locali permette di avvicinare i dati alle applicazioni rendendo i tempi di accesso tendenzialmente inferiori e la modularita' che ne consegue da la possibilita' di effettuare modifiche a basso costo (locali ai nodi). Inoltre e' resistente ai guasti in quando solo il nodo locale rotto sara' incapacitato, non tutto l'apparato.

La distribuzione dei dati generalmente riguarda la distizione tra **Utenti Locali** e **Utenti Globali** e puo' essere incrementale e progressiva, molto flessibile. Inoltre la possibilita' di avere ridondanza riduce i danni da guasto nei singoli nodi ("fail soft") dalno la possibilita' di riassegnare le rotte (operazione comunque non priva di costi, tanto che alcune aziende preferiscono pagare le penali che adottare alcuni sistemi di ripresa).

Infine per quanto riguarda le **Prestazioni** la possibilita' di effettuare operazioni locali introduce la parallelizzazione di esse per quanto possibile ma anche la necessita' di coordinamento tra i nodi per effettuare transazioni globali (con potenzialmente aumento del traffico in rete).

2.5 Funzionalita' Specifiche dei DDBMS rispetto ai DBMS Centralizzati

Ogni server mantiene la capacita' di gestire applicazioni in modo indipendente ma coopera con gli altri server, e queste interazioni sono un carico aggiuntivo sul sistema complessivo.

La rete e' l'elemento critico, quindi e' necessario distribuire i dati in modo che la maggior parte delle transazioni sia locale o si eviti l'invio di dati tra nodi.

Rispetto ai DBMS Centralizzati abbiamo sia la trasmissione di query che di dati del DB, che di dati di controllo. Inoltre ci sono in piu' da gestire la frammentazione, la replicazione in modo da garantire la trasparenza ai nodi.

Il query processor deve pianificare una strategia globale accanto a quelle locali e bisogna controllare la concorrenza durante le transazioni per evitare operazioni non consistenti. Inoltre si devono prevedere strategie di ripristino dei singoli nodi come parte della gestione di guasti globali.

2.5.1 Frammentazione, Replicazione, Trasparenza

La frammentazione e' la possibilita' di allocare porzioni diverse del DB su nodi diversi. Questa puo' essere sia verticale (sulle colonne), che orizzontale (sulle righe).

Chiaramente durante la frammentazione e' necessario garantire la **Completezza** dei dati, la loro **Ricostruibilita'** e la **Disgiunzione** (ogni elemento e' in un solo frammento a meno di Replicazione).

La replicazione e' la possibilita' di allocare le stesse porzioni di DB su nodi diversi

La trasparenza rende possibile al sistema di accedere ai dati senza sapere dove sono allocati. Questa in particolare e' una caratteristica che di solito si ottiene solo pagando una certa cifra (altrimenti solitamente bisogna indirizzare tutto a mano).

Questa e' solitamente di due tipi:

- trasparenza logica: indipendenza dell'applicazione da modifiche dello schema logico.

- trasparenza fisica: indipendenza dell'applicazione da modifiche allo schema fisico.

A sua volta la trasparenza si divide in:

- trasparenza di frammentazione: il sistema non sa che i dati sono frammentati.
- trasparenza di allocazione: sa che e' frammentato ma non sa dove sono allocati.
- trasparenza di linguaggio: e' possibile usare lo stesso linguaggio di query (SQL standard) a prescindere dalle interfacce dei nodi singoli.

Chapter 3

Richiamo sui DBMS Centralizzati

3.1 DBMS

E' un sistema in grado di gestire collezioni di dati che siano di grandi dimensioni, persistenti, condivise ed affidabili (resistenti ai guasti).

3.1.1 Caratteristiche

Abbiamo un unico schema logico, una unica base di dati e un unico schema fisico, quindi nessuna forma di eterogeneita' contettuale o fisica, e nessuna forma di distribuzione.

Inoltre abbiamo un unico linguaggio di interrogazione, un sistema di gestione e di accesso, una modalita' di ripristino e un amministratore dei dati.

La persistenza richiede una gestione in memoria secondaria sogisticata in modo da archiviare ed accedere in modo efficiente una grande mole di dati.

Inoltre richiede l'uso condiviso di risorse tra piu' applicazioni, e quindi la gestione della concorrenza di operazioni sulla stessa base di dati tramite meccanismi di autorizzazione e di controllo della concorrenza che garantiscano si' la sicurezza ma anche una ragionevole efficienza.

Le interrogazioni devono essere ottimizzate il piu' possibile e i dati devono essere conservati anche in presenza di malfunzionamenti.

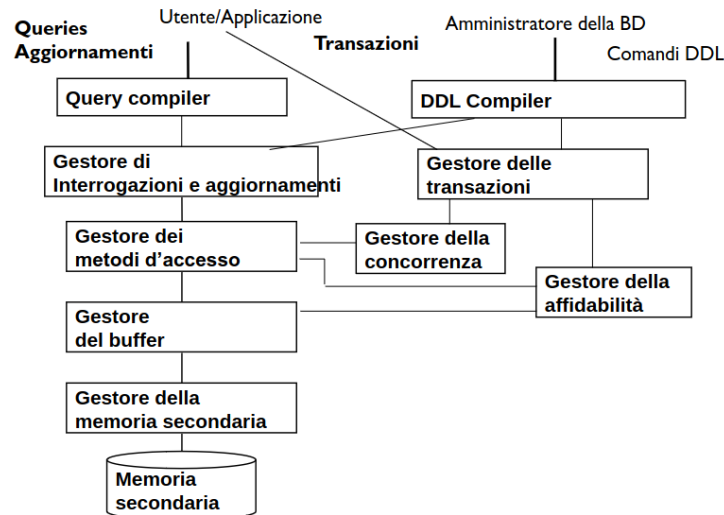


Figure 3.1: Schema di un DBMS

3.2 Ottimizzazione delle Interrogazioni

Si trasforma il query tree (parsed SQL) in un query plan logico contenente operazioni di algebra relazionale sulle tabelle logiche dello schema e lo ottimizza rispetto al costo.

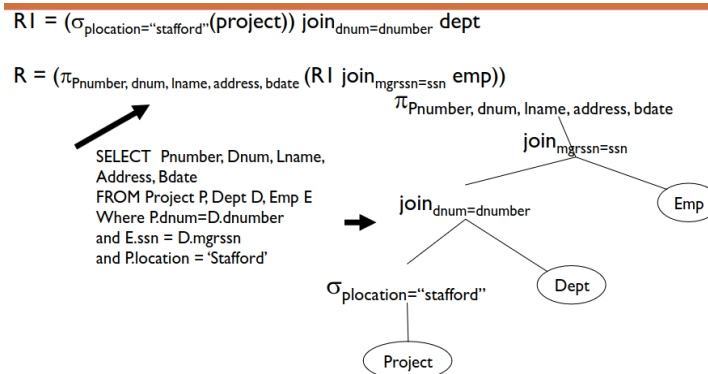


Figure 3.2: Query Tree/Plan

Visto che un query tree può essere trasformato in più Query Plan logici possibili, si sceglie quello computazionalmente migliore. Ad esempio, in genere conviene che le selezioni siano anticipate rispetto ai join, perché in questo modo i join operano su tabelle di minore dimensione.

La trasformazione è effettuata usando proprietà algebriche e stime dei costi delle operazioni accessibile tramite una tabella con delle statistiche. In generale il problema di ottimizzazione ha complessità esponenziale, quindi si introducono delle approssimazioni (si ottimizza finché si riesce in base a delle euristiche).

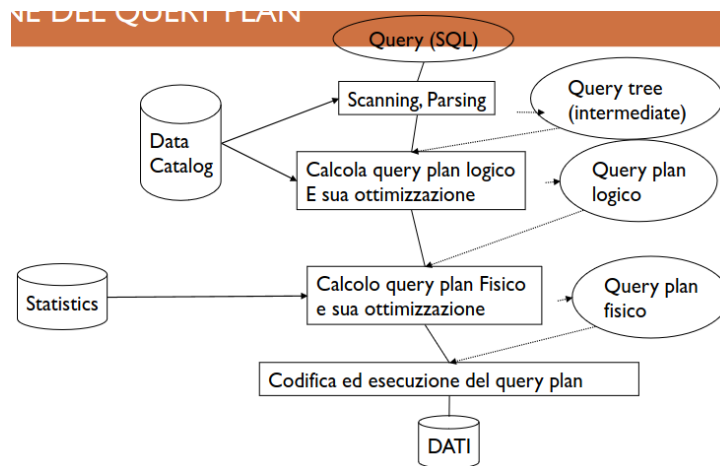


Figure 3.3: Schema di Ottimizzazione

3.3 Transazioni

Una transazione e' una serie di istruzioni di lettura e scrittura sulla base di dati (eventualmente in un linguaggio di programmazione) il cui esito puo' essere **commit work** o **rollback work**; ovvero memorizza quanto fatto o dimentica tutto e vai avanti, non sono ammesse soluzioni intermedie. Un sistema transazionale (OLTP) e' in grado di definire ed eseguire le transazioni per conto di un certo numero di applicazioni concorrenti.

3.3.1 Proprieta' ACID

Atomicita' : una transazione e' una unita' atomica di elaborazione, non puo' lasciare la base i dati in uno stato intermedio.

Consistenza : la transazione rispetta i vincoli di integrita', ovvero se lo stato iniziale e' corretto anche lo stato finale sara' corretto.

Isolamento : la transazione non risente degli effetti delle altre transazioni concorrenti (determinismo?).

Durata (Persistenza): un commit prsiste i dati anche in caso di quasti (necessario che il DBMS garantisca l'affidabilita').

Le anomalie che possono avvenire durante piu' transazioni concorrenti:

- **update loss**: una scrittura e' persa per colpa di una sovrascrittura
- **dirty read**: una lettura di un dato scritto e non confermato (commit)
- **unrepeatable read**: due letture di seguito restituiscono valori differenti
- **ghost update**: *non l'ho capito in realta'*



Figure 3.4: Foto di Maurino non spiegata, penso che sia un update loss (ha perso il posto in aereo)

3.3.2 Gestione della Concorrenza

Una sequenza di esecuzione di un insieme di transazioni e' detta **schedule**. Essa si dice seriale se una transazione termina prima che la successiva inizi. Una schedule e' serializzabile se l'esito della sua esecuzione e' lo stesso indipendentemente dall'ordine di esecuzione seriale delle transazioni in essa contenute.

Per controllare la concorrenza e' necessaria la presenza di uno scheduler a cui si richiedono e si rilasciano i **lock**, le cui assegnazioni sono salvate nella lock table. Si aggiunge anche la regola **Two Phase Locking** (2PL), ovvero in ogni tutte le richieste di lock precedono tutti gli unlock.

Chapter 4

DBMS Distribuiti

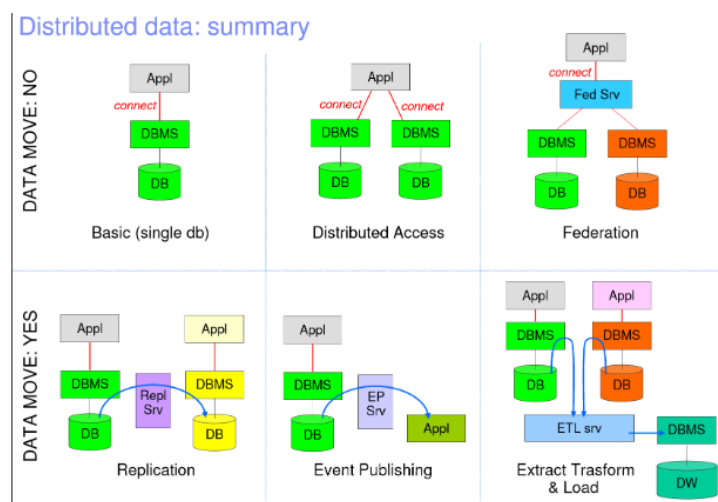


Figure 4.1: Di tutti i tipi di DDBMS ci interessano quelli ad Accesso Distribuito

4.1 Query Processing

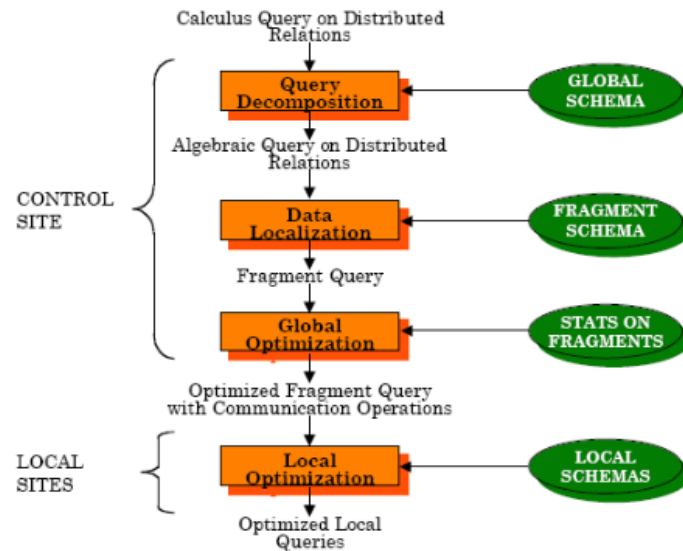


Figure 4.2: Schema di Query Processing Distribuito

- **query decomposition:** simile alla fase per i DBMS centralizzati, non considera la distribuzione dei dati e quindi non ottimizza rispetto ai costi di comunicazione.
- **data localization:** considera la distribuzione dei rammenti e ottimizza le operazioni in tal senso con tecniche di riduzioni, ma ancora non e' ottimizzata.
- **global query optimization:** utilizza le statistiche sui frammenti per ottimizzare le operazioni del query tree e trovare l'ordinamento migliore di esecuzione (in particolare i join).
- **local query optimization:** questa fase e' simile a quella dei DBMS centralizzati ed eseguita in modo indipendente sui nodi.

4.1.1 Esempio

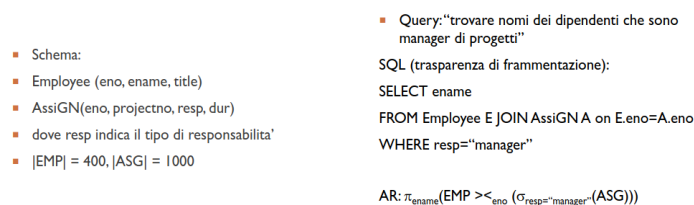


Figure 4.3: Query da Ottimizzare

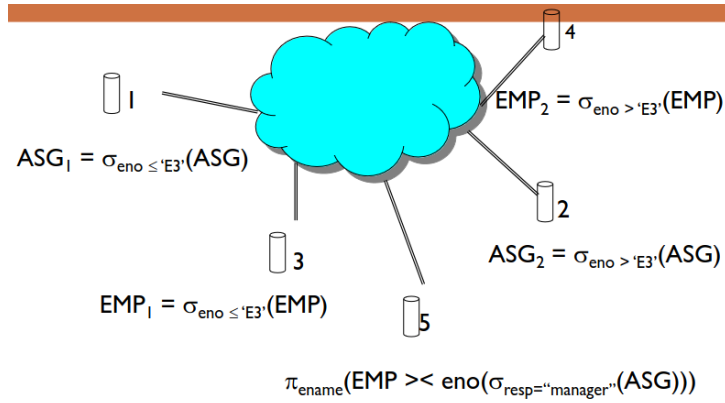


Figure 4.4: Disposizione dei Frammenti

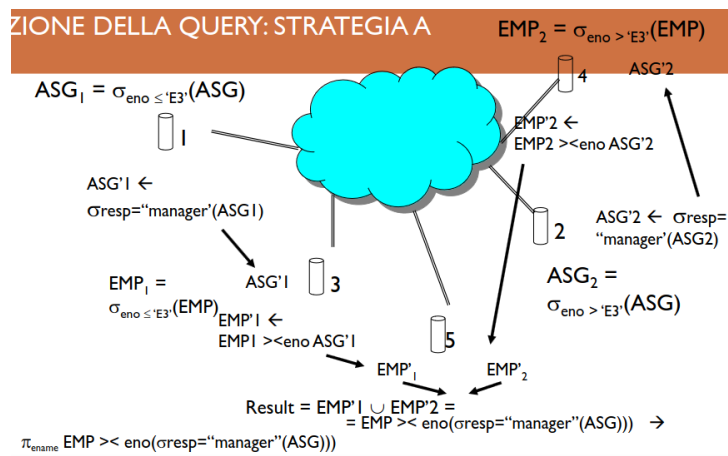


Figure 4.5: Strategia A

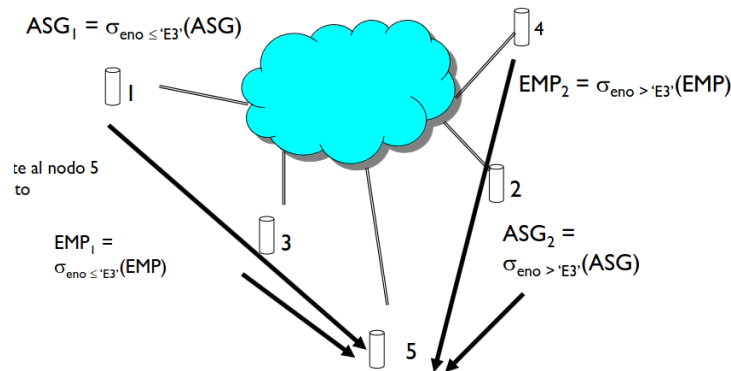


Figure 4.6: Strategia B

O invio i dati progressivamente dove devono essere elaborati (Strategia A) e in quel caso il tempo massimo di attesa e' dato da uno dei due rami. Oppure invio tutti i dati al nodo di elaborazione 5 e in questo caso il costo massimo e' dato da uno degli invii.

La scelta di una di queste due possibilita' non e' banale perche' posso avere casi in cui la B sia piu' vantaggiosa. Per esempio quando la rete e' poco congestionata e tempi di invio sono molto bassi. Ovvero se siamo interessati a minimizzare il tempo totale di elaborazione (A) o a massimizzare la parallelizzazione (B).

4.1.2 Semi Join

Il Semi Join in alcuni casi puo' risultare piu' efficiente. Consiste nell'inviare al Nodo 1 le chiavi primarie usate nel join dal Nodo 2 affinche' il Nodo 1 effettui un'operazione simile alla select. Quindi rimanda indietro i dati e il Nodo 1 li aggrega per formare il join finale. Attenzione: il Semi Join non e' commutativo.

Ci sono piu' strategie per il join tramite semijoin:

- $Rjoin_AS \Leftrightarrow (Rsemijoin_AS)join_AS$
- $Rjoin_AS \Leftrightarrow Rjoin_A(Ssemijoin_AR)$
- $Rjoin_AS \Leftrightarrow (Rsemijoin_AS)join_A(Ssemijoin_AR)$

La scelta di una strategia richiede la stima dei costi.

In generale comunque, l'uso del semijoin e' conveniente se il costo del suo calcolo e del trasferimento del risultato e' inferiore al costo del trasferimento dell'intera relazione e del costo del join intero.

4.2 Controllo della Concorrenza

Le transazioni distribuite di tipo read-write richiedono un protocollo transazionale di coordinamento (two-phase commit).

La distribuzione non ha conseguenze su consistenza (essa dipende solo dalla schema logico) e sulla durabilita' (garantita localmente dai singoli nodi). Invece e' necessario garantire Isolamento e Atomocita' in questo caso particolare.

Per quanto riguarda la serializzabilita' globale della schedule, si osserva che non e' automatica data la serializzabilita' delle schedule locali. Infatti se il DB non e' replicato e ogni schedule locale e' serializzabile non ci sono problemi (gli ordini di serializzazione sono gli stessi per tutti i nodi coinvolti), ma se abbiamo delle repliche possiamo avere casi in cui due schedule sia ordinate differentemente su due nodi replicati distinti provocando un'inconsistenza nei dati. Serve un protocollo di controllo delle repliche.

4.2.1 Protocollo ROWA per il controllo delle repliche

Read Once, Write All. Ovvero provo a scrivere una transazione su tutti i nodi replicati, e confermo la possibilita' di leggere i dati scritti se e solo se tutti i nodi hanno finito di scrivere.

4.2.2 2PL Centralizzato

Ogni nodo ha un Lock Manager e uno viene eletto coordinatore (esso gestisce i locks per l'intero DDB). Il Transaction Manager e' il coordinatore della transazione, esso richiede i lock al Lock Manager Centrale e li assegna ai nodi che devono effettuare l'accesso ai dati. Finite le operazioni chiede il rilascio dei lock al Lock Manager Centrale.

Il problema e' che avere un unico Lock Manager Centrale e' un importante collo di bottiglia.

4.2.3 Primary Copy 2PL

Per ogni risorsa e' individuata una copia primaria, che e' individuata prima dell'assegnazione dei lock. Diversi nodi hanno lock manager attivi e ognuno gestisce una partizione dei lock complessivi. Per ogni risorsa della transazione il Transaction Manager comunica le richieste di lock al Lock Manager responsabile della copia primaria, che assegna i lock. Così' si evita il bottleneck di avere un solo Lock Manager Centrale ma e' anche vero che e' necessario determinare a priori i lock manager che gestiscono le risorse e il mantenimento di una directory globale per sapere chi e' chi.

4.3 Deadlock Distribuito

Il problema delle transazioni con i lock e' anche stabilire quando e' avvenuto un deadlock, ovvero un'attesa circolare tra due o piu' nodi. Di solito l'algoritmo e' implementato con dei time-out come trigger.

E' attivato periodicamente sui nodi del DDBMS: in ogni nodo integra la sequenza di attesa con le condizioni di attesa locale degli altri nodi logicamente legati dalle condizioni di attesa. Quindi analizza le condizioni di attesa sul nodo e rileva i deadlock locali e comunica le sequenze di attesa ad altre istanze del DDBMS. Quest'ultima cosa e' dovuta al fatto che e' possibile evitare che uno stesso deadlock venga scoperto piu' volte, e quindi che tutte le transazioni coinvolte vengano uccise per poi ripartire da capo.

4.4 Recovery Management

La gestione dei guasti e' particolarmente problematica nei DDBMS perche' il fallimento di alcuni nodi durante una transazione o la perdita di messaggi in generale lasciano l'esecuzione di un protocollo in una situazione di indecisione che deve essere gestita. Per questo il protocollo 2PC permette di giungere ad una decisione di abort/commit su ciascuno dei nodi in modo affidabile.

4.5 Two Phase Commit

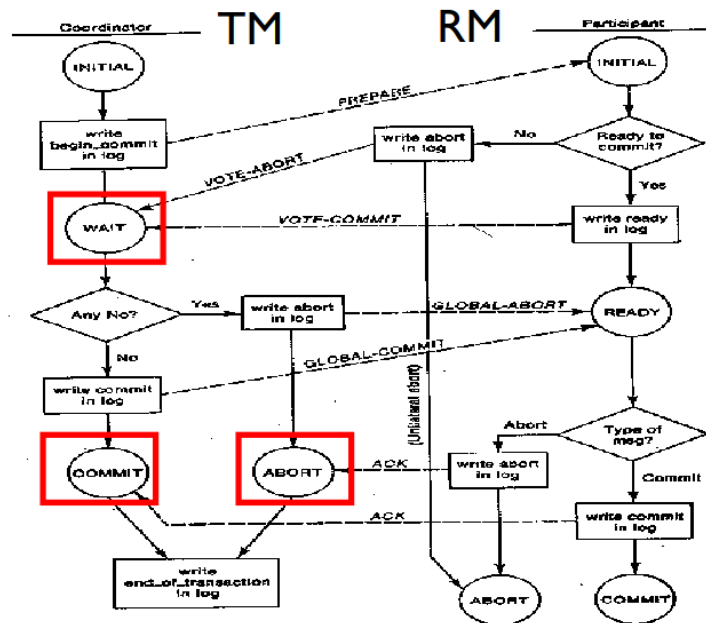


Figure 4.7: Schema 2PC

La decisione di commit/abort e' decisa da un coordinatore chiamato Transaction Manager che invia la richiesta se committare o abortire a tutti i nodi coinvolti (Resource Managers). Se anche uno solo di essi dichiara abort allora la transazione e' abortita, altrimenti si committa. Presa la decisione, il Transaction Manager la comunica ai Resource Manager e si assicura che ognuno di essi abbia capito la decisione. Una cosa importante e' che prima di ogni operazione ogni partecipante logga le decisioni prese in merito alle decisioni: in questo modo se un nodo dovesse morire per qualche motivo, al suo risveglio potrebbe subito in modo univoco stabilire cosa deve fare.

E questo e' un tipo di guasto. Ma che succede se un nodo non risponde alla domanda commit/abort? Gli rimanda il messaggio, infatti e' necessario stabilire se tutti i nodi sono favorevoli. Invece per quanto riguarda l'invio della decisione, anche in questo caso il TM deve assicurarsi che tutti i nodi abbiano compreso, di conseguenza deve rinviare la decisione finche' non e' afferrata.

Esistono vari paradigmi di comunicazione tra i nodi partecipanti, si rimanda alle immagini.

- Centralizzato (e' quello che abbiamo visto)
 - La comunicazione avviene solo tra TM e ogni RM
 - Nessuna comunicazione tra RM

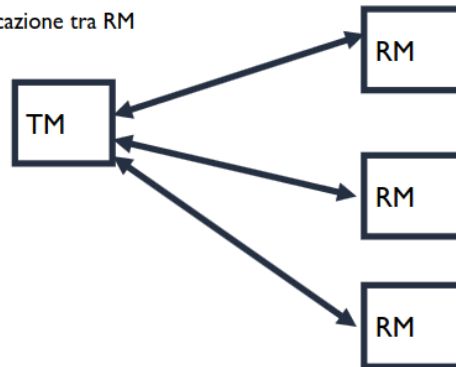


Figure 4.8: Schema Centralizzato

- Lineare:
 - Gli RM comunicano tra loro secondo un ordine prestabilito
 - Il TM e' il primo nell'ordine
 - Utile solo per reti senza possibilita' di broadcast

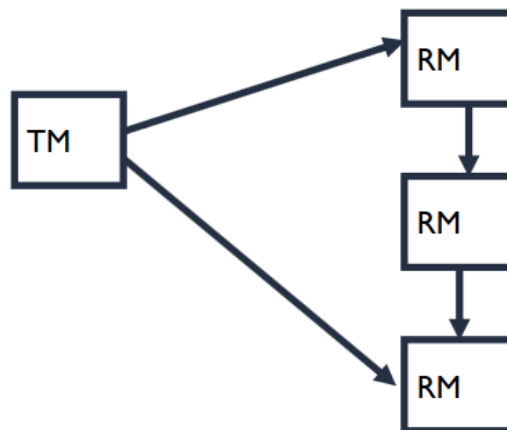


Figure 4.9: Schema Lineare

- Distribuito:
 - Nella prima fase, il TM comunica con gli RMs

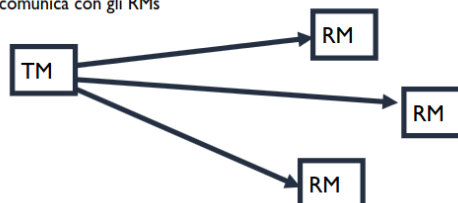


Figure 4.10: Schema Distribuito

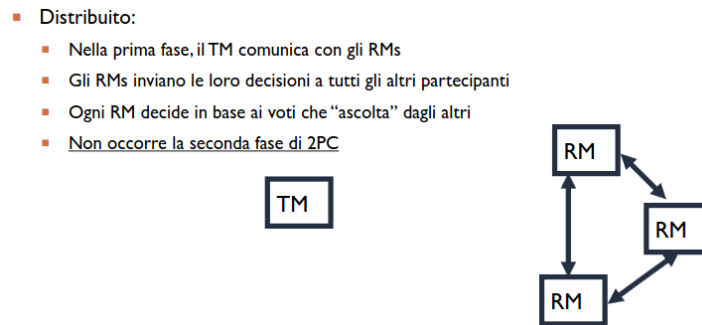


Figure 4.11: Schema Distribuito 2

Ora, cosa succede se un Transaction Manager muore? semplice, morto un Papa se ne fa un altro: un altro nodo e' eletto TM. Ora, se il guasto al TM e' risolto e si risveglia possono avvenire molte cose:

- l'ultimo record e' *prepare*: il guasto puo' aver bloccato alcuni RM, di conseguenza o si procede con un *global-abort* e si rifa' da capo, oppure si ripete la fase di richiesta sperando di giungere ad un *global-commit*.
- l'ultimo record e' *global-commit/global-abort*: il TM non ha idea di se tutti i RM hanno completato le operazioni quindi ripete la fase di invio della decisione.
- l'ultimo record e' *complete*: in questo caso non ha effetto.

Se invece e' il Resource Manager a morire utilizza il log per sapere cosa fare. Se l'ultimo record e' *ready* o chiede al TM cosa deve fare oppure sara' il TM a chiederglielo. Ad ogni modo si procede.

4.5.1 Ottimizzazioni

Si possono ridurre il numero di messaggi trasmessi tra i nodi e il numero di scritte nel log.

Quando un RM sa che la propria transazione non contiene operazioni di scrittura, non deve influenzare l'esito finale della transazione, quindi risponde al *prepare* con un *read-only* ed esce dal protocollo.

Un'altra possibilita' e' che il TM esca subito dalla transazione in seguito alla decisione di abort. In questo caso se il TM riceve una richiesta di remote recovery da parte di un RM che non sa come procedere in seguito a un guasto, e il TM non trova abort nel log, l'RM puo' chiedere solo in uno stato in cui non c'e' stato il commit. Quindi decide sempre per abort.

4.6 X/Opex DTP

E' uno standard "aperto" per l'implementazione di transazioni distribuite e blah blah blah.

- The AP, the RM, and the TM communicate via three distinct interfaces: native, TX, and XA.
- The native interface is the medium by which the AP makes requests directly to the RM.
- This interface is RM specific.
 - either Embedded SQL or Client-Library.
- The TX Interface is the medium between the AP and the TM. The AP uses TX calls to delineate transaction boundaries.
 - This interface is TM specific.
- The XA Interface is the medium between the RM and the TM.
- Using XA calls, the TM tells the RM when transactions start, commit, and roll back. The TM also handles recovery.

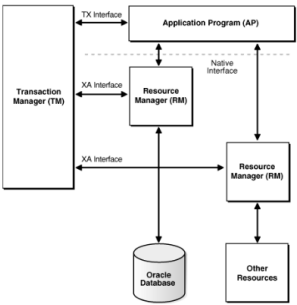


Figure 4.12: Architettura di X/Open DTP

XA Subroutine	Description
xa_open ()	Connects to the RM.
xa_close ()	Disconnects from the RM.
xa_start ()	Starts a new transaction and associates it with the given transaction ID (XID), or associates the process with an existing transaction.
xa_end ()	Disassociates the process from the given XID.
xa_rollback ()	Rolls back the transaction associated with the given XID.
xa_prepare ()	Prepares the transaction associated with the given XID. This is the first phase of the two-phase commit protocol.
xa_commit ()	Commits the transaction associated with the given XID. This is the second phase of the two-phase commit protocol.
xa_recover ()	Retrieves a list of prepared, heuristically committed, or heuristically rolled back transactions.
xa_forget ()	Forgets the heuristically completed transaction associated with the given XID.

Figure 4.13: Subroutines X/Open DTP

Part II

Use Case Poliglotta

Chapter 5

Use Case NOSQL

5.1 Use Case - 1

L'obiettivo e' costruire un DB per memorizzare le risposte di un API in ambito crypto: date le API response si vuole costruire delle tabelle per gestirne i dati.

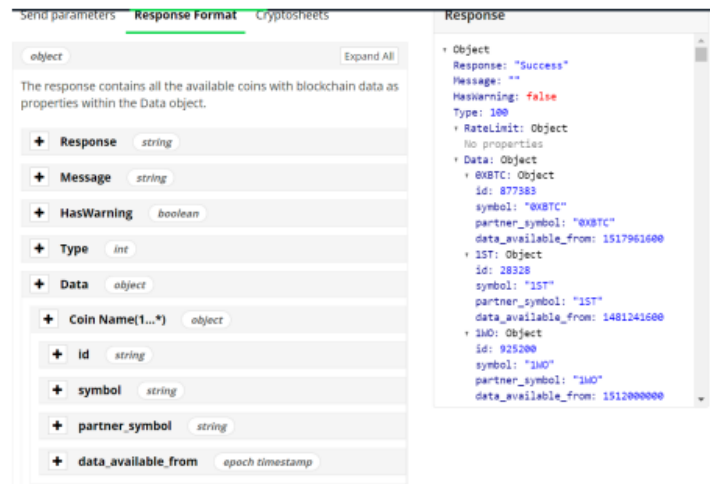
The screenshot shows an API client interface with the following sections:

- Send parameters:** A list of parameters to be sent in the request.
 - tryConversion:** boolean
 - fsyms:** string (Required)
 - tsyms:** string (Required)
 - relaxedValidation:** boolean
 - e:** string
 - extraParams:** string
 - sign:** boolean
- Info:** A section with a "Caching" toggle set to "10 seconds".
- Examples:** A list of example URLs.
 - `https://min-api.cryptocompare.com/data/pricemulti?fsyms=ETH,DASH&tsyms=BTC,USD,EUR`
 - `https://min-api.cryptocompare.com/data/pricemulti?fsyms=ETH,DASH&tsyms=BTC,USD,EUR&extraParams=your_app_`
- Response:** A section showing the JSON response from the API.

```
{
  "Object": {
    "BTC": {
      "USD": 67344.76,
      "EUR": 61813.12
    },
    "ETH": {
      "USD": 3566.19,
      "EUR": 3271.6
    }
  }
}
```

Figure 5.1: Esempio di Risposta

Si puo' pensare di costruire una tabella per ogni struttura di risposta, ma c'e' un problema, i dati possono essere strutturati:



<https://min-api.cryptocompare.com/documentation?>

Figure 5.2: Esempio di Schema Strutturato

Abbiamo bisogno di un modo per rappresentare un documento multi livello o strutturato senza esplosione delle tabelle, non possiamo creare una tabella per ogni possibile struttura diversa o per ogni array.

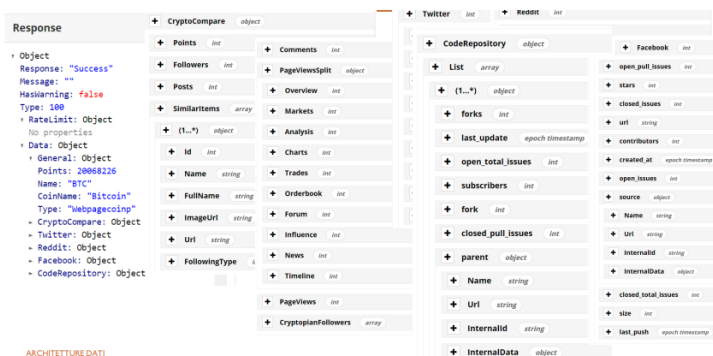


Figure 5.3: Esplosione delle Entita'

Qui si evidenzia il problema fondamentale delle applicazioni che lavorano su basi di dati: il collante tra il DB e l'applicazione, la configurazione. Esiste ORM come tanti altri paradigmi ma generalmente serve molta configurazione per mappare i dati da e al DB.

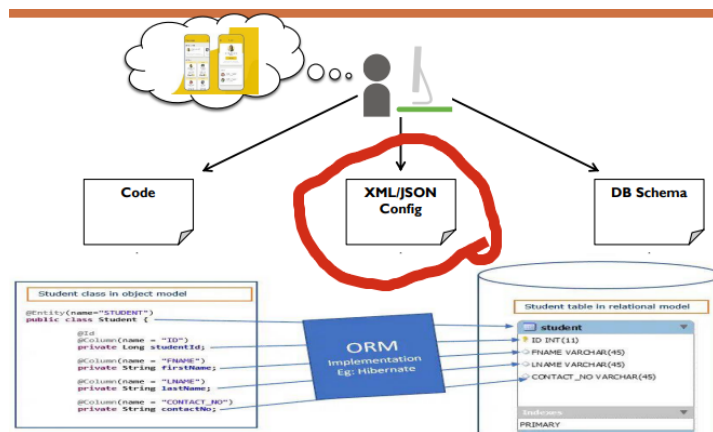


Figure 5.4: Il Collante

5.2 Use Case - 2

Si vuole creare un nuovo social network perche' non ne esistono gia' troppi. Serve una basi di dati **relazionale** per memorizzare gli utenti e le relazioni *segue* e *seguito da*.

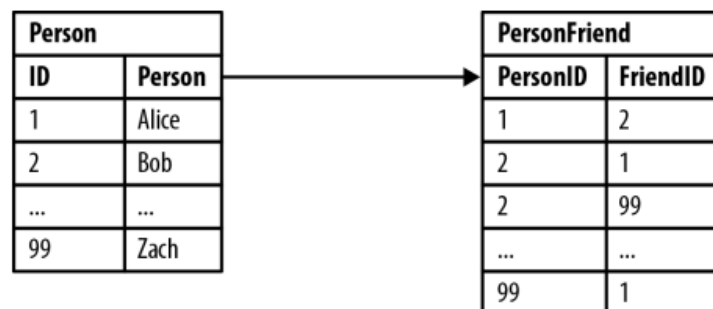


Figure 5.5: Schema del DB

A volte il problema non e' solo schematizzare i dati, ma anche interagirci ... gia' una query per trovare gli *amici degli amici* e' "complessa", figuriamoci se dovessi andare avanti con gli *amici degli amici degli amici degli amici* (se pensi che sia improbabile non hai lavorato sui piani di manutenzione e le configurazioni dei livelli). Il sistema potrebbe anche non reggere fisicamente a certi tipi di operazioni.

```
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN Person p1
  ON pf1.PersonID = p1.ID
JOIN PersonFriend pf2
  ON pf2.PersonID = pf1.FriendID
JOIN Person p2
  ON pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```

Figure 5.6: Query di "Amici di Amici"

5.3 NoSQL

E' un insieme di modelli di rappresentazione dati + software di gestione che sperimentano soluzioni alternative a SQL. Sono schemaless (senza schema), i dati sono lo schema (durante l'inserimento la memorizzazione implica la sua struttura).

Il vantaggio e' che i cambiamenti non sono onerosi, ma il problema e' che non si sa cosa contenga un database. Lo schema sono i dati stessi, e' necessario leggerli, non esiste un modo per sapere la loro struttura.

C'e' l'assunzione di mondo aperto (TODO). Piu' semplice e' il modello piu' scala bene ed e' flessibile.

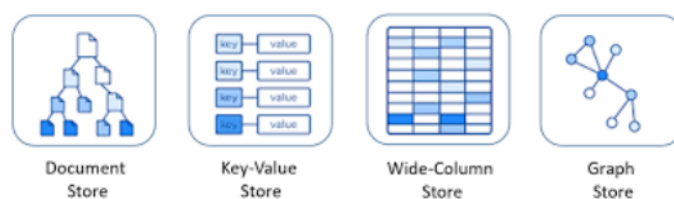


Figure 5.7: Modelli NoSQL

5.3.1 Key-Value

Un esempio e' Redis. E' il modello piu' semplice che si possa immaginare: una tabella chiave valore. Spesso e' tutto in memory.

5.3.2 Wide Column

Un esempio e' Cassandra. Ogni riga ha dei campi e qualcosa di diverso, non specifico uno schema ma solo i dati che conosco su una certa riga.

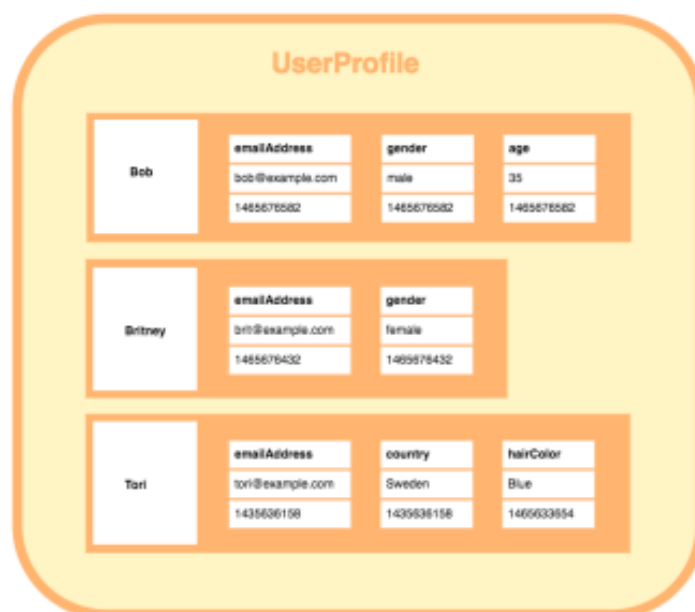


Figure 5.8: Esempio

5.3.3 Document Based

Un esempio e' CouchDB. E' un archivio di documenti (JSON per esempio). I documenti sono indirizzati nel DB tramite una chiave unica e si possono effettuare query di ricerca su essi.

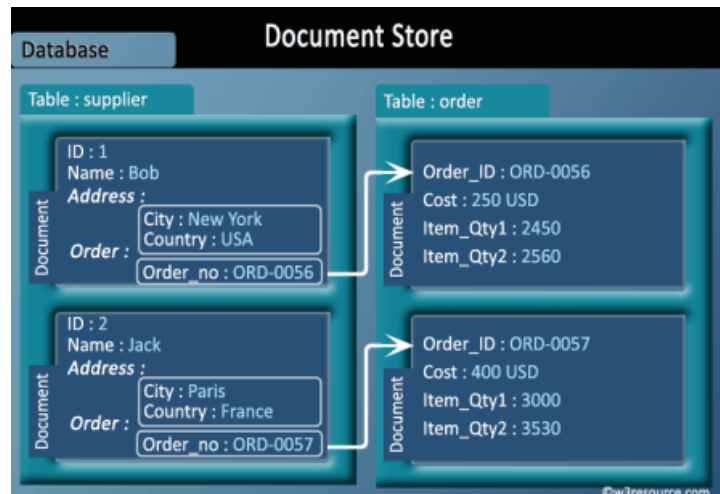


Figure 5.9: Esempio

5.3.4 Graph Based

Un esempio e' Neo4j.

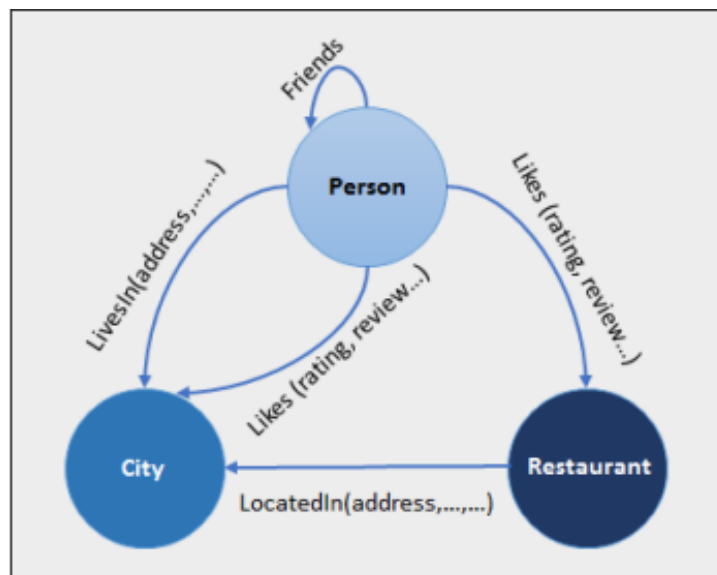


Figure 5.10: Esempio

5.3.5 Comfronto

Ogni modello ha i suoi pregi e i suoi difetti, per esempio un Grafo non puo' essere distribuito in modo orizzontale su piu' macchine mentre un Key Value non ha problemi.









	Type	Examples
Increasing Data Complexity ↓	Key-Value Store	 
	Wide Column Store	 
	Document Store	 
	Graph Store	 

Figure 5.11: Complessita

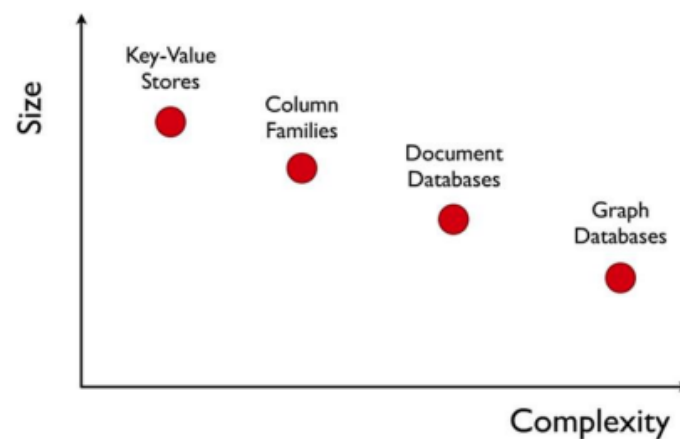


Figure 5.12: Dimensione e Gomplessita

5.4 Connettere i Dati

5.4.1 Model Data

Il problema rimane come mettere insieme dati legati fra loro.

PERSON

Pers_ID	Surname	First_Name	City
0	Miller	Paul	London
1	Ortega	Alvaro	Valencia
2	Huber	Urs	Zurich
3	Blanc	Gaston	Paris
4	Bertolini	Fabrizio	Rome

CAR

Car_ID	Model	Year	Value	Pers_ID
101	Bently	1973	100000	0
102	Rolls Royce	1965	330000	0
103	Peugeot	1993	500	3
104	Ferrari	2005	150000	4
105	Renault	1998	2000	3
106	Renault	2001	7000	3
107	Smart	1999	2000	2

Figure 5.13: Esempio Relazionale

Nel modello documentale i dati possono essere innestati, anche quelli delle relazioni delle "entita'":

```
{
  first_name: 'Paul',
  surname: 'Miller',
  city: 'London',
  location:
[45.123,47.232],
  cars: [
    { model: 'Bentley',
      year: 1973,
      value: 100000, ... },
    { model: 'Rolls Royce',
      year: 1965,
      value : 330000, ... }
  ]
}
```

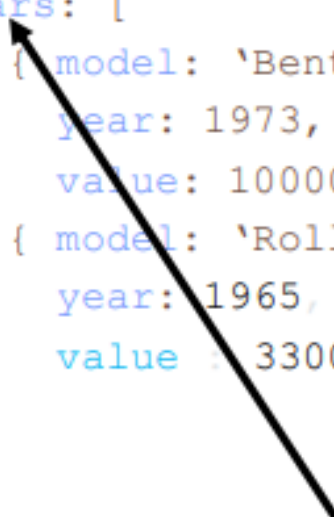


Figure 5.14: Esempio JSON

Il problema di un approccio del genere e' che l'unico modo per accedere ai dati delle relazioni sono costretto a passare per i singoli documenti. L'esempio di prima assume che si acceda sempre alle auto solo dopo alle persone.

E' necessario progettare la base di dati in base al problema che si deve risolvere e per come li deve processare l'applicazione. («A basi di dati si insegnava il contrario, ci scusiamo» - Maurino).

5.4.2 Model Comparison

Questi sono diversi modelli per rappresentare gli stessi dati. Non esiste un modello giusto o sbagliato, ne esiste solo un piu' corretto in base all'ambito e alle risorse del problema (e' appaltato al progettista l'onere della scelta).

ID	Name	Surname	DateofBirth
1	Tom	Hanks	...

Movie	Actor
1	1
2	1
3	1

Id	Title	Director
1	The Da Vinci Code	2
2	The Green Mile	3
3	That thing you do	1

Figure 5.15: Esempio Relazionale

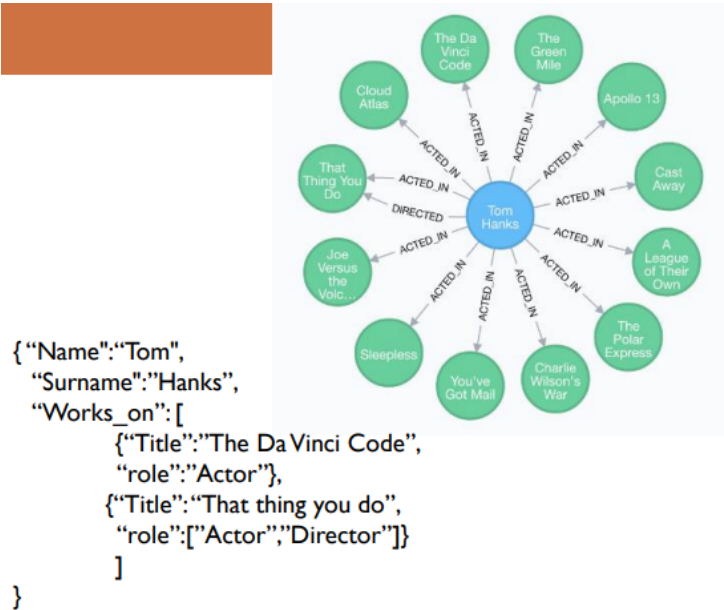


Figure 5.16: Esempi NoSQL

Diventa molto divertente lavorare con modelli ML su strutture a grafo o a documento. Il problema e' avere i dati nella forma d'uso, altrimenti e' necessario trasformarli o estrarre banche dati che possono introdurre errori (oversampling se porto i dati dell'entita' nella relazione per esempio).

5.5 Key Value

Il modello e' molto semplice, associo ad una chiave un valore (generalmente blob) accanto a cui posso salvare anche informazioni sul tipo del dato.

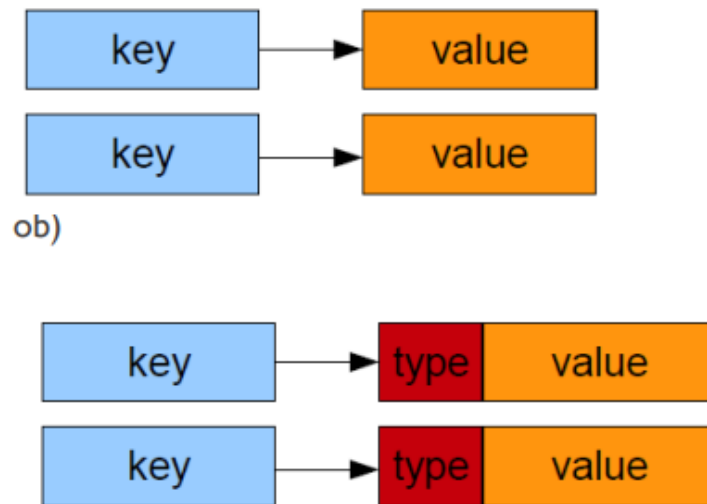


Figure 5.17: Modello Key Value

E' generalmente molto efficiente grazie all'uso delle HashTable e si presta alla distribuzione. Le operazioni sono le solite: aggiunta, cancellazione, ricerca.

5.5.1 Redis

Redis accomoda diversi tipi di dati: hashset, liste, code ... cosi' come i dati "primitivi" (stringhe, interi).

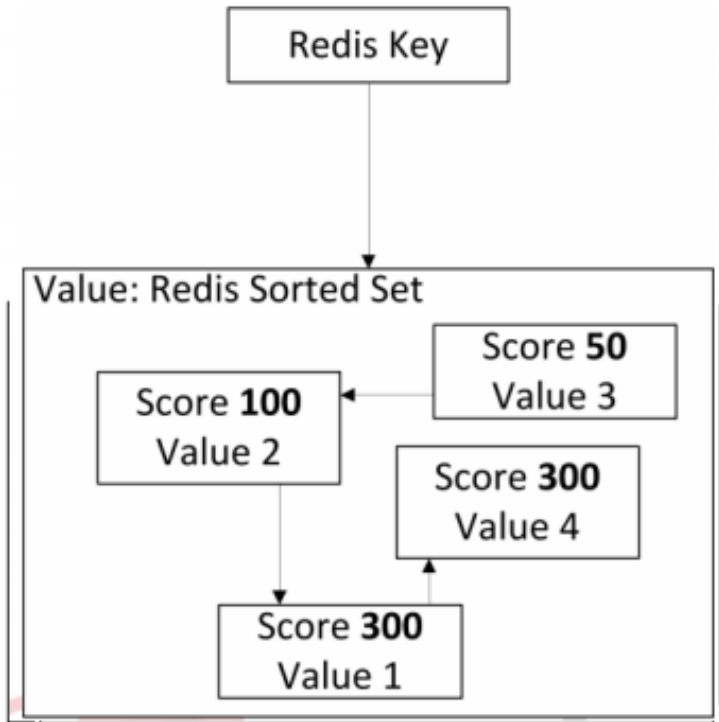
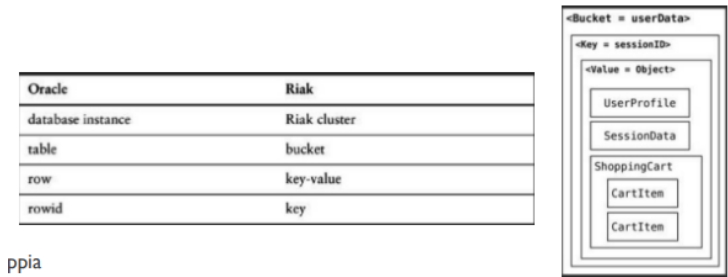


Figure 5.18: Esempio di Record



ppia

Figure 5.19: Modello di Redis

- 5.5.2 Wide Column
- 5.5.3 BigTable
- 5.5.4 HBase
- 5.5.5 Cassandra

Part III

Esercizi

Chapter 6

20/03

6.1 1

Si consideri la seguente base di dati:

- Production(SerialNumber, PartType, Model, Quantity, Machine)
- Pickup(SerialNumber, Lot, Client, SalesPerson, Amount)
- Client(ClientID, Name, City, Address)
- SalesPerson(SalesID, Name, City, Address)

Inoltre:

- Si assumano 4 production centers (Dublin, San Jose, Zurich, Taiwan), e tre punti vendita (San Jose, Zurich, Taiwan).
- Ogni production center e' responsabile per una parte: Cable (Dublin), CPU (San Jose), Keyboard (Zurich), Screen (Taiwan).
- Si supponga che i clienti di X siano serviti solo dalle persone del reparto vendite di X. (assumi che Zurich serva anche Dublin).
- Assumi che ogni area geografica abbia il suo database.

Crea una frammentazione orizzontale per le tabelle.

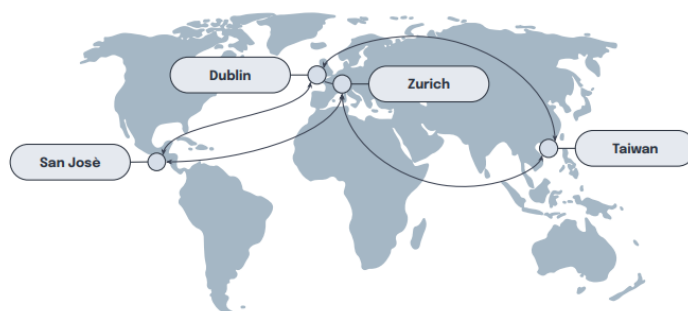


Figure 6.1:

6.1.1 Soluzione

Se ho già creato un frammento potrei riutilizzarlo in un'altra query ma qui assumiamo che ogni query veda solo lo schema globale.

Frammento Production:

- $\text{Production_1} = \sigma_{\text{PartType}=\text{CPU}}(\text{Production})$
- $\text{Production_2} = \sigma_{\text{PartType}=\text{KEYBOARD}}(\text{Production})$
- $\text{Production_3} = \sigma_{\text{PartType}=\text{SCREEN}}(\text{Production})$
- $\text{Production_4} = \sigma_{\text{PartType}=\text{CABLE}}(\text{Production})$

Frammento Pickup:

- $\text{Pickup_1} = \pi_{* \text{Pickup}}(\text{Pickup} \bowtie_{\text{Pr.SerialNumber}=\text{Pi.SerialNumber}} (\sigma_{\text{PartType}=\text{CPU}}(\text{Production})))$
- $\text{Pickup_2} = \pi_{* \text{Pickup}}(\text{Pickup} \bowtie_{\text{Pr.SerialNumber}=\text{Pi.SerialNumber}} (\sigma_{\text{PartType}=\text{KEYBOARD}}(\text{Production})))$
- $\text{Pickup_3} = \pi_{* \text{Pickup}}(\text{Pickup} \bowtie_{\text{Pr.SerialNumber}=\text{Pi.SerialNumber}} (\sigma_{\text{PartType}=\text{SCREEN}}(\text{Production})))$
- $\text{Pickup_4} = \pi_{* \text{Pickup}}(\text{Pickup} \bowtie_{\text{Pr.SerialNumber}=\text{Pi.SerialNumber}} (\sigma_{\text{PartType}=\text{CABLE}}(\text{Production})))$

Frammento Client:

- $\text{Client_1} = \sigma_{\text{City}=\text{SanJose}}(\text{Client})$
- $\text{Client_2} = \sigma_{\text{City}=\text{Zurich} \vee \text{City}=\text{Dublin}}(\text{Client})$
- $\text{Client_3} = \sigma_{\text{City}=\text{Taiwan}}(\text{Client})$

Per qualche ragione non esistendo una sede di Dublin ci sono comunque SalesPerson con City = Dublin, quindi le devo assegnare a Zurich. Frammento SalesPerson:

- $\text{SalesPerson_1} = \sigma_{\text{City}=\text{SanJose}}(\text{SalesPerson})$
- $\text{SalesPerson_2} = \sigma_{\text{City}=\text{Zurich} \vee \text{City}=\text{Dublin}}(\text{SalesPerson})$
- $\text{SalesPerson_3} = \sigma_{\text{City}=\text{Taiwan}}(\text{SalesPerson})$

6.2 2

- 1) Determine the available quantity of the product 77y6878
- 2) Determine the client IDs who have bought a lot from the retailer Wong, who has an office in Taiwan
- 3) Determine the machines used for the production of the parts type Keyboard sold to the client Brown (only one is present)
- 4) Modify the address of the retailer Brown (only one is present), who is moving from 27 Church St. in Dublin to 43 Park Hoi St. in Taiwan
- 5) Calculate the sum of amounts of the orders received in San Josè, Zurich and Taiwan (note that the aggregate functions are also distributable)

6.2.1 Soluzione

1

- `SELECT Quantity FROM Production WHERE SerialNumber = 77y6878`
- `(SELECT Quantity FROM Production_1 WHERE SerialNumber = 77y6878) UNION (SELECT Quantity FROM Production_2 WHERE SerialNumber = 77y6878) UNION (SELECT Quantity FROM Production_3 WHERE SerialNumber = 77y6878) UNION (SELECT Quantity FROM Production_4 WHERE SerialNumber = 77y6878)`
- `(SELECT Quantity FROM Production_1@Server1 WHERE SerialNumber = 77y6878) UNION (SELECT Quantity FROM Production_2@Server2 WHERE SerialNumber = 77y6878) UNION (SELECT Quantity FROM Production_3@Server3 WHERE SerialNumber = 77y6878) UNION (SELECT Quantity FROM Production_4@Server4 WHERE SerialNumber = 77y6878)`

2

- `SELECT UNIQUE Client FROM Pickup_3 JOIN SalesPerson_3 ON Pickup_3.SalesPerson = SalesPerson_3.SalesID WHERE SalesPerson_3.Name = Wong GROUP BY (Client) HAVING COUNT(*) > 10`

3

4

5