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.

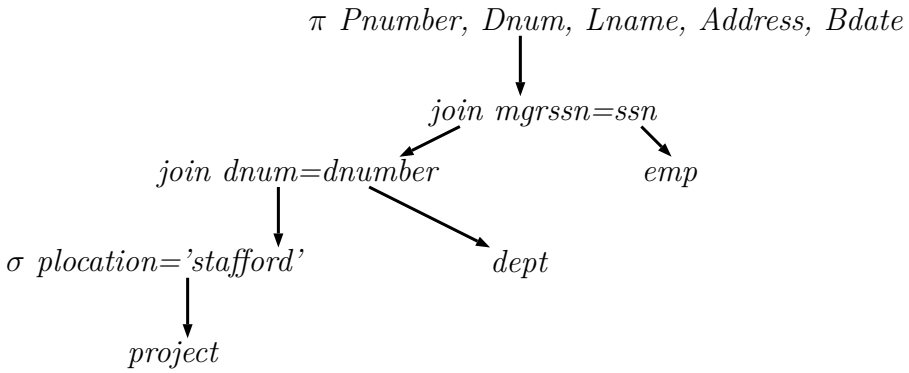
2.1 Ottimizzazione delle query

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
- **Durabilità** (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 è una descrizione architetturale abilitante il protocollo di esecuzione di transazioni distribuite), 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 Accesso in lettura

Studiamo prima le **query in scrittura**.
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 Accesso in scrittura e controllo di concorrenza

In questo caso la situazione si complica. Un conto è avere delle **remote requests (read-only)**, che possono essere un numero arbitrario di query SQL in sola lettura, un altro è avere delle **remote transactions (read-write)**, ovvero un numero arbitrario di operazioni SQL che prevedono anche *insert* e *update*. Ragionando in un'ottica in cui si ha un numero arbitrario di server si parla di **distribuited requests**, dove ogni singola operazione SQL si può riferire, grazie ad un *ottimizzatore distribuito*, a qualunque insieme di server, e di **distribuited transactions**, dove ogni operazione è diretta ad un unico server e dove le transazioni possono modificare più di un db, tutto ciò grazie ad un *protocollo transazionale di coordinamento distribuito*, detto **two-phase commit** (in questo caso si ha spesso a che fare con sistemi *eterogenei* e *federati*). Deve valore la **proprietà di atomicità** di *ACID*. Sempre riguardo *ACID* si ha che la **proprietà di consistenza** non dipende dalla distribuzione, in quanto le proprietà sono indipendenti dall'allocazione, e si ha che la **proprietà di durabilità** viene garantita localmente. Vanno invece rivisti architetturalmente la **proprietà di atomicità**, tramite componenti come il *reliability control* e il *recovery manager* (in caso di guasti), e la **proprietà di isolamento**, tramite il *concurrency control* (senza il quale si può incorrere in *update fantasma*, dove un *update* viene cancellato da uno seguente).
Rivedendo i **principi del controllo di concorrenza**.

Definizione 3. *Data una transazione t_i si ha che essa viene scomposta in sotto-transazioni t_{ij} a seconda del nodo j-simo su cui viene eseguita. A sua volta una transizione t_{ij} viene nominata r_{ij} per le operazioni di lettura e w_{ij} per le operazioni di scrittura. L'uso di una risorsa x viene indicata, per esempio, con $r_{ij}(x)$ o $w_{ij}(x)$.*

Ogni sotto-transazione viene schedulata in modo indipendente di server di ciascun nodo e quindi la **schedule globale** (dove *schedules* indica le sequenze delle transazioni da eseguire) dipende dalle **schedules locali** di ogni nodo.
Purtroppo **la serializzabilità locale di ogni schedule non garantisce**

la sua serializzabilità globale. Si viene a creare un **grafo globale dei conflitti** in quanto i conflitti non sono a livello locale ma a livello globale (infatti si hanno risorse occupate tra i vari nodi, si arriva, in certi casi, ad una **situazione di deadlock**).
Si ha quindi che lo *schedule globale* è serializzabile sse **gli ordini di serializzazione sono gli stessi per tutti i nodi coinvolti** (nel caso in cui il db non sia replicato).
Qualora si abbia un db replicato si aggiunge un altro problema qualora le scritture riguardino due repliche diverse. In tal caso si può violare la **mutua consistenza** (che dice che al termine della transazione tutte le copie devono avere lo stesso valore) dei due db locali, anche con due schedule localmente seriali. Si introduce quindi un **protocollo di controllo delle repliche**. Il **protocollo di controllo delle repliche** viene chiamato **ROWA (*Read Once Write All*)**. In base a questo protocollo, dato un item logico X (con $x_1 \dots x_n$ items fisici), si ha che le transazioni vedono solamente X ed è il protocollo che si occupa di mappare *read(X)* su una copia qualunque e *write(X)* su tutte le copie. Questo meccanismo torna utile nell'uso standard delle repliche, permettendo al client di leggere dal nodo più vicino ma imponendo, eventualmente, la scrittura su tutte le copie e bloccando le operazioni fino a quando l'ultima scrittura non è avvenuta. Si hanno purtroppo problemi di perdita di prestazioni a causa di questa scrittura "di massa". Per lo stesso problema si hanno anche condizioni di rilascio tramite *protocolli asincroni*.

3.2.3 2 phase locking

Anche l'**algoritmo 2PL (*2 Phase Locking*)** viene esteso al caso di schemi distribuiti. Si hanno due possibili strategie:

1. **primary site**, *centralized*, in quanto basata sui siti
2. **primary copy**, in quanto basata sulle copie

2PL prevede che prima di rilasciare un *lock* debba averli richiesti tutti, e nello *stricted 2PL* solo dopo che sia anche stato effettuato il *commit*.
Nel caso di *centralized 2PL* si ha un **lock manager (LM)** per ogni nodo, è un'architettura "master-slave". Il "master" è appunto il *lock manager* che gestisce i *lock* per l'intero db distribuito. Gli "slave" sono invece i *data processor*, che seguono quanto fa il *lock manager coordinatore* (che se non è disponibile per problemi tecnici del nodo comporta seri problemi in quanto la scelta di un nuovo *lock manager* tra quelli di ogni nodo è parecchio complicata).
Si ha inoltre che il **transaction manager (TM)** del nodo in cui inizia la

transazione sarà ritenuto il *TM coordinatore* dei transaction manager. La transazione anche in questo caso sarà eseguita dai *data processor* nei vari nodi. Il TM coordinatore formula all'LM coordinatore le richieste di *lock*, che vengono concesse tramite l'*algoritmo 2PL*. Una volta concesse il TM coordinatore le comunica ai vari *data processor*, assegnando ad essi i vari lock e l'accesso ai dati. AL termine delle operazioni i *data processor* comunicheranno il termine al TM coordinatore che a sua volta lo comunicherà all'LM coordinatore, che rilascerà i lock. Si ha però un effetto "collo di bottiglia" sul nodo del LM che deve gestire moltissime richieste e fino a che non risponde sistema va in *wait*. Una soluzione a questo problema è individuata nella tecnica della **copia primaria**. Prima dell'assegnazione del *lock*, viene individuata per ogni risorsa una *copia primaria*. Inoltre si ha che i diversi nodi hanno diversi *lock manager* attivi, ognuno che gestisce una partizione dei lock complessivi, relativi alle risorse primarie residenti nel nodo. Inoltre, per ogni risorsa nella transazione, il TM comunica le richieste di lock al LM responsabile della copia primaria, che assegna i *lock*. Si evita quindi il "collo di bottiglia" ma è necessario determinare a priori il LM per ogni risorsa. Inoltre si necessità di una **directory globale** dove tutti i nodi "vedono tutto".

3.2.4 Gestione dei deadlock

Indipendentemente da quanto appena discusso si può creare un'**attesa circolare** tra transazioni di due o più nodi. Bisogna quindi applicare un algoritmo distribuito. Per costruire l'algoritmo dobbiamo ragionare che siamo in una "rete tra pari" *Peer-to-Peer* (e non "master-slave") e quindi bisogna definire un protocollo su cui costruire l'algoritmo *asincrono e distribuito*. L'algoritmo potrebbe partire su uno qualsiasi dei nodi.
Si ipotizza innanzitutto che tutte le sotto-transazioni siano attivate in modo sincrono, tramite *Remote Procedure Call (RPC)* bloccante, ovvero una transazione chiede di fare un'operazione su un certo nodo facendo una RPC ad un altro nodo, mettendosi in attesa fino a che non finisce. Si possono generare due tipi di attesa:

1. **attesa da RPC**, una sotto-transazione su un nodo attende un'altra sotto-transazione, della stessa transazione, su un altro nodo

2. **attesa da rilascio di risorsa**, una sotto-transazione su un nodo attende un'altra sotto-transazione, della stessa transazione, sullo stesso nodo a causa del rilascio di una risorsa (che normalmente è la tipica situazione che porta ad un deadlock in un sistema centralizzato)

La composizione dei due tipi di attesa può generare un **deadlock globale**. E' possibile caratterizzare le condizioni di attesa su ciascun nodo tramite condizioni di precedenza e serve quindi specificare qualche notazione, per rappresentare il fatto che ogni nodo deve capire quali sono le transazioni sono in attesa per una chiamata esterna o per l'accesso ad una risorsa interna:

- EXT_i per una chiamata all'esecuzione di una transazione sul nodo i
- $x < y$ per indicare che x sta aspettando il rilascio di una risorsa da parte di y (che può essere anche EXT)
- indichiamo quindi la **sequenza di attesa generale** al nodo k come:
$$EXT < T_{ik} < T_{jk} < EXT$$

Esempio 4. *Sul DBMS1 si ha:*

$$EXT2 < T_{21} < T_{11} < EXT2$$

e sul DBMS2:

$$EXT1 < T_{12} < T_{22} < EXT1$$

ovvero sul nodo 2 c'è la transazione 1 che sta aspettando che finisca. La transazione 1 sul nodo 1 sta aspettando che la transazione 2 sul nodo 1 finisca, che a sua volta sta aspettando che la transazione sul nodo 2 finisca. Però sul DBMS2 scopriamo che la transazione 1 sul nodo 1 è in attesa della transazione 2 sul nodo 2 finisca. Quest'ultima sta aspettando che finisca la transazione 1 sul nodo 2 che a sua volta attende la transazione sul nodo 1. Si ha quindi un ***deadlock distribuito****, rappresentato nell'immagine 3.1.*

Per risolvere il problema del **deadlock distribuito** ogni nodo, ad un certo punto con un suo ordine temporale, prende la sua sequenza di attesa e la aggiunge ad alle condizioni di attesa locale degli altri nodi legati da EXT. Dopodiché analizza la situazione, rilevando potenziali **deadlock locali**, e comunica le sequenze di attesa alle altre istanze dell'algoritmo, ovvero agli altri nodi. Qualora si abbia un *deadlock locale* si crea un grafo dedicato allo stesso dove sarà possibile notare un ciclo, che rappresenta il deadlock. Ovviamente è possibile evitare che due nodi scoprano lo stesso deadlock, rendendo così quindi più efficiente l'algoritmo che invia le sequenze di attesa solo in alcuni modi:

1. **in avanti**, verso il nodo dove è attiva la sotto-transizione attesa (nodo nel quale vede che non ci siano deadlock (???)). Nei sistemi

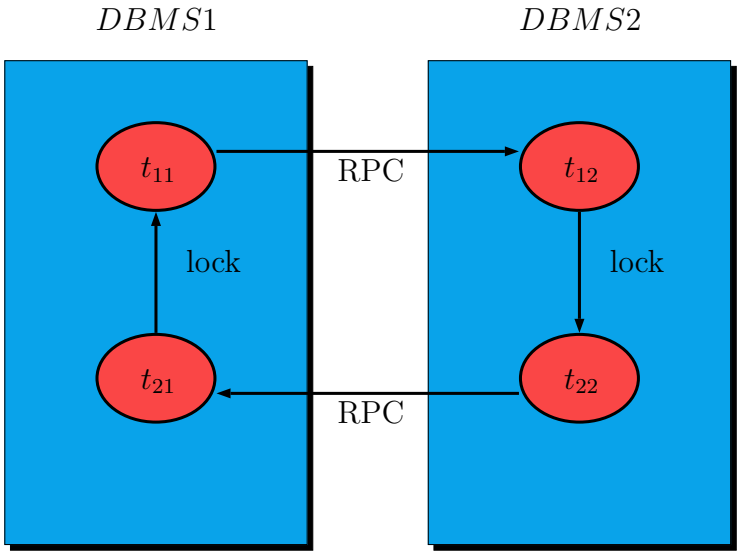


Figura 3.1: Grafico dell'esempio di deadlock distribuito

Peer-to-Peer questi sono meccanismi *coreografati*, decisi a priori. *Esempio alla slide 68 del quarto PDF della costruzione di un grafo delle transazioni con presenza di ciclo*

2. solamente quando l'identificatore del secondo nodo attende il rilascio della risorsa identificatore del primo nodo

3.2.5 Recovery management

Approfondiamo quindi come garantire la **proprietà di atomicità**.
Abbiamo visto come uno dei guasti possibili in un sistema distribuito sia quello legato alla perdita di messaggi sulla rete, nonché al partizionamento delle stessa (comportando magari l'isolamento dei nodi). Ricapitolando abbiamo diversi tipi di guasti:

- **guasti nei nodi**, sia *soft* che hard
- **perdita di messaggi**, che lascia l'esecuzione di un protocollo in uno stato di *indecisione*, in quanto ogni messaggio del protocollo è seguito da un *ack* e la perdita o del messaggio o dell'*ack* stesso genera incertezza (non potendo decidere se il messaggio sia arrivato o meno)
- **partizionamento della rete**, dove una transazione distribuita può essere attiva contemporaneamente su più sotto-reti temporaneamente isolate. In questa situazione i singoli nodi non riescono

a capire bene chi sia isolato e la cosa può portare i nodi a fare scelte contraddittorie

Si è quindi studiato il **protocollo two phase commit (*2PC*)** che cerca di funzionare in presenza di guasti di rete. Questo tipo di protocollo consente ad una transizione di giungere ad un eventuale *commit* o *abort* su ciascuno dei nodi che partecipano alla transazione. In questo protocollo la decisione di *commit* o *abort* tra due o più **resource managers (RM)** (i server) viene certificata da un **transaction manager (TM)** (il coordinatore). Lo scambio dei messaggi (e il salvataggio di un log per ciascuno) tra TM e RMs è ciò su cui si basa il protocollo 2PC. Si ha quindi sempre un'architettura "master-slave" (in modo metaforico si può rappresentare come il prete che unisce in matrimonio i due sposi). Si ha quindi il TM che interroga i RMs riguardo allo stato della loro esecuzione.

2PC in assenza di guasti

Si hanno diverse fasi:

1. durante la prima fase il TM interroga (con un *prepare* o *ready_to_commit*) tutti i nodi per capire come ciascun nodo intenda terminare la transazione, autonomamente o irrevocabilmente *commit* o *abort* (magari per violazione della concorrenza locale o per violazione di qualche vincolo di consistenza etc...). I nodi risponderanno quindi o con *ready_to_commit* o con *not_ready_to_commit*

2. nella seconda fase il TM prende la decisione globale. Si ha che se anche solo un nodo richiede un *abort* allora si avrà *abort* per tutti i nodi, altrimenti *commit*, chiudendo la transazione. Il TM si occupa anche di comunicare ai RMs la decisione finale per poter procedere con le azioni locali

Le fasi sono schematizzate in figura 3.2.
Come abbiamo detto precedentemente si ha la raccolta di *log* nei quali compaiono due tipi di record:

1. **record di transazione**, con le informazioni sulle operazioni effettuate

2. **record di sistema**, con l'evento di *checkpoint* e di *dump* (ovvero la copia esatta del db in un certo stato)

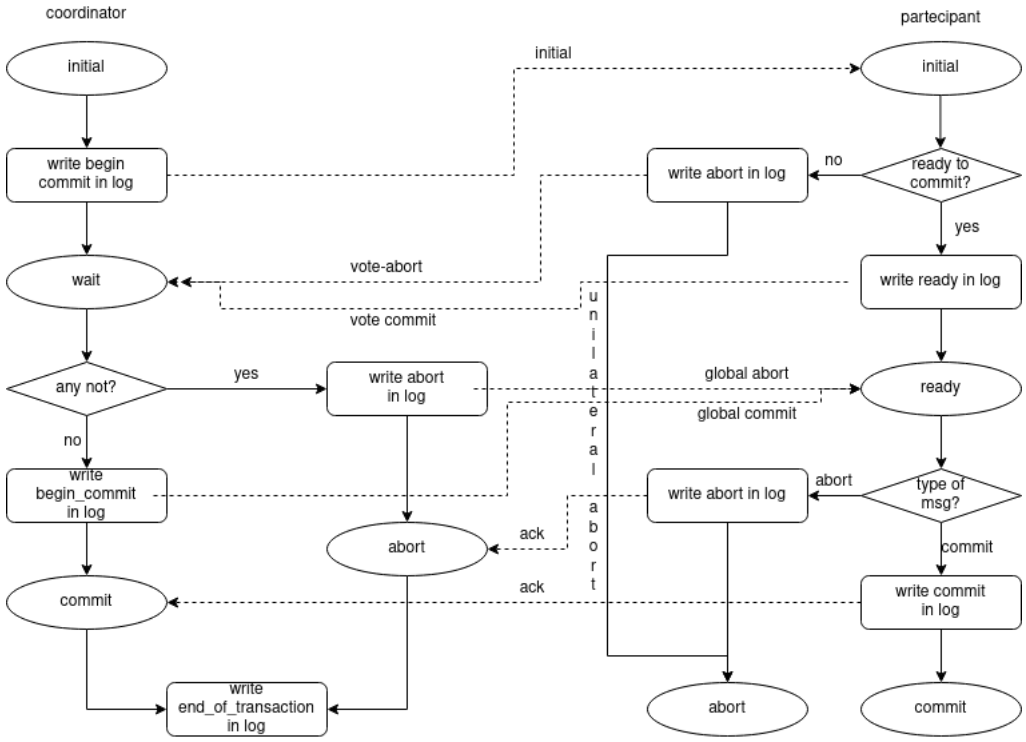


Figura 3.2: Diagramma del protocollo 2PC, dove negli ovali abbiamo le *decisioni* e nei rettangoli le *azioni sui log*. Entrambi i tipi di frecce indicano un *messaggio*. La prima fase è fino alle decisioni globali incluse.

Le scritture sui log avvengono prima della decisione delle operazioni, che a loro volta si suddividono in *prepare* e *global decision*. Ai log del TM vengono aggiunti ulteriori dettagli:

- **prepare record** (in figura 3.2 *begin_commit*) che contiene l'identità (nodi e transazioni) di tutti i RMs
- **global commit** o **global abort** che descrive la decisione globale. La decisione del TM diventa esecutiva quando scrive nel proprio log **global commit** o **global abort**
- **complete record** (in figura 3.2 *begin_of_transaction*), che viene scritto alla fine del protocollo

Al log dei RMs vengono aggiunti ulteriori dettagli:

- **ready record**, per segnalare la disponibilità irrevocabile a partecipare alla fase di commit. Si hanno diverse politiche sul **pro-**

tocollo 2PL (*recoverable, 2PL, ACR, strict 2PL*). Inoltre questo log contiene l'indicazione del TM e i records (come nel caso centralizzato), ovvero *begin, insert, delete, update, commit*

- **not ready record** (in figura 3.2 *abort*) per segnalare l'indisponibilità del RM al commit

In entrambe le fasi del 2PC possono avvenire guasti e in entrambe tutte le componenti devono poter decidere in base al loro stato. Viene quindi introdotto l'uso di **timers**, stabilendo un **timeout** entro il quale il TM aspetta una risposta (in entrambe le fasi, anche se nella prima è più importante). Se il timeout viene superato si lancia un *abort*. Vediamo distinte le due fasi

1. il TM scrive *prepare* nel suo log e invia *prepare* ai RMs, fissando un timeout massimo per le riposte. Gli RMs *recoverable*, ovvero pronti al *commit*, scrivono *ready* nel loro log e inviano *ready* al TM. Gli RMs non *recoverable*, ovvero non pronti al *commit* a causa di un deadlock, scrivono *not-ready* nel loro log e terminano il protocollo, con un *log unilaterale*. Il TM, come detto scrive nel suo log **global commit** o il **global abort**, il secondo nel caso in cui ci sia anche solo un *not-ready* o che scatti il timeout (assumendo che i nodi che non hanno risposto siano in *failure*).

2. il TM trasmette la decisione globale e fissa un secondo *timeout*. Gli RMs *ready* agiscono di conseguenza scrivendo *commit* o *abort* nei loro log e inviando un *ack* al TM. Solo dopo effettuano in locale *commit* o *abort*. Il TM raccoglie gli *ack* e, in assenza di qualche risposta, fissa un nuovo *timeout* ripetendo la trasmissione per gli RMs problematici. Questo fino a che non avrà ricevuto da tutti un *ack* e in quel momento scrive *complete* nel suo log

Abbiamo quindi appena visto un **paradigma centralizzato** "master-salve" dove i vari RMs non comunicano tra loro. Il TM è quindi il collo di bottiglia". Si hanno altri paradigmi:

- **lineare**, dove gli RMs hanno un ordine prestabilito e comunicano sempre secondo un ordine prestabilito. Il TM è solo il primo di tale ordine. Questo paradigma è utile per reti senza possibilità di *broadcast*

- **distribuito**. In questo caso nella prima fase il TM comunica coi vari RMs, che però rispondono a tutti. I vari RMs decidono in base alle informazioni ricevute dagli altri RMs (comunicano

tra loro in *broadcast*) e quindi non è più necessaria la seconda fase di *2PC*. Si genera una gran quantità di messaggi tra i vari RMs, avendo una situazione "tra pari" *Peer-to-Peer*, creando un problema di prestazioni, dovendo garantire la comunicazione tra tutti i nodi (anche tramite *ack*)

2PC in caso di guasto

In uno stato di guasto, nel caso **centralizzato**, un RM nello stato *ready* perde la sua autonomia e attende la decisione del TM. Nel caso di guasto del TM i vari RMs sono lasciati in *stato di incertezza* e le risorse allocate alla transazione restano bloccate.

Definizione 4. *Definiamo* ***finestra di incertezza*** *come la finestra temporale fra la scrittura di ready nel log dei RMs e la scrittura di* commit *o* abort*. Questo intervallo è ridotto al minimo da* ***2PC****.*

Durante la finestra di incertezza tutte le risorse acquisite tramite meccanismi di *lock* restano bloccate e, in caso di guasto durante la *finestra di incertezza*, TM e RMs usano i **protocolli di recovery**.
Si possono avere diversi guasti:

- **guasti di componenti**, ovvero guasti al TM o ai RMs.
- **perdita di messaggi o partizionamento della rete**

Guasti di componenti In questo caso devono essere usati protocolli con due compiti:

1. assicurare la terminazione delle procedure. Sono i **protocolli di terminazione**
2. assicurare il ripristino. Sono i **protocolli di recovery**

In entrambi i casi questi protocolli funzionano sia che il guasto interessi un solo componente che più di uno.
Partiamo dal caso in cui cada un RM. Può cadere prima di iniziare il protocollo, non rispondendo al *prepare* e portando all'*abort*. Può cadere dopo il *ready_to_commit* e se quando si riprende vede nei log lo stato *ready* si mette in *wait* non sapendo bene come fare (nel caso di *abort* semplicemente chiuderà il protocollo). Il TM provvederà a inviare a tale RM *commit* o *abort*. Non si hanno quindi problemi al protocollo, o si va diretti all'*abort* o si aspetta. Il TM capisce che un RM è caduto grazie al *timeout*.

Più complesso è il caso in cui cada il TM. Se cade prima di ricevere le risposte al *prepare*, quando si sarà ripreso, guarderà lo stato delle risposte al *prepare* ricevute, altrimenti si avrà un *abort unilaterale*.
Vengono introdotti gli **algoritmi bizantini** nel caso in cui il TM cada e il RMs debbano decidere insieme cosa fare. Per decidere serve visibilità.
Quindi, ricapitolando per le cadute del TM:

- cade quanto l'ultimo record del log è *prepare*, magari bloccando alcuni RMs. In tal caso si hanno 2 opzioni di recovery:
    1. decidere *global abort*, e procedere con la seconda fase di 2PC
    2. ripetere la prima fase, sperando di giungere a un *global commit*, richiedendo nuovamente lo stato dei RM
- cade quanto l'ultimo record del log è *global-commit* o *global-abort* il TM deve ripete la seconda fase in quanto alcuni RMs potrebbero essere bloccati o comunque ignari della decisione presa prima della caduta
- l'ultimo record nel log è una *complete*, in tal caso non si hanno problemi

Ricapitoliamo anche le cadute dell'RM:

- l'ultimo record nei log è di *azione*, *abort* o *commit*, come nel caso centralizzato e in tal caso di procede con un **warm restart**:
    - nel caso di *abort* o *azione* si procede con l'*undo* dell'operazione
    - nel caso di *commit* di effettua nuovamente la transazione
- l'ultimo record nei log è *ready* e in tal caso l'RM si blocca non conoscendo la decisione del TM e si inseriscono, durante *warm restart*, nel *ready set* le transazioni dubbie. Si hanno quindi due alternative:
    1. **remote recovery**, ovvero l'RM chiede al TM cosa è accaduto
    2. il TM riesegue la seconda fase del protocollo

Perdita di messaggi e partizionamento della rete In questo caso il TM non riesce a distinguere tra perdita di messaggi *prepare* o *ready* nella prima fase e procede, scattando i *timeout*, con un **global abort**.
Durante la seconda fase la non distinzione tra perdita di *ack* o di decisioni dei gli RM porta alla ripetizione della seconda fase dopo il *timeout*.
Se durante lo svolgimento del protocollo 2pc si partiziona la rete avendo due sottoreti una con TM e RM1 che e la seconda sottorete con RM2, RM3. In questo caso il TM continua a mandare il messaggio della global decision a RM2 RM3 dopo lo scadere del timeout. Si concluderà comunque, con alte probabilità, con un *abort* da parte del TM.

Ottimizzazioni di 2PC

2PC può essere ottimizzato per:

- ridurre il numero di messaggi tra TM e RMs
- ridurre le scritture nei log

Si hanno due tipi di ottimizzazione:

1. **ottimizzazione read-only**, quando un RM sa che se la propria transazione è *read-only* allora non influenza l'esito finale della transazione. Al *prepare* risponde *read-only* e termina il protocollo. Il TM ignora i partecipanti *read-only* dalla seconda fase e, qualora si sapesse a priori, possono anche essere direttamente esclusi dal protocollo
2. **ottimizzazione presumed abort** che si basa sulla regola "scordarsi gli *abort* e ricordasi i *commit*". In questo caso il TM abbandona la transazione dopo la decisione di abort senza scrivere *global abort* nel log e senza aspettare risposta dai vari RMs. Se il TM riceve richiesta di un *remote recovery* il TM decide per il *global abort*. Non sarà più necessario quindi scriver e*global abort* o *prepare* nei log ma solo *global commit, ready* e *commit*

Protocollo X/Open

Il protocollo 2PC è stato adottato nel protocollo **X/Open DTP (*Distributed Transaction Processing*)**, che è un consorzio di vendors che vogliono rendere portabile lo standard dell'ambiente UNIX. In questo protocollo si ha il *TX interface* per la comunicazione tra il TM e l'applicazione e si ha l'*XA interface* per la comunicazione tra TM e RMs. Ogni vendor ha la sua implementazione del modello.

3.3 Repliche

Parliamo ora delle tecnologie per la **replicazione di dati**.
Tra le principali soluzioni architetturale troviamo **IBM replication technologies** e **Microsoft SQL Server replication technologies** (*i dettagli delle implementazioni non saranno oggetto d'esame*).

Definizione 5. *La **replica** è il processo di cerare e mantenere istanze dello stesso db allineate tra loro, consentendo la condivisione di dati ma anche comportando cambiamenti architetturali. Si ha che le eventuali modifiche devono essere viste da tutti i nodi.*

Definizione 6. *Definiamo **sincronizzazione** è il processo che m i consente di allineare le copie, prima o poi.*

In base all'ultima definizione si capisce che spesso le copie non sono aggiornate istantaneamente. Si ha quindi la **replica sincrona** o la **replica asincrona** (non avendo allineamento *realtime*). IBM preferisce un approccio *based and mode based* mentre Microsoft uno basato su *snapshot, transactional* e *merge*.
Vediamo le differenze tra le due repliche:

- le **repliche sincrone** cercano di far si che tutte le repliche vengano aggiornate contemporaneamente (ad esempio si ha il protocollo ROWA). Scrivo in modo sincrono su tutti nodi e solo quando tutte le repliche confermano la scrittura avanzo con le transazioni (che fallisce se ho nodi non disponibili). Nelle repliche sincrone si necessitano molti scambi di messaggi. Di fatto si obbliga due o più *storage* ad aggiornarsi e a fare *rollback* in caso di fallimento. Si hanno quindi alte disponibilità, un auto *fail-over* (bloccando la transazioni in caso di guasti sui nodi) e un *data loss* minimo. Le repliche sincrone vengono soprattutto usate nei *disaster recovery*, ovvero in situazioni *mission critical* (ad esempio sistemi bancari dove i db di backup, almeno 3, devono stare a centinaia di kilometri di distanza, in zone sismiche tra loro diverse). Gli svantaggi delle repliche sincrone sono la necessità di una rete valida, si hanno problemi di scalabilità, di costi e minor flessibilità

- nelle **repliche asincrone** prima si aggiorna il db *target* e poi le repliche (normalmente dopo pochi secondi ma anche dopo giorni). Si hanno evidenti vantaggi di costo, scalabilità e flessibilità (perché in caso di problema lavoro in primis sul db principale) ma a

rischio di *data loss* (nell'intervallo di tempo tra la scrittura del db principale e delle repliche). Normalmente si usano soluzioni asincrone per accessi online e la loro efficienza, per bilanciamento del calcolo etc...

In caso di perdita di dati bisogna analizzare i singoli contratti per capire legalmente come rispondere di dati che non verranno recuperati probabilmente.
Ci sono vari contesti in cui pensare alla replica dei dati:

- condivisione di dati da utenti tra loro scollegati. Un esempio è una copia su un portatile con una replica usata da un commerciale. Si possono avere conflitti nel momento in cui più utenti con db replicati lavorano offline. Si ha il *merge conflict*
- *data consolidation*, ovvero quando un'azienda vuole tenere più copie dei dati in vari punti e alla fine bisogna riportare i dati a livello centrale a cadenza periodica. Può servire per fare data warehousing o anche solo semplicemente per monitorare le vendite delle varie filiali o per aggiornare il catalogo
- *data distribution* che è il caso degli *e-commerce*. È un caso tipicamente *mission-critical* e bisogna aumentare l'accesso ai dati e si ha una costante sincronizzazione realtime bidirezionale per evitare problemi. Un altro caso è la distribuzione tra diversi uffici, dove le repliche locali hanno magari dati non presenti nel db globale
- prestazioni, accesso efficiente, *load balancing* e accesso offline. Se non si hanno necessità di update immediati (tipo un sito vetrina) allora la replica garantisce la disponibilità e l'accessibilità a basso costo. Con il load balancing scarico gli utenti su diverse macchine replicate, in primis per le molte realtà di sola lettura o comunque con pochissime scritture (un esempio può anche essere un social network dove anche eventuali ritardi di qualche secondo non sono problematici). Per la disponibilità bisogna fare un forte testing dell'intera architettura, se cade un server bisogna puntare ad una replica e bisogna essere sicuri e spesso i costi sono troppo alti (*RIVEDERE QUESTA PARTE*)
- separazione tra *data entry* e *reporting*, se si usa lo stesso server per entrambi i compiti (che in pratica sono scrittura costante e lettura costante) può essere utile separare in due server. Si evitano così i rallentamenti dati dai *lock*. Bisogna studiare i tempi di sincronizzazione

- coesistenza di applicazioni, questo è un caso particolare. Qualora sia necessario cambiare applicazione devo, eventualmente, cambiare anche i sistemi. Bisogna quindi travasare i dati vecchi e durante il trasferimento bisogna comunque mantenere funzionanti le applicazioni. Quindi bisogna far coesistere i due database durante il trasferimento (facendo il travaso di notte bloccando le transazioni). I costi sono incredibili e si può avere anche coesistenza delle applicazioni e non solo dei dati, con migrazioni parziali (magari per area geografica) etc... questo comporta che magari due filiali devono collaborare con due applicazioni e due db diversi

Ci sono anche casi in cui non si dovrebbe replicare:

- quando ci sono frequenti update su più copie, portando le copie a possibili conflitti che devono essere scoperti e gestiti "manualmente"
- quando la consistenza è *critical* magari in contesti di trasferimento di fondi etc.... In questo caso solitamente si impone un protocollo ROWA (con transazioni *ACID compliant*), riducendo le prestazioni per avere l'autorizzazione dei *commit* da parte delle repliche

Ma spesso bisogna comunque replicare "scegliendo il male minore" (ad esempio nelle banche) e bisogna quindi analizzare il singolo caso, anche in base al budget. Inoltre non bastano le tecnologie serve un'ottima organizzazione.
Concludendo si ha che i benefici della replica sono:

- disponibilità
- affidabilità
- prestazioni
- riduzione di carico
- lavoro offline
- supporto a molti utenti

Distinguiamo anche delle classi di tipologie di replica:

- **data distribution**, di tipo *1:many*, con un *source* che distribuisce, in modo sincrono o asincrono, le varie copie passive ai *target*

- **Peer-to-Peer**, dove i vari nodi sono interconnessi e si aggiornano tra di loro. Si usa un approccio ROWA
- **data consolidation**, di tipo *many:1*, dove ho più *source* che aggiornano un *target* a livello centrale
- **bi-directional** (per il *conflict detention resolution*), dove una copia primaria e uno secondaria possono leggere e scrivere a vicenda tra loro (è quindi una versione semplificata del *Peer-to-Peer* con due Peer)
- **multi-Tier staging**, in cui si hanno meccanismi intermedi tra *source* e *target* con "aree di deposito" dette aree di *staging*

Per realizzare una replica posso fare in diversi modi:

- faccio letteralmente il backup del disco con una persona che stacca il disco dal server e lo copia, riattaccando infine copia e disco originale
- posso prima fare il backup attaccando un altro disco e poi mettere nella nuova macchina il disco copia
- posso fare una *replica incrementale*, ovvero faccio un *full backup* e sposto solo il file di *log* delle transazioni nel nuovo server, rieseguendo quanto fatto (essendo contenuto nel *log*). Quindi prima faccio un *full backup* e poi un *backup* del *log*, questo per ogni replica. Un'alternativa è l'**event publish**. Un **event publish** è una replica senza *apply* e leggo i file di log. Analizzando gli eventi tramite particolari meccanismi riscrivo quindi sull'architettura target. Da un *publisher* si passa ad un *distributor* e infine ai *subscriber*

Sulle slide *repliche* si ha un approfondimento delle architetture IBM e Microsoft opzionali per il corso.

Vengono qui aggiunte le cose dette in live.
Un **db parallelo** è studiato per le prestazioni. Si ha accesso parallelo ai dati, parallelismo *intra-query* (stessa query su frammenti diversi), parallelismo *inter-query* (tante query diverse) e sono fatti da elementi hardware posti vicini tra loro.
Parliamo di **persistenza dei dati**. Bisogna garantire la durabilità dei dati,

bisogna quindi usare un db. Si hanno anche problemi per creare oggetti persistenti a partire da un linguaggio OOP, si ha il cosiddetto l'**object-relational paradigm mismatch**. Si separa quindi l'aspetto OOP e l'aspetto relazionale per risolvere il problema per poi "mappare" l'uno nell'altro, tramite i **data mapper**. Si hanno operazioni **CRUD**:

- Create
- Read
- Update
- Delete

Si ha quindi il cosiddetto **Object–relational mapping**, ovvero questa tecnica di mapping. Si hanno vari framework che lo implementano.
Storicamente si è provato a fare db ad oggetti ma con scarsi risultati.
Parliamo ora del *caso Gitlab*. Questo è un esempio paradigmatico di cosa succede in caso di mala gestione delle repliche. Il 31 Gennaio 2017 dei malintenzionati erano entrati nel db PosgreSQL (con due repliche) e stavano scrivendo in modo significativo sul nodo primario DB1 e sul secondario DB2. I tantissimi lock si bloccavano a vicenda rendendo non disponile il db. Alle 21 prendono il db secondario e cercano di riorganizzare tutto ma il db primario aveva fatto chiamate al secondario che però non aveva risposto. I tecnici iniziano a fare danni extra, svuotando la directory dei dati del secondario ma per errore ha "droppato" il database primario, cancellando 300GB di dati dal primario. C'erano ancora i backup, fatti a *snapshot* ogni 24h. C'era un backup manuale di 6 ore prima per altri motivi. Il problema è che non sono stati trovati i backup, una volta trovati c'erano problemi a causa di incompatibilità tra PosgreSQL 9.2 e 9.6. I dati su Azure non erano backuppati per scelta di Azure, che non faceva i backup dei database. Nel complesso 5 sistemi di backup in totale non funzionavo (alcuni, tipo S3, per errori di codice erano vuoti). Fortunatamente c'era il backup manuale di 6 ore prima e si è risolto il problema ma perdendo 5000 progetti, 700 user e migliaia di righe di codice.
Bisogna premiare la trasparenza di Gitlab che ha documentato tutto quello che è accaduto (senza specificare i nomi dei DBA).

3.4 Prima esercitazione

Esercizio 1. *Si consideri un db con le seguenti relazioni (e quindi tabelle):*

- *PRODUCTION (SerialNumber, PartType, Model, Quantity, Machine)*
- *PICKUP (SerialNumber, Lot,* ***Client, SalesPerson****, Amount)*
- *CLIENT (Name, City, Address)*
- *SALESPERSON (Name, City, Address)*

(con sottolineate le chiavi primarie e in grassetto le chiavi di integrità referenziale)

e ci poniamo l'obiettivo di partizionare il db secondo determinate specifiche.
Si assume che si abbiano le seguenti specifiche organizzative:

- *si hanno 4 centri di produzione (Dublino, San Jose, Zurigo* e *Taiwan, ciascuno responsabile, rispettivamente, di cpu, keyboard, screen* e *cable) e 3 centri di vendita (San Jose, Zurigo* e *Taiwan)*
- *le vendite sono distribuite secondo le località geografiche, i clienti a Zurigo sono serviti solo dai venditori di Zurigo etc.... SI ha però che i venditori di Zurigo servono anche Dublino*
- *ogni area geografica ha il proprio db (avremo quindi 4 db)*

Vogliamo studiare una ***frammentazione orizzontale*** *delle 4 tabelle.*
Ricordiamo quindi che abbiamo 4 centri di produzione (ciascuno responsabile di un prodotto e con un db ciascuno) e 3 punti di vendita.
Partiamo con la frammentazione della relazione PRODUCTION*, ottenendo 4 tabelle, una per componente prodotto, ottenendo (con σ abbiamo l'operazione di selezione nell'algebra relazionale):*

- $PRODUCTION_1 = \sigma_{partType=cpu}(PRODUCTION)$
- $PRODUCTION_2 = \sigma_{partType=keyboard}(PRODUCTION)$
- $PRODUCTION_3 = \sigma_{partType=screen}(PRODUCTION)$
- $PRODUCTION_4 = \sigma_{partType=cable}(PRODUCTION)$

Passiamo alla relazione PICKUP. *Anche in questo caso si frammenta per il prodotto facendo il join con la tabella* PRODUCTION*(ricordando che* π *è l'operazione di proiezione/join nell'algebra relazionale). Per comodità indichiamo tutti gli attributi di* PICKUP *con pick, avendo quindi:*

$$pick = SerialNumber, Lot, Client, SalesPerson, Amount$$

indico anche con SN $SerialNumber$

- $PICKUP_1 = \pi_{pick}(\sigma_{partType=cpu}(PICKUP\ SN = SN(PRODUCTION)))$
- $PICKUP_2 = \pi_{pick}(\sigma_{partType=keyboard}(PICKUP\ SN = SN(PRODUCTION)))$
- $PICKUP_3 = \pi_{pick}(\sigma_{partType=screen}(PICKUP\ SN = SN(PRODUCTION)))$
- $PICKUP_4 = \pi_{pick}(\sigma_{partType=cable}(PICKUP\ SN = SN(PRODUCTION)))$

Prendo quindi una proiezione di tutti gli elementi di PICKUP separando nei vari PICKUP in base al prodotto.
Passo a alle tabelle SALESPERSON. *Abbiamo 3 punti di vendita, quindi, circa come per* PRODUCTION*, frammento in base alle città di vendita:*

- $SALESPERSON_1 = \sigma_{City="SanJose"}(SALESPERSON)$
- $SALESPERSON_2 = \sigma_{City="Zurigo"}(SALESPERSON)$
- $SALESPERSON_3 = \sigma_{City="Taiwan"}(SALESPERSON)$

Manca solamente CLIENT. *Anche in questo caso divido in base alle città, ricordando che Zurigo e Dublino sono clienti entrambi di Zurigo:*

- $CLIENT_1 = \sigma_{City="SanJose"}(CLIENT)$
- $CLIENT_2 = \sigma_{City="Zurigo"\ or\ City="Dublino"}(CLIENT)$
- $CLIENT_3 = \sigma_{City="Taiwan"}(CLIENT)$

Abbiamo finito la frammentazione e quindi dobbiamo solo distribuire tali tabelle:

- *le quattro tabelle con indice 1 andranno a San Jose, in* `company.sanjose.com`
- *le quattro tabelle con indice 2 andranno a Zurigo, in* `company.zurigo.com`
- *le quattro tabelle con indice 3 andranno a Taiwan, in* `company.taiwan.com`
- *le due tabelle con indice 4 andranno a Dublino (che quindi avrà solo parte di* PRODUCTION *e parte di* PICKUP *in quanto a Dublino non si ha un punto vendita), in* `company.dublino.com`

Esercizio 2. *Vediamo un esercizio in merito alla trasparenza, che ricordiamo essere a tre livelli:*

1. *di frammentazione*
2. *di replicazione/allocazione*
3. *di linguaggio*

Si chiede di fare delle interrogazioni, tenendo conto dei livelli di trasparenza, sul db costruito nell'esercizio precedente.
La prima query ci chiede di determinare la quantità dei prodotti che hanno valore "77y6878" (abbiamo quindi a che fare con la trasparenza di frammentazione, infatti interroghiamo come se avessimo a che fare con un solo db):

```
Procedure Query1(:Quan):
 Select Quantity in :Quan
 From PRODUCTION
 Where SerialNumber="77y6878"
End Procedure
```

(con `:Quan` *indichiamo il nome della tabella).*
Vediamo ora come fare nel caso di trasparenza di allocazione, quindi si sa di avere a che fare con un db distribuito. La query quindi si "sposterà" alla ricerca del giusto frammento:

```
Procedure Query2(:Quan):
 Select Quantity in :Quan
 From PRODUCTION_1
 Where SerialNumber="77y6878"

 if :empty then
 Select Quantity in :Quan
 From PRODUCTION_2
 Where SerialNumber="77y6878"

 if :empty then
 Select Quantity in :Quan
 From PRODUCTION_3
 Where SerialNumber="77y6878"

 if :empty then
```

```
 Select Quantity in :Quan
 From PRODUCTION_4
 Where SerialNumber="77y6878"
End Procedure
```

Vediamo ora come funziona per la trasparenza di linguaggio. In tal caso dobbiamo considerare sia le frammentazioni che i vari indirizzi di allocazione, ovvero i `company.città.com`*:*

```
Procedure Query3(:Quan):
 Select Quantity in :Quan
 From PRODUCTION_1@company.sanjose.com
 Where SerialNumber="77y6878"

 if :empty then
 Select Quantity in :Quan
 From PRODUCTION_2@company.zurigo.com
 Where SerialNumber="77y6878"

 if :empty then
 Select Quantity in :Quan
 From PRODUCTION_3@company.taiwan.com
 Where SerialNumber="77y6878"

 if :empty then
 Select Quantity in :Quan
 From PRODUCTION_4@company.dublino.com
 Where SerialNumber="77y6878"
End Procedure
```

Nelle slide usa `union` *al posto di* `if :empty then`*.*

Esercizio 3. *Sempre sul db del primo esercizio effettuiamo la seguente query: determinare le macchine che utilizzano come componente "keyboard" e sono vendute al cliente "Brown".*
Per praticità vediamo solo la trasparenza di frammentazione e quella di allocazione.
Partiamo con la trasparenza di frammentazione:

```
Procedure Query1(:Machine)
 Select Machine in :Machine
 From PRODUCTION join PICKUP on
 PRODUCTION.SerialNumber=PICKUP.SerialNumber
```

```
 Where PartType = "keyboard" AND Client="Brown"
End Procedure
```

Vediamo il caso di trasparenza di allocazione (e sappiamo che "keyboard" è solo in $PRODUCTION_2$ quindi interroghiamo un solo frammento e senza chiedere la specifica del partType):

```
Procedure Query2(:Machine)
 Select Machine in :Machine
 From PRODUCTION_2 join PICKUP on
 PRODUCTION_2.SerialNumber=PICKUP_2.SerialNumber
 Where Client="Brown"
End Procedure
```

Se avessimo voluto fare anche la trasparenza di linguaggio non sarebbe cambiato nulla dato che $PRODUCTION_2$ è solo a Zurigo.

Esercizio 4. *Sempre sul db del primo esercizio effettuiamo il cambiamento di indirizzo del cliente "Brown" che si sposta da "27 Church St.", Dublino, a "43 Park Hoi St.", Taiwan. Abbiamo quindi un cambio di allocazione nel db. Partiamo con la trasparenza di frammentazione:*

```
Procedure Update1
 Update Client
 Set Address  = "43 Park Hoi St.", City="Taiwan"
 Where Name="Brown"
End Procedure
```

La cosa si complica nel caso di trasparenza di allocazione, tenendo conto delle due città e dei loro db:

```
Procedure Update2
 Delete CLIENT_2
 Where Name="Brown"

 Insert into CLIENT_3 (Name, Address, City)
 values ("Brown", "43 Park Hoi St.", "Taiwan")
End Procedure
```

Se avessimo voluto fare anche la trasparenza di linguaggio non sarebbe cambiato nulla poiché le frammentazioni sono in locazioni diverse

Esercizio 5. *Sempre sul db del primo esercizio effettuiamo la seguente query: calcolare la somma di tutti gli ordini ricevuti a SanJose, Zurigo e Taiwan. Partiamo con la trasparenza di frammentazione:*

```
Procedure Query1
 Select City, sum(Amount)
 From PICKUP join SALESPERSON on
 SalesPerson = Name
 Groub by city
End Procedure
```

Passiamo alla trasparenza di allocazione. Mi serviranno tutti i frammenti di `SALESPERSON`*. Inoltre devo considerare i vari* `PICKUP`*, tutti e quattro per il discorso delle vendite su Dublino:*

```
Procedure Query2
 Create view PICKUP as
 PICKUP_1 union PICKUP_2 union PICKUP_3 union PICKUP_4

 Select City, sum(Amount)
 From SALESPERSON_1 join PICKUP on
 SalesPerson=Name
 union
 From SALESPERSON_2 join PICKUP on
 SalesPerson=Name
 union
 From SALESPERSON_3 join PICKUP on
 SalesPerson=Name
End Procedure
```

Esercizio 6. *Sempre sul db del primo esercizio cerchiamo di massimizzare il parallelismo delle inter-query.*
Prendiamo la seguente query: estrarre la somma delle quantità di produzione che sono raggruppate secondo i tipi e i modelli delle componenti.

```
Procedure Query1
 Select sum(Quantity), Model, PartType
 from PRODUCTION
 Group by (Model, PartType)
End Procedure
```

Vogliamo però massimizzare il parallelismo, divido quindi tra le varie frammentazioni:

```
Procedure Query1
 Select sum(Quantity), Model, PartType
```

```
    from PRODUCTION_1
    Group by (Model, PartType)

    Select sum(Quantity), Model, PartType
    from PRODUCTION_2
    Group by (Model, PartType)

    Select sum(Quantity), Model, PartType
    from PRODUCTION_3
    Group by (Model, PartType)

    Select sum(Quantity), Model, PartType
    from PRODUCTION_4
    Group by (Model, PartType)
End Procedure
```

massimizziamo il parallelismo inter-query se si consideriamo che ogni partizione ha un DBMS diverso. Volendo potrei dividere la query ancora a seconda del modello, evitando il `group by` *(cosa utile nel caso in cui si abbia a che fare con un sistema fortemente multicore)*

Esercizio 7. *Vediamo un esempio di db replicato che può produrre inconsistenza.*
Prendiamo l'esempio del db prodotto nel primo esercizio. Supponiamo che ogni frammento di PRODUCTION *sia allocato a tutti i DBMS. Però ogni DBMS utilizza un frammento e trasmette i cambiamenti del frammento agli altri DBMS, permettendo di avere copie del db. In caso di fallimento di un DBMS il db sarebbe comunque accessibile dagli altri sistemi e non so avrebbe problemi con le query, avendo che il fallimento è trasparente sia ai client che al db stesso. Purtroppo quando si ha un fallimento si può generare un partizionamento di rete e questo può comportare delle inconsistenze. Se, per esempio, due transazioni tolgono 800 ad una certa quantità la seconda in ordine temporale fallirà ma se avvengono in due DBMS non connessi non falliranno, producendo inconsistenza.*

Esercizio 8. *Data una replicazione simmetrica dire quando produce inconsistenza. Un esempio è un db senza il concurrency control e quindi due transazioni come quelle dell'esercizio precedente possono causare inconsistenza anche senza fallimento della rete.*

Capitolo 4

Blockchains

Questa lezione è stata tenuta dal prof. Leporati.
Nonostante qualche ambiguità ed errore di scrittura, nel capitolo si userà il termine "nodo" quando si parla di rete P2P e il termine "blocco" quando ci si riferisce ai blocchi interni alla blockchain.

Definizione 7. *Una **blockchain**, in poche parole, è un registro pubblico, condiviso e decentralizzato che memorizza la proprietà di beni digitali.*

È quindi un registro in cui vengono memorizzate informazioni relative alla proprietà di qualcosa che può essere rappresentato tramite sequenze di bit (quindi qualcosa di digitale come i *bitcoin* o altre criptovalute).
Vengono memorizzate anche le **transazioni**, ovvero i cambi di proprietà.
Essendo pubblico tutti possono vedere cosa è stato registrato e in particolare "chi possiede cosa" e la storia di una certa proprietà.
Questo registro è condiviso in quanto gestito da più persone ed è decentralizzato in quanto non esiste un nucleo che abbia di poteri da amministratore rispetto agli altri, tutti sono allo stesso livello.
Questo registro è organizzato in blocchi. Si ha un primo blocco detto **genesis block** che da il via a tutto. Si hanno poi altri blocchi, in nero, collegati a questo e che formano una catena. Un blocco si lega al successivo tramite particolari funzioni crittografiche, dette **funzioni di hash crittografico**. Nel dettaglio il collegamento tra un blocco e il successivo è dato dal fatto che il valore di hash del blocco è contenuta all'interno del blocco successivo, facendo in modo che si estremamente difficile alterare il contenuto di un blocco, ovvero diverse transazioni, che sono organizzate secondo una precisa struttura dati che consente di verificare la validità in modo veloce ed efficiente. Per ogni blocco si calcola quindi l'hash e lo si salva nel blocco successivo. Per modificare una transazione quindi dovrei calcolare l'hash e dovrei fare il check con il blocco successivo. Quindi tutti possono verificare la validità

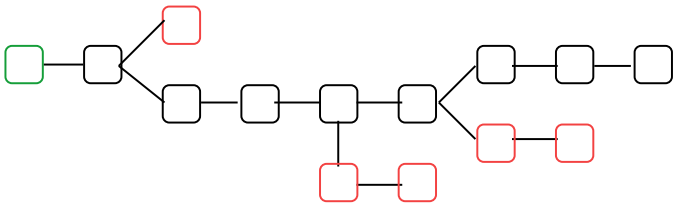


Figura 4.1: Esempio di schema di blockchain. In verde a sinistra troviamo il **genesis block**. In nero i blocchi che formano la catena. In rosso abbiamo i nodi *orfani*. Per comodità è stata rappresentata in orizzontale con la parte "alta" a destra

della blockchain. Modificare i dati in modo tale da ottenere lo stesso hash però è davvero difficile.

Si hanno vari nodi per l'uso della blockchain. I vari nodi sono nodi di una **rete Peer-to-Peer (*P2P*)** (come quelle per la condivisione di file come *torrent*, dove quindi ogni computer fa sia da *client* che da *server*). I vari nodi osservavano le proposte di transazione che vengono fatte dagli utenti (anche utenti che solo usano la blockchain), verificano che siano valide (ad esempio verificando che non avvenga il *double-spending*, ovvero, per esempio, una doppia concessione dello stesso bene), eseguono un **protocollo di consenso** (in quanto potrebbero esserci nodi "disonesti", con per esempio utenti che vorrebbero appropriarsi di beni altrui come bitcoin, si procede quindi a maggioranza secondo il *proof-of-work*) e infine procedono validando la transazione aggiungendo il blocco alla fine della blockchain ("in cima"). Una copia della blockchain viene memorizzata in ogni nodo della rete P2P alla fine della transazione, quindi quando i nodi sono d'accordo sull'aggiunta del blocco allora ciascuno lo aggiunge alla propria copia della blockchain (altrimenti qualsiasi proposta futura verrebbe bocciata).

In molte blockchain se si stabilisce che due blocchi possono essere attaccati ad un certo blocco e si inizia a lavorare su entrambi formando quindi due nuove catene. Alla fine "vince" la catena più lunga, fermando la continuazione dell'altro ramo. I blocchi del ramo "perdente" vengono resi *orfani* lasciando quindi una sola catena attiva. Le transazioni nei nodi *orfani* vengono dimenticate e bisognerà reinserirle. Questa cosa è molto inefficiente.

Ci sono reti, come quella *bitcoin*, in cui una transazione è valida sse vengono aggiunti 6 blocchi al di sopra di quello che contiene la transazione.

Le transazioni sono **pseudo-anonime** (comodo ambito bitcoin dove, come per i contanti, non si sa per quali mani sono passati quei soldi e per cosa sono stati usati prima). Ci sono vari meccanismi per rendere anonime le cose, alcuni migliori (come quelli di *monero* o *zcash*) e alcuni peggiori (come quelli

di *bitcoin* ed *ethereum*).
Si ha un ramo della *computer foreniscs* che si occupa di capire chi ha fatto certe transazioni, esplicitamente di criptovalute, sulla blockchain. Quindi si prova a raggiungere l'anonimato ma non sempre si riesce (basta un utente che paghi in *bitcoin* su un sito rivelando la propria identità).

4.1 Bitcoin

Bitcoin è una criptovaluta proposta da Satoshi Nakamoto (probabilmente un nome falso in quanto in molti stati creare valuta, anche digitale, è reato) nel 2008. Non è nato all'improvviso ma molte idee all'interno di *bitcoin* erano già presenti i diversi paper di crittografia precedenti (ad esempio la *proof-of-work* era già presente all'interno della gestione dello spam delle mail, dove veniva imposto un certo sforzo computazionale per mandare una mail in modo che l'invio non fosse completamente "gratuito", comportando che si possano mandare al massimo circa 2000 mail al giorno, al più di attrezzarsi con dei super computer). Ci sono stati tanti precedenti di tentativi di creazione di un sostituto digitale del denaro, con diverse difficoltà a causa di anonimato e *double-spending*. Inoltre rappresentare una moneta con una sequenza di bit consente la copia illimitata di tale sequenza, creando copie perfettamente identiche. Una soluzione per evitare il *double-spending* è quella di usare una *trust third party (TTP)*, ovvero tipicamente una banca che segna le transazioni monetarie, diventando però un "collo di bottiglia", venendo interpellata in tutte le transazioni. Inoltre la banca è conscia delle transazioni di denaro, che perdono così l'anonimato. L'idea di Satoshi Nakamoto e stata quella di unire vari protocolli crittografici usando al posto della banca un registro condiviso, una blockchain appunto, in cui vengono salvate le transazioni e dove vengono anche controllate, permettendo di evitare il *double-spending*. Si usa la blockchain quindi per creare un *trust*, sostituendo la funzione delle banche. Nel **genesis block** di *bitcoin* c'è infatti un messaggio (nel posto del blocco dedicato alla "causale") che fa riferimento al potere eccessivo delle banche.
Le proprietà memorizzate nella blockchain *Bitcoin* sono chiamate direttamente, come abbiamo già scritto, **bitcoins (*BTC*)** o frazioni di essi. Le transazioni sono parecchio complicate, con una struttura dati complessa, con diverse transazioni in ingresso (infatti una transazione può contenere transazioni), campi per generare nuovi bitcoin,0diversi *script di transazione* in output (scritti in un linguaggio "particolare").
Non verrà trattata nel dettaglio la conformazione delle transazioni.
Se un utente A vuole mandare un *bitcoin* ad un utente B, usando il pro-

prio client, che spesso viene chiamato **wallet**, specifica la quantità che vuole mandare e l'indirizzo di B. Ogni utente ha associato quindi un indirizzo, in qualità di lunga sequenza di numeri. Per ottenere l'indirizzo viene usata una **chiave pubblica**, usando quindi la *crittografia a chiave pubblica* in cui si usano algoritmi di cifratura e de-cifratura. Ciascuno crea con questi algoritmi una copia *chiave pubblica/chiave privata*, rende pubblica la *chiave pubblica* e comunica di usare quella chiave pubblica per risalire all'indirizzo a cui farsi spedire i *bitcoin*, infatti data la chiave pubblica all'utente che deve inviare i bitcoin la userà per cifrare il messaggio in modo che, tramite la chiave privata, solo il legittimo destinatario possa decifrare il messaggio.
Dalla chiave pubblica quindi ottiene l'indirizzo al quale A deve mandare i soldi per farli ricevere a B. Quindi l'indirizzo è una sorta di identità per B che però può generarsi tutte le coppie di chiavi che vuole, combinando poi i vari bitcoin ricevuti ai vari indirizzi che ha generato tramite la complessa struttura della transazione. Questa possibilità di generare infinite chiavi però è solo uno *pseudo-anonimato*.
Bisogna però anche dimostrare che A è proprietario dei *bitcoins* che vuole inviare a B. Per farlo A "firma" digitalmente la transazione tramite la sua *chiave segreta*. I nodi della rete P2P, che sono detti **miners**, verificano che la firma di A sia valida, verificano che non ci sia il *double-spending* e infine validano la transazione mettendola in un nuovo blocco della blockchain. Se B, che ha ricevuto i *bitcoins*, vuole a sua volta mandarli a C avvia una transazione esattamente come descritto sopra, firmando con a chiave privata che era accoppiata alla chiave pubblica con la quale A gli aveva mandato i *bitcoins*, permettendo che la firma sia verificata e validata. Quindi sulla blockchain il possedimento di un *bitcoin* è rappresentato dal fatto che si può inviare una transazione per cui quel *bitcoin* può essere dato a qualcun altro (ovviamente si è usato *bitcoin* ma si poteva parlare anche di più *bitcoins* o, vedremo in seguito, frazioni, che molto piccole, di esso). Non è segnato da nessuna parte quanti *bitcoins* possiede un certo utente ma solo le catene di transazioni che sono state fatte da ciascun *bitcoin*, vedendo quindi chi è l'ultimo proprietario. È quindi essenziale memorizzare le chiavi in quanto possedere equivale a conoscere una chiave segreta.
La blockchain registra ogni singola transazione (e sono circa un migliaio ogni 10 minuti, con un blocco che riesce a contenere circa un migliaio di transazioni e i blocchi vengono aggiunti uno ogni 10 minuti circa) e attualmente, in data 19 Ottobre 2020, si è arrivati a 290GB di blockchain.

4.1.1 Miners

Abbiamo parlato prima dei membri della rete P2P della blockchain *Bitcoin*, detti appunto **miners**.
I *miners* lavorano su un *pool* di transazioni proposte dai vari *wallet*. I miners scelgono circa un migliaio di queste transazioni alla volta e cercano di formare il nuovo blocco, validando le transazioni. Effettuano quindi i calcoli computazionali del *proof-of-work* per cui dimostrano di aver fatto un certo sforzo per avere il **diritto** di essere quelli che aggiungono il prossimo blocco alla catena. Avendo un'aggiunta ogni 10 minuti si ha una fortissima concorrenza tra i *miners*. Inoltre il carico di lavoro richiesto diventa sempre più difficile in quanto, grazie alla **legge di Moore**, diventa sempre più facile risolvere il "puzzle" crittografico, per il *proof-of-work,* che serve a risolvere il nuovo blocco e quindi il "puzzle" viene reso sempre più difficile (ovvero se un blocco arriva in meno di 10 minuti il blocco successivo sarà più difficile da produrre, in modo che nuovamente servano almeno 10 minuti, anche se equivalentemente verrà reso più facile se ci vogliono troppi minuti in più di 10, avendo così **autoregolazione**). Bisogna quindi parlare di questo "problema" crittografico da risolvere e per farlo bisogna un attimo specificare meglio le *funzioni di hash*.

Definizione 8. *Le **funzioni di hash** sono funzioni crittografiche che prendono in input una sequenza di bit, in teoria arbitrariamente lunga, anche se ogni funzione ha un limite teorico (ma praticamente irraggiungibile), e produce una sequenza che vorrebbe essere univoca (ma che non lo è) di poche centinaia di bit. Per esempio SHA1 produce in output una sequenza di 160 bit, MD5 di 128 bit, SHA256 di 256 bit (una di quelle usate in Bitcoin), RIPEMD di 160 bit (anch'essa usata in Bitcoin) etc...*
Le funzioni di hash devono essere sufficientemente facili da calcolare a partire da un certo input

L'idea è quindi è che prendo un file e, usando ad esempio SHA256, mi esce una sequenza di 256 bit, univoca per quel file. Purtroppo il dominio della funzione (ovvero i bit del file) è molto più grande del codominio (ovvero tutte le possibili sequenze di 256 bit) e quindi non si può avere davvero una **funzione iniettiva** e quindi si avranno sempre due sequenze in input che producono lo stesso output e quando questo avviene si ha una **collisione**. Le collisioni sono inevitabili ma le funzioni di hash sono fatte in modo tale che sia estremamente difficile trovare due input diversi che producano lo stesso output, motivo per cui si può anche pensare che le hash siano univoche. Si ha anche che è estremamente difficile cercare esplicitamente un input che abbai lo stesso hash di un altro, rendendo quindi molto difficile sostituire un

input dato con un altro ottenendo comunque lo stesso output, imbrogliando. Quest'ultimo problema è più difficile di quello della *collisione* (dove ho, nella pratica, un grado di liberà in più avendo due input).
Si hanno altre due proprietà:

1. dato un hash è estremamente difficile trovare un input, per questo si dice che la funzione di hash è *one-way* (essendo molto facile da calcolare ma difficilissimo da invertire)

2. anche se cambio un solo bit dell'input ottengo un output completamente diverso a quello ottenuto prima del cambiamento, rendendo impossibile capire la regola di calcolo o anche solo fare indagini statistiche. Infatti hash significa anche "polpettone", cosa che rappresenta bene l'azione di spezzettamento, calcolo e rimescolamento (con anche valori semi casuali) dell'input fatte dalle funzioni per calcolare l'output

Quindi nel *proof-of-work*, per dimostrare di aver svolto una certa quantità di lavoro viene preso il dato di cui devo calcolare l'hash, gli viene aggiunta una quantità casuale detta **nonce (si pronuncia "nons")** e calcolo l'hash del dato concatenato al *nonce* e vado a vedere se il risultato ha un certo numero prefissato di bit più significativi uguali a 0, riduco quindi il codominio, ovvero il numero di possibili output validi, dicendo che devono cominciare con un certo numero di zeri. Aumentando il numero di zeri riduco la probabilità di ottenere un risultato valido scegliendo un *nonce* a caso (e diminuendo ottengo l'opposto). Variando gli zeri ottengo quanto detto sopra in merito al variare del carico computazionale per restare sui 10 minuti.
Quindi, ricapitolando:

- il miner sceglie un migliaio di transazioni più o meno a caso
- il miner spara a caso un valore del *nonce*
- calcola l'hash di tutto il blocco:
  - se ottiene un valore con un numero di bit più significativi uguali a zero accettabile, prima degli altri, manda il blocco nella rete P2P per far validare il nuovo blocco e, se il controllo viene superato, il blocco viene accettato e messo in cima alla blockchain (quindi ogni nodo della P2P lo attacca in cima alla propria copia)
  - se non ottiene tale valore spara a caso un altro *nonce* e ci riprova

Se mentre si sta lavorando un nuovo blocco viene validato (anche dal nodo che stava lavorando al nuovo blocco) un altro blocco bisogna "buttare" quanto costruito anche se non tutto, butto infatti le transazioni che già compaiono nel nuovo blocco convalidato. Inoltre cambio l'hash del nuovo blocco a cui sto lavorando mettendo quello di quello appena convalidato (per il discorso spiegato all'inizio del capitolo).
L'importanza di essere colui che crea il blocco è data dal fatto che l'unico momento in cui si possono creare *bitcoins* è durante la creazione del nuovo blocco, chi aggiunge il blocco ha dei nuovi *bitcoins* che vengono creati insieme al blocco.
Il mining può essere fatto da macchine sempre più performanti (all'inizio bastava un portatile, poi si è passati ad usare i pc di interi uffici di notte (o qualche furbo anche, illegalmente, dell'università), poi si è passati a cluster "home-made" tramite hardware spesso dedicato solo al mining, mentre ora servono cluster estremamente potenti ma che comunque facciano ritornare le spese). Si hanno anche soluzioni di sub-affitto di hardware o di gente che condivide l'hardware per poi dividere i guadagni. A causa di queste *farm* professionali enormi, spesso collocate in paesi freddi per il risparmio del raffreddamento e dove la corrente costa meno (ovvero in zone disabitate di paesi spesso poveri e poco democratici), rischia di danneggiare l'idea di decentralizzazione della blockchain. **Bitcoin** ad un certo punto era in mano di pochissime persone (cinesi) con farm sparse nei monti asiatici. Si rischiano anche manipolazioni truffaldine del mercato, ad esempio proposte di *trading coi bitcoins* (cose comunque vietate nei mercati regolamentati ma non su quello delle criptovalute, non essendo regolamentato, e quindi se il broker di imbroglia sei fregato).
Un altro problema dell'*accentramento* è il cosiddetto **attacco del 51%**. Dato che ogni miner deve vincere contro tutti gli altri per poter essere quello che ha diritto di mettere il nuovo blocco allora se uno riesce a possedere il 51% dell potenza di calcolo dei miners, controllando il 51% dei nodi della rete P2P, praticamente vince sempre, validando sempre le transazioni, anche se non valide (ovviamente almeno il 51%).
"Tutti contro tutti" è anche molto inefficiente dal punto di vista energetico (la rete P2P che gestisce *bitcoin* consuma più di tutta l'Argentina). Per questo si cercano sempre nuovi algoritmi/alternative per il *proof-of-work*, come ad esempio *proof-of-stake*, dove stake sta per "quantità di soldi", ovvero l'idea in cui solo pochi nodi della P2P fanno il mining. Tali nodi vengono scelti in base alla quantità di soldi che ogni nodo ha deciso di rendere disponibili agli altri (su un conto speciale). Quindi chi mette più soldi ha più possibilità di essere scelto.
Al massimo si ha che si potranno costruire 21 milioni di *bitcoins* e questi sono

sempre più difficili da creare (è quindi paragonabile all'oro da un certo punto di vista).
Torniamo a parlare delle transazioni.
Nel caso in cui A voglia dare a B una certa porzione di *bitcoins* deve creare due transazioni:

- una in cui dal totale produce la frazione complementare a quella che deve dare a B, questa transazione è verso se stessi stessi
- una in cui da a B la frazione voluta

In poche parole se A ha 10 *bitcoins* deve fare una transazione a se stessa di 9 e una di 1 a B.
Questo può essere fatto anche avendo più indirizzi di partenza in cui si hanno i *bitcoins*, grazie al fatto che una transazione ha più input, e verso più destinatari, contemporaneamente, in quanto la transazione ammette anche più output nella sua struttura. **La somma totale degli input non deve essere minore a quella in output** (infatti la transazione non verrà validata). Può comunque essere maggiore in quanto la differenza, detta *fee*, può essere un compenso per il miner per far scegliere quella transazione da validare (che quindi non sceglie proprio a caso il migliaio di transazioni in quanto ordina le transazioni in base alle *fee*). Queste *fee* saranno l'unica fonte di guadagno per i miner quando non si potrà più produrre *bitcoins* per il limite visto precedentemente (in quel momento le *fee* aumenteranno di valore probabilmente).
Due blocchi possono essere aggiunti contemporaneamente perché magari due miner in luoghi distanti propagano insieme l'avviso di aver trovato un nuovo blocco valido. Può succedere quindi che vengano aggiunti entrambi anche se uno è destinato a diventare *orfano*.
Proporre una transazione con *fee* pari a 0 può portare al fatto che nessun miner la prenda in carico, comportando la creazione di *transazioni zombie* (che sono parecchie).
Un altro problema di *bitcoin* è che per essere sicuri che una transazione si andata a buon fine devo aspettare 6 blocchi, quindi un'ora di tempo, e questo limita i casi d'uso della moneta (di certo non va bene per prendere il caffè). Il numero di transazioni supportate è comunque minimo rispetto a quelle che si possono effettuare con la moneta fisica.

4.2 Ethereum

Il futuro delle blockchains non ha potenzialmente limiti. Ci sono molte applicazioni e molte varianti, una di queste è **Ethereum**.

Listing 1 Esempio di contratto in Solidity tratto dalla documentazione di Solidity `https://solidity.readthedocs.io/en/v0.7.4/`

```
pragma solidity >=0.4.16 <0.8.0;

contract SimpleStorage {
  uint storedData;

  function set(uint x) public {
    storedData = x;
  }

  function get() public view returns (uint) {
    return storedData;
  }
}
```

Ethereum è un altro tipo di blockchain in cui si pone l'attenzione non tanto sulle transazioni quanto sulle **computazioni**. Si ha un modello diverso di blockchain in quanto vengono memorizzate i cosiddetti *ethereum account*. Ogni utente che si registra quindi viene quindi associato ad un account con una certa quantità di criptovaluta, che in questo caso è chiamata **ether (*ETH*)**. Le transazioni avvengono più o meno come in un conto in banca, togliendo *ether* ad uno e dandoli all'altro mentre la validazione delle transazioni avviene quasi come in *bitcoin* anche se si sta cercando di passare alla *proof-of-stake*. La novità di *ethereum* è la possibilità di scrivere/programmare i cosiddetti **smart contract**, ovvero il corrispettivo dei contratti tra persone, dove, per esempio, un utente, per un tot guadagno al mese, concede l'uso di una sua proprietà ad un altro utente. Si usa un linguaggio di programmazione chiamato **Solidity**, simile ad un linguaggio OOP dove al posto degli oggetti si hanno i **contratti** (ma si hanno comunque struct, function etc...). I contratti scritti con Solidity vengono compilati in un bytecode che viene memorizzato sulla blockchain di ethereum (e quindi una copia viene salvata su tutti i nodi della rete P2P). Si possono fare transazioni che trasferiscono soldi oppure posso fare transazioni in cui si invocano funzioni poste all'interno di un certo contratto, in questo caso un miner raccoglierà la richiesta ed eseguirà sulla sua macchina il codice del contratto. Ogni esecuzione di ogni singola operazione del bytecode (che viene eseguito sulla *ethereum virtual machine*) costa una certa quantità di **gas** e quindi bisogna dare una certa quantità di soldi, rappresentati il costo del **gas**, per far eseguire il con-

tratto e, qualora non siano sufficienti, viene sollevata l'eccezione `outOfGas`, il contratto non va a buon fine e i soldi finora spesi vanno persi.
Gli *smart contract* sono codice di cui chiunque può vedere il bytecode e sono una sorta di codice lato server e posso quindi fare dei client che gestiscono le parti "delicate" dei trasferimenti di soldi facendo chiamate ai contratti, che sono elaborati dai miner. La scrittura dei contratti è rischiosa quindi si hanno quindi delle comunità (come *OpenZeppelin*) che controllano i contratti stessi e forniscono delle linee guida e delle librerie da cui attingere, ma nel momento in cui vengono modificati non sono più testati, ovviamente. Un esempio è stato di una startup che ha lasciato una moltiplicazione non protetta da overflow modificando un contratto, preso da *Openzeppelin*, per permettere pagamenti simultanei (banalmente se si doveva dare 10 ETH a due utenti si faceva 10·2 e si controllava che ci fossero effettivamente 20 ETH nel conto prima di effettuare il doppio pagamento). Essendo il bytecode pubblico se ne sono accorti e qualcuno si è fatto un pagamento a due suoi altri account dando una cifra esorbitante, in modo da scatenare l'overflow, il prodotto per 2 dava 0 a causa dell'overflow e quindi si autorizzava la transazione. Questa cosa non sarebbe stata possibile su *bitcoin* ma la diversa implementazione della blockchain di ethereum rendevano possibile la cosa (e irriconoscibile al miner). Creare quindi *smart contract* è estremamente difficile, dovendo essere estremamente sicuri. Inoltre una volta che il contratto è sulla blockchain ci resta, anche se il creatore può chiedere al contratto stesso di autodistruggersi (nel senso che vengono rese invalide le chiamate alle funzioni di quel contratto).
Ci sono vari strumenti per scrivere contratti, l'ide *Remix* e *MetaMask*, che permette di avere un **wallet** su ethereum e di collegarsi sia alla rete principale che a diverse reti, anche locali sul proprio computer, per testing. Tra le reti di testing principali abbiamo *Rapsten* che cerca di emulare al meglio possibile quella principale (dove gli ETH che trasferisco o pago come **gas** sono finti). In altre reti di test si hanno altri algoritmi di consenso, come per esempio, nel caso della rete *Rinkeby* si ha il **proof-of-autority** (dove chi ha diritto a scrivere una certa informazione lo può fare senza il consenso della rete P2P).
L'iter normale di sviluppo di contratti non prevede in primis l'uso delle reti di test in quanto servirebbe troppo tempo ma si usano strumenti come *Ganache* o *Truffle* che sono strumenti di sviluppo in locale che simulano una rete in locale dove si può testare senza **gas** in modo veloce. Solo dopo si passa alla rete di test e poi alla *main net*, la rete principale.
Esiste anche una libreria chiamata *web3*, scritta in *js*, per permettere alle webapp di connettersi a MetaMask.

4.3 Altre blockchains

Si hanno comunque davvero tante diverse blockchains.
Si ha, ad esempio, *Quorum*, una variante di *Ethereum* in cui si possono usare altri algoritmi di consenso e in cui si possono anche creare dei canali di comunicazione privati tra i nodi della rete P2P.
Si hanno quindi diversi tipi di blockchain e in particolare ci sono le blockchain di tipo **permissioned** e non solo quelle di tipo **permissionless** o **pubbliche**.
per blockchain di tipo *permissioned* si intende che i nodi della rete P2P, che tipicamente sono anche quelli che scrivono informazioni e quindi avviare le transazioni, si conoscono ma eventualmente non si fidano, formando così un **consorzio** in cui i partecipanti sono potenzialmente in conflitto (dove magari il fallimento di uno è il successo dell'altro). Se le dichiarazione pubbliche siano vere o meno viene deciso con un servizio di *audit* esterno che valuta le dichiarazioni pubbliche. Non c'è quindi ne mining ne *proof-of-work*, magari nemmeno una criptovaluta. La blockchain diventa quindi un punto in cui implementare politiche di trust tra i partecipanti, obbligandosi a vicenda a dichiarare in modo pubblico (e se imbrogli esci dal consorzio perdendone i vantaggi), fattore che viene aiutato dal fatto che è estremamente difficile alterare la blockchain (dovendo rompere una catena e rifarla, imbrogliando sugli hash etc...) rendendo le dichiarazioni praticamente eterne.
Si hanno quindi:

- **permissionless blockchain**, come *bitcoin* o *ethereum*, dove chiunque può scaricarsi il software e diventare miner
- **permissioned blockchain pubblica**, dove possono vedere tutti, come *Quorum*, *EOS* (con una virtual machine basata su *webassembly* e *smart contract* scritti in *C++*), *Hyperledger* (sviluppata da IBM e ospitata dalla *Linux Foundation*, essendo completamente *Open Source*, di tipo modulare, molto complessa ma efficiente, nata per il business con algoritmi di consenso basato sul **problema dei generali bizantini** e che scrive sulla blockchain solo se necessario etc...) etc... Con queste blockchain si perde il discorso della decentralizzazione, non permettendo che partecipi chiunque ma si ha una *certificate authority* che stabiliscono se uno ha il diritto di scrivere o leggere sula blockchain (e si hanno spesso più amministratori che si controllano a vicenda atti a stabilire chi può fare cosa)
- **permissioned blockchain privata**, dove possono vedere solo quelli che scrivono

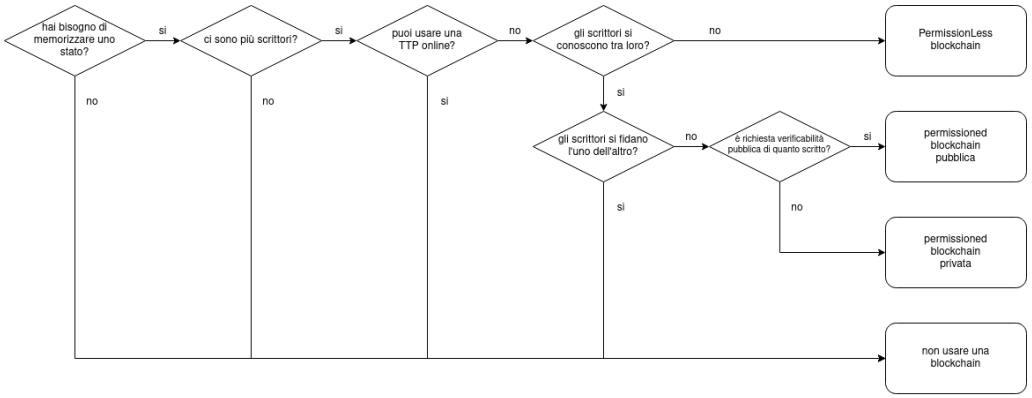


Figura 4.2: Diagramma dei casi di scelta di una blockchain

Grandi aziende, come IBM, Microsoft, SAP etc... che stanno concentrando sullo sviluppo di blockchain ma anche piccole applicazioni, come *CryptoKitties*, basata su Ethereum, che permette di collezionare e crescere "gattini digitali" collezzionabili.
Bisognerebbe inoltre fare un discorso sul rapporto che si ha tra *smart contract* e dispositivi fisici, nonché sull'interpretazione legale degli stessi (e gli avvocati dicono che non sono ne legali ne "smart" per i problemi detti sopra).

Vengono qui aggiunte le cose dette in live.
La blockchain potrebbe essere usata per la tracciabilità di prodotti, per evitare *falsi veri* o *veri falsi*. Infatti non è un problema lontano dal **double-spending** (un prodotto viene fatto in numero limitato e quindi non posso avere più di quel numero sul mercato). Per il tracciamento del prodotto associo un hash-code univoco. Metodi come il bar-code non funzionano bene in quanto facilmente duplicabili. Un metodo più moderno prevede l'uso proprio delle blockchain, introducendo anche la tracciabilità dei documenti, permettendo anche di verificare tutta la filiera per arrivare al prodotto finale. Ogni step della filiera è come se avesse un file di log, dove viene segnato tutto quello che viene fatto in quello step. Si deve anche garantire che nessuno step della filiera "bari". Inoltre non si può avere una TTP online (i vari consorzi, gli unici che potrebbero fare da TTP, potrebbero anche loro "barare", non essendo un ente che farebbe solo da TTP) e si ha che gli scrittori "non si fidano" tra loro. Abbiamo quindi tutti i requisiti per una blockchain, come spiegato immagine 4.2. Mantenere una blockchain comunque costa (più di un ipotetico timbro univoco da apporre al prodotto, tramite i concetti di *lotto di produzione*) ma ha più senso in un contesto multi-aziendale.

Ovviamente si ha anche la difficoltà culturale di far interagire piccole imprese con concetti complessi come la *proof-of-work*. Un ragionamento simile si applica sia al mondo del cibo che della moda, che ad altri ambiti.
Il tracciamento dei clienti non è fattibile causa GDPR, in quanto non si può facilmente cancellare i dati (cosa garantita dalla GDPR) nella blockchain, che infatti non è *GDPR-compatible* (e potrei non inserire i dati in chiaro ma cifrati, dimenticando la chiave nel caso il cliente voglia essere dimenticato, in quanto potenzialmente prima o poi sarebbe facile "bucare" i dati).

Capitolo 5

NoSQL

Dopo aver trattato la gestione dei dati distribuita in **ambiente relazionale**, dopo aver trattato le blockchain anche da punto di vista di gestione dati in **ambiente sicuro**, affrontiamo i sistemi **NoSQL**.
Nel 1970 un ricercatore di IBM chiamato Codd pubblica un articolo intitolato *A relational model of data for large shared data banks* in cui presenta un modello di rappresentazione dati indipendente dall'implementazione. Si era passati dal **modello gerarchico** (in cui bisognava usare i puntatori per accedere ai dati) al **modello relazionale**. Questo passaggio è stato fondamentale per lo sviluppo del mondo dell'informatica, un mondo dove l'hardware dedicato allo storage era estremamente costoso, estremamente poco capiente (nel range di pochissimi megabyte) e logisticamente parecchio ingombrante. Il modello relazionale quindi era studiato in primis per questo contesto, proponendo strategie per la rappresentazione compatta dei dati.
Gli aspetti positivi del modello relazionale si possono comunque riassumere in:

- è un modello molto ben definito, con il **principio della minimizzazione**, in merito allo spazio, e il **principio della closed world assumption**, ovvero tutto quello che l'applicazione deve sapere è nel database e a tal fine si ha anche l'uso del `NULL` *value* per:
  - assenza di informazione sul valore
  - assenza di senso di un certo dato
  - assenza di applicabilità di un certo dato

  Quindi il modello relazionale è una sorta di *modello chiuso* che contiene tutto il necessario ed era un modello ragionevole in un

contesto come quello "pre anni 2000", dove un'azienda tipicamente necessitava solo di informazioni interne all'azienda stessa (non c'erano le informazioni dei *social network* etc... infatti, di fatto, non esiste più la *closed world assumption*)

- si hanno circa 35 e più anni di sviluppo su *sicurezza, ottimizzazione* e *standardizzazione*. Da questo punto di vista si ricorda l'uso delle proprietà ACID che comunque rendono ancora valido il modello anche nel 2020. Inoltre ormai si hanno talmente tante informazioni memorizzate tramite il *modello relazionale* che le aziende sono restie a cambiare modello (a causa dei rischi di perdita dei dati durante un porting, anche dati dal fatto che i db preesistenti sono enormi, considerando che i dati persi non sono recuperabili potenzialmente, a meno di repliche)
- è ben conosciuto, è studiato anche nelle scuole e nelle università
- è comunque la scelta migliore per diversi casi d'uso

Si hanno però anche delle limitazioni per il modello relazione, come ad esempio:

- il fatto stesso di essere un **sistema chiuso**, quasi per gli stessi motivi descritti nei vantaggi, può anche essere un aspetto negativo, infatti:
  - il **principio di minimizzazione**, ai giorni nostri, va a discapito, inutilmente, delle performance
  - il **principio della closed world assumption**, come già detto, non è praticamente più valido nell'era dei social network etc...
  - il modello relazionale prevede per un attributo uno e un solo valore e non sono quindi ammissibili array o comunque altre strutture per dati con multipli valori e non si possono fare interrogazioni in modo strutturato all'interno dei valori stessi, dovendo ricorrere a diversi *workaround* e, secondo "il manuale", il metodo standard è quello di fare una tabella di join, detta *tabella di part-of*, ovvero una tabella a parte coi possibili multi valori (questa soluzione porta ad operazioni di join molto dispendiose)

  - non è compatibile con i linguaggi di programmazione moderni. Molti linguaggi moderni sono infatti OOP, con la *object persistency*, paradigma non supportato dal modello relazionale. Si hanno quindi framework (come *Spring*) per far comunicare, tramite l'**object relational mapping** il modello relazionale e gli oggetti della OOP (aggiungendo complicazioni e perdite di performance). Database ad oggetti sono stati creati e implementati nei principali vendors ma non sono usati nelle aziende per varie ragioni, anche di performance e di impatto rispetto ai db relazionali
  - non supportano i **comportamenti ciclici**, comportando limiti modellistici

- i database relazionali sono difficili da modificare dal punto di vista schematico tramite `alter table`. Cambiare gli schemi è complesso e comporta il fermo del database

- per garantire le proprietà ACID nei sistemi transazionali e per poter garantire il 2PC si ha un limite fisico alla **scalabilità**. Dopo un certo punto di crescita i costi di organizzazione rendono impossibile l'esecuzione dei protocolli che garantiscono ACID, in ambiente distribuito. Oggigiorno ci sono varie applicazioni, legate in primis ai social network, per le quali è impossibile mantenere certi protocolli. Il problema della scalabilità è uno degli aspetti più limitanti, specialmente in ottica **big data** (anche se vedremo che le alternative non relazionali coprono casi d'uso anche oltre il solo ambito dei *big data*). I database relazionali si prestano bene alle **scale up (*scalabilità verticale*)** (semplicemente aumentando l'hardware a disposizione, arrivando a costi esorbitanti e fino al punto in cui si raggiunge il limite tecnologico) ma pochissimo allo **scale out (*scalabilità orizzontale*)** (che comporterebbe non l'acquisto di un nuovo hardware intero ma solo di una parte di esso, a prezzo più basso con hardware detto in gergo **comodity**, magari per avere più dischi etc..., ma hardware dedicato storicamente ai db relazionali non prevede questa cosa). Oggigiorno si hanno sempre più spesso le **architetture a microservizi** (che "implementano" bene lo *scale out*)

- il **time in market**, ovvero il tempo in cui un applicativo resta in commercio, è sempre più ridotto (spesso, per esempio, un videogioco ha una vita anche inferiore ad un anno) portando a rendere

obsoleto il processo in cui normalmente si integrano i db relazionali. Anche il *time to market*, ovvero il tempo di preparazione si è ridotto

Oggigiorno lo standard dal punto di vista hardware è drasticamente cambiato con dischi di capienza enorme (nell'ordine dei terabyte per qualche decina di euro). Il rapporto costo-gigabyte è crollato rispetto a pochi anni fa. Il contesto storico è quindi cambiato e in tal senso anche il modello per la rappresentazione dei dati sta cambiando e sta cambiando anche la scelta tra *spazio* e *tempo*, non essendo più la prima una grossa problematica ed essendo la seconda di grande interesse, dovendo rispondere il più in fretta possibile alle interrogazioni.
Sulle slide un elenco dei vari step storici.

5.1 I modelli NoSQL

Il termine **NoSQL** è stato coniato da Carlo Strozzi nel 1998 quando si inventa una sorta di API per Linux per accedere ai dati relazionali senza usare SQL. Il termine è stato ripreso nel 2009 con la "logica" **Not Only SQL** dopo che per circa 9 anni, a partire dal 2000, erano nati nuovi modelli, a grafi, a documenti etc... (da parte di Amazon, Google, Facebook etc...).
Quindi NoSQL è un'insieme di modelli di rappresentazione dati, con relativi software di gestione. Si hanno 3 caratteristiche fondamentali:

1. tutti i modello sono **schema free** o **schemaless**
2. tutti i **DBMS NoSQL** usano il **CAP theorem**
3. si passa da ACID a BASE (da "acido" a "basico") ovvero:
    - *Basic Available*
    - *Soft state*
    - *Eventual consistency*

Schema free

Nel modello relazionale prima veniva definito il modello, ovvero l'insieme degli attributi che descrivevano i dati e le varie relazioni, e poi veniva popolato con i dati veri e propri. Questo iter comportava che se durante lo sviluppo ci si accorgeva che bisognava aggiungere, ad esempio, un attributo bisognava modificare l'intero schema (nonché i dati già caricati).

Con NoSQL si è quindi passati quindi ad un'architettura dove il modello è basato sui dati che vengono inseriti quindi di fatto per ogni "riga" ho potenzialmente uno schema diverso. Questo concetto, detto **schema free** concede una grandissima libertà di sviluppo, al costo di un'alta probabilità di avere inconsistenza tra i dati. Si ha il cosiddetto **open word assumption**, dove i valori `NULL` vengono eventualmente messi solo per motivi applicativi in quanto se un dato non c'è semplicemente non c'è e non bisogna specificare non `NULL` il fatto che non ci sia, garantendo una grande flessibilità (anche se questo può generare problemi). Tale flessibilità ha comunque un costo anche dal punto di vista delle query, in quanto non si può partire dallo schema ma si deve partire da frammenti del modello in studio. Si ha comunque il guadagno che in caso di aggiunta di attributi etc... non bisogna modificare l'intero schema, aggiungo dove serve e basta, anche se ovviamente serve una *governance* del dato più forte (in quanto può anche succedere che non si abbia il tipo di dato).

CAP theorem

Vediamo questo teorema essenziale[1], proposto da Gilber e Lynch nel 2002 dopo che, qualche anno prima, nel 2000, Brewer aveva proposto una sua *congettura sulla consistenza, disponibilità e partizionamento dei webserver*. Il teorema dimostra la congettura di Brewer:

Teorema 1 (CAP theorem)**.** *Preso un* ***sistema distribuito di nodi*** *si considerino tre aspetti:*

- ***Consistency (consistenza)****, ovvero che tutti i nodi abbaino gli stessi i dati nello stesso istante (che è diverso dalla consistenza di ACID)*
- ***Availability (disponibilità)****, ovvero la garanzia che ogni richiesta ricevuta da un sistema riceva una risposta in ogni caso, sia in caso fallimento che successo (non si deve avere quindi un'attesa infinita)*
- ***Partition tolerance (partizionamento)****, ovvero il sistema deve continuare ad operare comunque anche se si hanno perdite di messaggi o fallimenti di parti del sistema*

Il teorema garantisce che tra queste tre caratteristiche, in un sistema distribuito, se ne possano avere contemporaneamente due soddisfatte e mai tre.

[1] così L.T. è contento

Si nota quindi come, prendendo ad esempio i DDBMS, si rinunci alla *partition tolerance*, garantendo le prime due caratteristiche (anche se con un numero limitato di nodi si può arrivare ad una buona resa anche dal punto di vista della *partition tolerance*).
I sistemi NoSQL tipicamente divisi tra:

- sistemi che garantiscono *consistency* e *partition tolerance* (CP), non potendo garantire che il DBMS lavori 24/7. Tra questi troviamo: *BigTable, Hypertable, HBase, MongoDB, Terrastore, Scalaris, Berkeley DB, MemcacheDB, Redis ...*
- sistemi che garantiscono *availability* e *partition tolerance* (AP), non potendo garantire che i dati siano sempre consistenti. Tra questi troviamo: *Dynamo, Voldemort, Tokyo Cabinet, KAI, Cassandra, SimpleDB, CouchDB Riak ...*

Non è detto che due db NoSQL che implementano lo stesso modello, ad esempio documentale, garantiscano le stesse due caratteristiche CAP.
Quindi in fase progettuale la scelta tra due delle caratteristiche CAP è un punto chiave per poi scegliere eventualmente su che DBMS NoSQL appoggiarsi (o anche partire da un DBMS NoSQL che già si conosce ma essendo consci della caratteristica non garantita).
Ci sono contesti applicativi in cui ognuna delle tre caratteristiche può, a seconda, non essere fondamentale. In ogni caso si ha che:

- *consistency* mi garantisce che tutti i client hanno sempre la stessa vista sui dati
- *availability* mi garantisce che ogni client può sempre leggere e/o scrivere
- *partition tolerance* mi garantisce che il sistema continui a funzionare nonostante partizioni fisiche della rete

In fase progettuale quindi ho 3 step:

- scelta del modello
- scelta del DBMS
- scelta delle caratteristiche del CAP theorem

BASE

Analizziamo meglio l'acronimo.

- *Basic Available*, ovvero, comunque vada, la risposta viene sempre data anche nel caso di consistenza parziale
- *Soft state*, ovvero si rinuncia al concetto di consistenza di ACID
- *Eventual consistency*, ovvero, ad un certo punto prima o poi nel futuro (magari anche dopo pochissimi secondi), i dati "convergeranno" ad uno stato di consistenza (in opposizione alla consistenza immediata di ACID)

Vediamo quindi i vari modelli NoSQL:

- **key-value stores** (come *Dynamo, Voldemort, Redis, Riak ldots*) che è il modello più semplice. È infatti il semplice modello *chiave-valore*, semplice da memorizzare, e utile per il caching. Il valore è un *blob* quindi non si ha modo di interrogare internamente il valore me deve essere l'applicazione a manipolarlo. La chiave garantisce tipicamente meccanismi di hash, con la chiave che punta ad un certo valore, massimizzando le performance. Viene garantito l'*hashing distribuito*, ovvero la *distributed hashed table*
- **column family stores**, (come *BigTable, Cassandra, HBase* ...) dove la chiave punta a colonne multiple di valori, ottenendo un formato *pseudo-relazione*. Ogni riga, in questo schema **wide column**, ha una sua struttura interna diversa a quella di altre righe, che sono comunque indipendenti. La chiave ovviamente può essere hashata. Non posso fare *ranged query*
- **document databases** (come *CouchDB, MongoDB* ...) è formato da documenti con coppie chiave-valore o chiave-oggetto, dove l'oggetto può anche essere un'altra chiave-valore, un array di valori, un array di oggetti etc.... Un esempio pratico di documento è il tipico file json, che è ottimo per dire che nel modello documentale si ha un elemento più importante, detto *root* che ha al di sotto tutti gli altri
- **graph databases** (come *Neo4J, FlockDB, GraphBase, InfoGrip* ...) con nodi e relazioni tra gli stessi rappresentate dagli archi. I nodi possono rappresentare entità a cui sono associate varie proprietà (e si chiamano *property graph*). le proprietà sono quindi pertinenti ai nodi. I graph db non scalano bene

- **RDF databases**

(*Spesso l'azienda stessa non è attenta al tipo di modello usato quindi si hanno spesso confusioni*).
A livello di complessità (anche dei dati che rappresentano) possiamo dire che, seguendo l'ordine elencato, si passa dal più semplice al più complesso. Anche dal punto di vista del volume abbiamo lo stesso ordine, dal più voluminoso (quindi il meno complesso è anche il più voluminoso ma più scalabile) a quello meno voluminoso e scalabile.
Il problema fondamentale in tutti questi modelli è come connettere le informazioni e questo diventa un problema fondamentale in termini di prestazioni e analisi da effettuare. Si ha che le relazioni sono implicitamente incluse nei dati. La scelta del modello dipende dall'applicativo e ci sono applicativi che possono obbligare ad avere due o più modelli.
Ovviamente ogni vendor ha il suo linguaggio di query per il suo DBMS NoSQL.

5.2 Document based system

È uno dei modelli più ricchi nonché più vicini al modello relazionale. Nella sezione approfondiremo anche il più "famoso" DBMS documentale, ovvero **MongoDB**.
Si ha un'evoluzione del modello chiave-valore in quanto si ha una chiave, ovvero un *object identifier*, che può essere indicizzata tramite un *meccanismo di hashing*. I documenti solitamente vengo invece memorizzati in file **json** o **XML** e possono essere cercati a qualsiasi livello.
A differenza del modello relazionale il modello documentale è alla fine un albero e quindi l'accesso ai dati deve comunque partire dal nodo radice, a scendere verso le foglie (anche se si può avere un accesso **a indici** per accedere alle informazioni necessarie). Si ha quindi il concetto di **elemento più importante degli altri**, concetto assente nel modello relazionale, definibile "piatto" da questo punto di vista (a meno delle *data warehouse* dove si ha effettivamente una tabella principale).
La rappresentazione documentale è più vicina al pensiero umano di divisione delle cose in "cartelle" con varie tipologie di informazioni.
Si hanno quindi le seguenti caratteristiche per il modello documentale:

- è multidimensionale, ad albero
- ogni campo può contenere zero valori, un valore, multipli valori o altri documenti

- si possono effettuare query ad ogni campo e in ogni livello
- lo schema è flessibile, infatti due documenti della stessa **collezione** possono avere uno schema diverso di rappresentazione dei dati
- posso aggiungere "in linea" nuove informazioni
- avendo "annegato" uno dentro l'altro le relazioni che sussistono tra gli elementi, sono richiesti molti meno indici per effettuare le query migliorando le performance in quanto le informazioni sono legate tra loro nel documento (e non dovendo, come nel caso relazionale, prendere n tabelle in n file diversi nel *filesystem* facendo poi il join)

Il modello documentale permette due tecniche principali:

- **referencing**, ovvero, come per il modello relazione, si relazionano due documenti tramite un certo attributo, ad esempio:

```
contact
{
  "id": 2,
  "name": "Carlo",
  "address_id": 1
}
```

```
address
{
  "_id": 1,
  "city": "Milano",
  ...
}
```

- **embedding**, ovvero, aggiungere quello che nel *referencing* è un secondo documento come specifica di un certo valore, aggiungendo quindi un livello di profondità (quindi, per esempio, posso avere un documento relativo ad un contatto telefonico contenente anche indirizzo di casa e tale informazione, con i vari attributi, sarà specificata non in un altro documento ma nello stesso), Aumentando lo spazio che però non è più un problema. Ad esempio si ha:

```
contact
{
  "id": 2,
  "name": "Carlo",
  "address": {
    "city": "Milano",
    ...
  }
}
```

A causa del *time to/in market* ormai sempre più ogni applicazione ha il suo db e questo obbliga la flessibilità dello schema, che garantisce che una modifica allo schema, con aggiunta o modifica degli attributi, non è un problema (mentre con il modello relazionale sarebbe stato un problema). Questa flessibilità però comporta che in fase di query non ho uno schema prefissato tramite il quale interrogare (e non posso nemmeno vedere i primi n documenti in quanto l'$n+1$ potrebbe essere comunque diverso). Questa problematica è relativa all'uso di database costruiti da terzi.
Si hanno diversi linguaggi di query, tra cui *jsoniq, jmespath, jaql, JLINQ, etc...*, non avendo più quindi un unico linguaggio come era SQL per i modelli relazionali, ma avendone uno per vendor.
Dal punto di vista delle terminologia si hanno le seguenti "relazioni" con il modello relazione:

SQL	NoSQL
database	database
tabella	collezione
riga	documento
colonna	chiave

5.2.1 MongoDB

MongoDB è un DBMS non relazionale documentale con i dati memorizzati in un modello chiamato **binary json (*Bson*)**, che consente maggior efficienza tramite una rappresentazione binaria. Tramite l'embedding dei dati consente di non eseguire i *join* e permette anche l'accesso tramite indici. Supporta comunque il referencing anche se con cali di performances.
Dal punto di vista architetturale si hanno una serie di *engine* diversi fra loro (attualmente si usa **WiredTiger (*WT*)**, dopo acquisto di una società che si occupava di db relazionali distribuiti con transazioni, mentre prima si usava *MMAP V1*). Grazie a WT si è introdotto, nella v4, il supporto alle transazioni. Si ha poi il *data model*, il *MongoDB query language (MQL)* (proprietario), i componenti di controllo degli accessi, il meccanismo di gestione etc.... Le repliche sono organizzate in **replicaSet**.
Nel caso di MongoDB si ha:

RDBMS	MongoDB
database	database
tabella/view	collezione
riga	documento (Bson)
colonna	campo
indice	indice
join	documento con embedding
chiave esterna	referencing
partizione	shard

Vediamo quindi un esempio di schema in Mongodb:

Listing 2 Esempio di aggiunta di un documento *p* nella collezione *posts*

```
var p = {
  '_id': '342',
  'author': DBRef('User',2),
  'timestamp': Date('26-10-20'),
  'tags': ['MongoDB','NoSQL'],
  'comments':['author': DBRef('User',4),
              'date': Date('26-10-20'),
              'text': 'hello'.
              'upvotes': 7,...]
}
>db.posts.save(p);
```

Analizzando il codice abbiamo:

- la definizione di *p* come un oggetto json
- tra gli elementi abbiamo *tags* che contiene un array di valori
- tra gli elementi abbiamo *comments* che è un array di documenti
- l'ultima riga è tra le chiavi di successo di Mongodb in quanto ha un linguaggio simile a quelli di programmazione. Nel dettaglio si ha il caricamento del documento *p* nella collezione *posts*. Si ha un meccanismo simile alla OOP e l'integrazione coi linguaggi di programmazione è facilitata dall'ormai molto diffuso supporto ai json (come esistono già meccanismi per il passaggio da oggetti a json e viceversa)

Si possono inoltre fare indici, tramite `ensureindex` (che prende come argomento un json), a ogni campo del documento, per esempio:

```
>db.posts.ensureIndex({'author': 1});
>db.posts.ensureIndex({'comments.author': 1});
>db.posts.ensureIndex({'tags': 1});
>db.posts.ensureIndex({'author.location': '2d'});
```

Si nota che posso creare indici a livello che voglio, a documenti o array, o anche, come nell'ultimo esempio, usare indici geo-spaziali.
MongoDB, rispetto al modello relazionale, si avvicina tanto alla programmazione.

Repliche in MongoDB

Innanzitutto si ha che MongoDB rientra nella categoria CP rispetto al *CAP theorem*, non garantendo disponibilità in caso di guasto.
La replica viene effettuata in modalità master-slave anche se, nel gergo di MongoDB si ha **primary-secondary**, avendo scrittura sul primary e replica sui secondary. Si ha in realtà che uno dei nodi replica viene definito a posteriori *primary*. La replica avviene tramite il file di log del *primary*. Si ha un sistema che procede per repliche parziali incrementali, all'inizio il nodo fa un sync, prendendo i dati dal primary e poi prende dal file di log del primary solo gli ultimi n movimenti.
La garanzia della tolleranza si ha tramite il meccanismo di elezione dei nodi, nel caso il nodo primary vada down.
Si ha che un nodo può assumere uno dei seguenti stati:

- STARTUP: un nodo che non è membro effettivo di nessun replica set
- PRIMARY: l'unico nodo nel replica set che accetterà le operazioni di lettura
- SECONDARY: un nodo che di effettuare la sola replicazione dei dati (può essere promosso a PRIMARY in fase di elezione)
- RECOVERING: un nodo che sta effettuando una qualche operazione di recupero dei dati, magari dopo un *rollback* (può essere promosso a PRIMARY in fase di elezione)
- STARTUP2: un nodo che è appena entrato nel set (può essere eletto a PRIMARY)

- ARBITER: un nodo non atto alla replicazione dei dati con lo scopo di prendere parte alle elezioni per promuovere i nodi a PRIMARY (può essere eletto a PRIMARY in fase di elezione)
- DOWN/OFFLINE: un nodo irraggiungibile
- ROLLBACK: un nodo che sta svolgendo un rollbackper cui sarà inutilizzabile per le letture (può essere promosso a PRIMARY in fase di elezione)

L'elezione elegge il nodo con la versione più aggiornata dei dati e in questa fase le scritture sono interrotte (motivo per cui non è garantita la A di CAP). L'elezione dura comunque pochissimi.
Durante il processo di elezione ogni *replicaSet* manda, con timeout per assumere che un nodo sia down, un "hearbeat" (ovvero un bit ogni due secondi) agli altri nodi. Se il nodo primario risulta down si procede con l'elezione, che in termini umani è di tipo *maggioritario con liste bloccate* nel replicaSet. Ogni nodo ha un identificativo con un punteggio rappresentante la priorità nelle elezioni (più alto più probabile che venga eletto). Quindi appena il primary va down il secondario con valore più alto chiamerà le elezioni. Si ha che un replica set può avere al massimo 50 membri di cui solo 7 votanti, che esprimono un solo voto. I nodi SECONDARY non votanti hanno priorità 0 e accettano solo letture ma sono comunque in grado di votare. Nel caso di partizionamento to della rete si rischia un problema di riconciliazione tra vecchio primary e nuovo, dovendo quindi bloccare la scrittura.

Scrittura

La scrittura avviene sul nodo primario e si hanno diverse politiche in base tramite l'opzione *writeConcern* che dice cosa fare per le replicaSet. Si hanno tre parametri:

1. w, indicante il numero di nodi in cui il dato deve essere replicato prima di essere considerata conclusa l'operazione. Ovvero è il numero di nodi sui quali deve essere confermata la scrittura per essere confermata a livello globale
2. j, ovvero l'intenzione di scrivere sul log di *Wiredtiger*, tramite la logica *write ahead logging* (ovvero prima scrivo sul file di log e poi sul disco), prima che l'operazione sia stata eseguita. Questo per garantire la persistenza del dato, avendolo comunque nel file di log. Senza l'opzione lo scriverà in un secondo momento

3. *wtimeout*, ovvero il tempo limite, espresso in ms, da aspettare nel caso in cui $w > 0$ (se 0 non devo aspettare risposta da nessuno)

Aumentare w comporta aumentare il tempo di esecuzione. Nel dettaglio:

- $w = 0$ non da alcuna certezza di inserimento, utile per operazioni veloci
- $w = 1$ implica la scrittura sul nodo primary
- $w = n$ implica la scrittura su n nodi
- $w = majority$ dove almeno la metà dei nodi più 1 deve aver scritto prima della conferma dell'operazione (nella realtà si tenta di scrivere su più nodi ma appena si ha un ack dalla metà più 1 si conferma la scrittura anche se magari si sta ancora scrivendo su altri nodi)

Qualora si debba caricare una gran quantità di dati, di cardinalità conosciuta, per motivi di efficienza non conviene aspettare che il nodo dia l'ack ma si caricano tutti subito sul primary, controllando poi con `count` se tutto è stato caricato.

Frammentazione

Parliamo ora di frammentazione e scalabilità.
Si hanno 3 tipi di sharding per la scalabilità orizzontale:

- *hash-based*, quindi sulla chiave
- *range-based*, quindi in base ad un certo range di elementi
- *tag-aware*, quindi in base a certi attributi

Si ha un meccanismo dinamico basato su *pay as you go*, ovvero nel momento in cui ho lo sharding sarà MongoDB ad adattarlo nel momento in cui cambia il volume di dati, facendo un *bilanciamento automatico*. Si ha uno "spazio" della mia chiave di shard che posso dividere in n chunks per $n + 1$ chiavi di shard. Il numero massimo di chunk che è possibile definire su una sharded key può definire il numero massimo di shard in un sistema.
Si hanno 3 ruoli fondamentali nella frammentazione:

1. **mongos**, che è il punto di accesso software per le query, è l'entry point del cluster, tutte le query verranno eseguite su questo processo. Appena lanciato mongos va a leggere il config server

2. **config server**, ovvero un file che conosce la struttura degli shard, conoscendo frammentazioni e repliche. Dalla versione 4.0 ogni nodo distribuito è anche replicato

3. i vari **shard**, ovvero istanze di MongoDB

Quindi il router quando riceve una query capisce dove sono i frammenti, grazie al config server, e quindi procede con la query di tipo:

- **target query** se deve interrogare solo un nodo
- **broadcast query** se deve interrogare più nodi

Sia gli shard che il config server sono replicati su replicaSet per evitare che diventino *single point of failure* (anche per questo dalla 4.0 ogni volta che frammento replico).
Si ha un **primary shard**, che contiene l'intero db (ovvero tutte le collezioni) e i **secondary shard** che contengono frammenti e repliche di alcune collezioni.
Solitamente si ha quindi una configurazione a 3 nodi:

1. uno primary
2. due nodi secondari con frammenti diversi e le repliche del primary

Letture

Ho una politica di *readConcern* (che deve essere consistente come del caso del *writeConcern*, facendo scegliere se è più dispendioso scrivere o leggere):

- *local*, che è il valore di default t per leggere i dati dai nodi in un replicaSet. La query è fatta secondo la località spaziale e quindi se legge un nodo non primary rischio di leggere dati non ancora replicati
- *available*, di default per lettura su nodi secondari, che fa vedere che il dato c'è ed è consistente
- *majority*, quando almeno metà più 1 repliche ha ricevuto un ack in scrittura sul dato allora posso leggere

Vediamo quindi un esempio di interrogazione, tramite il metodo `find`:

db.user.find(age:$gt:18,name:1,address:1).limits(5)

che equivale a:

```
SELECT name, address
FROM user
WHERE age>18
LIMIT 5
```

$gt significa *greater than* e si hanno vari identificativi per i confronti (per non avere i simboli >, < etc... che darebbero fastidio nelle stringhe).
Vediamo anche una *insert*:

```
db.user.insertOne({
  name: "Andrea" ,
  age: 26,
  Teach[«datawarehouse», «architecture»]
})
```

con `insertOne` che specifica che voglio inserire un solo documento mentre avrei `insertAll` per tutti i documenti.
Un *update* avviene con la stessa logica con `updateAll`:

```
Db.user.updateAll({
  name:{$eq: «Andrea»},
  {$set: {Role: «PA»}},
  {$upsert:true}
})
```

$upsert sarebbe un *insert if not exist*, per permettere l'inserimento e non la modifica in caso di assenza. Tale cosa è assente in SQL ma servirebbe un doppio comando.
Si hanno varie utility per importare i dati, da specificare come opzioni al comando `mongoimport`, ad esempio:

mongoimport –d database –c collection –type csv -ignore blanks –file PATH

per importare un csv al path PATH ignorando gli spazi bianchi.

Transazioni

Dalla versione 4.0 MongoDB ha introdotto nel modello documentale le transazione tramite l'engine *Wiredtiger*. Introduce quindi il concetto di *log* che indica che tutto è nel nodo primary. Si opera sempre sul nodo facendo un *lock* che può essere di tipo:

- **sharded (*S*)**, per le letture

- **exclusive (*X*)**, per le scritture
- **intent sharded (*IS*)**, per esprimere l'intenzione di bloccare in modo condiviso uno dei nodi che discende dal nodo corrente
- **intent exclusive (*IX*)**, per esprimere l'intenzione di bloccare in modo esclusivo uno dei nodi che discende da quello corrente

Il lock garantisce transazioni a livello di singola collezione mentre prima l'atomicità era garantita a livello della singola operazione. Usando l'embedding, anche prima della v4.0, potevo gestire un sacco di modiche della singola collezioni. Nel caso del referencing invece serviva una gestione più approfondita delle transazioni. Essendo WT creato originariamente per un sistema distribuito funziona solo per sistemi distribuiti, infatti avendo un solo nodo si presuppone che si abbia una sola applicazione che accede, cosa che renderebbe a priori più semplice la gestione.

Modello contro efficienza

esempio a partire dalla slide 37
Spesso il cambiamento del modello può comportare un miglioramento delle prestazioni non indifferente, più del cambiamento dell'hardware (**verrà approfondito in live**).
Avere più piccoli documenti separati migliora il tempo, rendendolo lineare (mentre un grosso documento embedding aveva tempi esponenziali nell'inserimento delle righe).
Con questa soluzione poi devo sistemare le query.
Quindi l'importanza del modello è decisiva dal punto di vista delle prestazioni.
Nelle slide brevissima parte su CouchDB inutile.

Vengono qui aggiunte le cose dette in live.
Nella storia di NoSQL compaiono sempre grandi aziende, come Google, Amazon o Facebook che creano il software, fanno il paper e rendono il software open. Il rilascio dei codice in modo open source viene fatto in quanto in primis la community permette continui miglioramenti "a costo zero", inoltre le aziende in questione guadagno in primis su pubblicità (Google e Facebook in primis) e cloud (vedisi Amazon che fa il 90% degli utili dal cloud). Nessuno di questi vende direttamente tecnologia (anche perché, esempio, Android ha un valore economico basso) ma la usano. Si ha quindi esigenza applicativa,

non affidandosi a società esterne ma facendosi le cose internamente, rilasciando poi le cose open source per far abituare gli utenti alle proprie tecnologie, facendo così in modo che la gente voglia usare quelle tecnologie in ambito cloud. In tutto ciò si contribuisce a creare pubblicità mirate etc... in quanto il vero "bene" sono i dati non l'hardware o le tecnologie (e quelli non sono open). Si ha la **data-driven economy**, i dati sono il nuovo petrolio. IBM, Oracle etc... che vendevano invece la tecnologia ora sono più in crisi e hanno dovuto cambiare *business*.
Per progettare una review di prodotti servono informazioni in merito a chi fa la review (utente verificato, numero di vecchie review etc...). In ogni elemento del documento salvo queste informazioni. Non è poi possibile avere tutte le review nello stesso documento, quindi le primissime review (che vengono visualizzate subito) le lascio nel documento principale e per le altre lascio una reference ad un altro documento. Le review possono anche essere usate per valutare l'affidabilità dell'utente ed eventuali scelte alternative dei prodotti. Si punta comunque all'efficienze e all'efficacia.
I db documentali scalano tramite **distributed hashed table (*DHT*)**. Definendo una funzione di hash definisco implicitamente il dominio della funzione, posso quindi distribuire in modo efficiente sui nodi i vari valori di hash. Posso quindi avere diversi server con un nodo che salva le coppie chiave-valore con tutte le chiavi di sua responsabilità, è il **chord ring**. Ogni nodo quindi sa che dati deve gestire tramite la funzione di hash, avendo un algoritmo di scelta del nodo tramite *finger table*, con complessità logaritmica. L'aggiunta di un nodo può portare a buchi di sicurezza ma permette una scalabilità orizzontale incredibile.
sistemare questa parte!

5.3 GraphDB

È il modello più ricco e complesso tra i NoSQL.
Un grafo, da un certo punto di vista, ha proprietà simili a quelle di un modello ER. È infatti un insieme di nodi e di relazioni tra loro, dove i concetti più importanti (quelle che in ER sarebbero le *entità*) sono modellate come nodi. Bisogna quindi studiare come connettere i nodi, infatti il modo con cui le entità sono semanticamente associate si modellano come relazioni, tramite archi.
Si hanno campi di applicazione dei db a grafo:

- social network

- sistemi di raccomandazione, che suggeriscono oggetti simili a utenti simili
- ambito geografico
- reti logistiche (per spedizioni etc...)
- transazioni finanziarie per la *fraud detenction* (riconoscendo certi pattern sospetti, anche a livello geografico)
- master data management, rappresentazione unica della realtà
- bioinformatica
- controllo di autorizzazioni e accessi

Esempio 5. *Per esempio su Twitter avrei i nodi con gli utenti e gli archi rappresentanti la relazione follow.*

Si usa una **descrizione estensionale dei dati**. Si può avere infatti:

- **descrizione estensionale dei dati** quando si descrivono le istanze via-via, descrivendo anche istanze di tipo diverso (magari utenti e messaggi) usando la stessa sintassi del nodo (ugualmente per gli archi). La semantica viene data dall'etichetta
- **descrizione intensionale dei dati**, che bisognerebbe inferirla direttamente dai dati e quindi non è applicabile

Nei RDBMS si aveva il *join* basato su valori, facendo uno *scheme understanding* sulle varie tabelle, ottenendo quindi il risultato tramite anche più operazioni di *join*. Si può avere però il problema applicativo, avendo magari **relazioni ad anello** tra le varie tabelle ad esempio una tabella "persona" che tramite una tabella "amici" ritorna su se stessa, avendo la tabella "amici" che esplode in numerosità (magari cercando amici di amici etc... complicando le cose di livello in livello). Avendo potenzialmente livelli infiniti di *join* si va incontro al **join bombing**, che fanno soffrire i DBMS. Si può considerare comunque il **six separation degree**, uno studio sociologico, che dice che con sette connessioni si può raggiungere chiunque (ora coi social ne bastano meno). Simile è il **Kevin Bacon number**, che dice il numero medio di film fatti tra artisti per ottenerne uno fatto con Kevin Bacon, anche in questo caso si arriva allo stesso risultato visto con il **six separation degree**.
Nei sistemi a grafo abbiamo quindi gli archi mentre nei documentali avevamo identificatori che puntavano ad altri documenti (con un modello basato sui

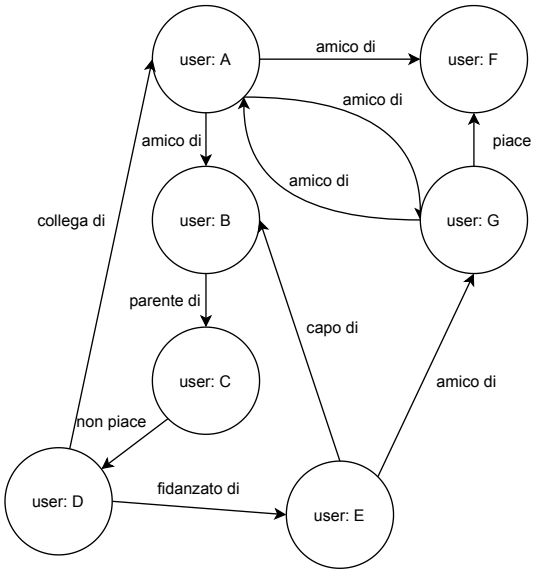


Figura 5.2: Esempio di db a grafo

valori come se fossero *foreign keys* dei RDBMS), comportando, nel caso dell'embedding, task di modellazione complessi, scarsa efficienza e complessità nel gestire dati fortemente connessi, mentre, con il referencing starei simulando una sorta di modello a grafo.
I db a grafo hanno forte potenza espressiva permettendo di caratterizzare i nodi con un certo *tipo* e definire le varie relazioni tramite archi. Coi grafi si possono facilmente rappresentare relazioni umane, sociali, affettive, lavorative etc... usando dei semplici archi. Si possono connettere elementi tramite un'infinità di tipi di relazioni.
Nei nodi posso rappresentare anche entità complesse, con vari attributi, ovvero varie **proprietà** (ovviamente ogni nodo può avere un set di attributi diversi, avendo fortissima libertà di schema).
È un modello molto più immediato rispetto ad un RDBMS. Inoltre, nel caso si abbiano potenzialmente più relazioni tra due nodi basta contare quanti cammini orientati (che possono quindi passare per nodi intermedi diversi) esistono tra quei nodi per ottenere la cardinalità di tali relazioni.
Un grafo con le proprietà nei nodi è detto **property graph** È un modello simile a quello che si fa su una lavagna, è una modellazione simile a quella umana, basti pensare alle mappe concettuali.
Si hanno comunque vari tipi di *property graph*;

- quando sia nodi che relazioni possono avere proprietà espresse da chiave-valore, accettando quindi che i nodi abbiano un tipo

Figura 5.3: Esempio di db con rappresentato da un **property graph**, con *user* come chiave e un array di valori

- altri ammettono array di valori

Si nota che invece in altri modelli NoSQL come RDF i nodi possono avere un tipo ma non proprietà e le relazioni non hanno proprietà, dovendo sempre rappresentare il tutto tramite triple "soggetto-predicato-oggetto".
Il problema di modellazione dei grafi si ritrova quindi nella scelta di cosa mettere come nodo e cosa come relazioni (ed eventualmente le proprietà), avendo varie scelte modellistiche con vari e pro e contro.
Ovviamente anche qui ogni vendor ha il suo linguaggio, anche se ne esiste uno *standard*, chiamato **gremlin**, che più che essere un linguaggio di interrogazione è un *linguaggio di attraversamento dei grafi* (dato che cercare sottografi e attraversare il grafo è la principale tecnica di studio degli stessi che si ha a disposizione). Questo tipo di linguaggio viene detto **vertex-based** che si usano per studiare la validità di predicati tramite l'attraversamento dei nodi.
Si hanno vari vendor quindi e si hanno due tipi di approccio:

1. **graph processing**, dove ci si concentra sul processamento dei dati, processamento che viene eseguito tramite i grafi
2. **graph storage**, dove i dati vengono memorizzati in forma di grafo

In entrambi i casi la cosa può essere fatta in modo:

- **nativo**:
  - *native graph storage* per indicare che sono ottimizzati e progettati per le gestione stessa dei grafi. In questo caso comunque i dati vengono memorizzati in modo intelligente, memorizzando in modo vicino i nodi e il loro archi, facendo *index free adjacency*, in modo che seguire gli archi uscenti da un nodo è molto più immediato. Tra gli esempi di db abbiamo **Neo4j**, che in memoria salva, per ogni nodo, tutti i suoi riferimenti, tramite pointers. In scrittura comunque si perdono performance, a vantaggio di una lettura molto rapida

(al costo comunque di molto uso di RAM). Si evita il *join bombing* seguendo i riferimenti vicini ad ogni nodo, facendo l'attraversamento

- **non nativo**:
  - *non native graph storage* per indicare che i dati vengono trasformati da grafo ad un formato diverso tra cui relazionale, object oriented db etc... Si rischia comunque il *join bombing*

Potrei avere un db con una rappresentazione a tabelle (e non a grafo) studiata come se fosse un grafo (traducendo poi le query in SQL), questo è l'approccio di **FlockDB**.
Un'alternativa è avere, come nel caso di **Titan**, dove posso comunque usare *Gremlin* ma con dati memorizzati in modo colonnare.
Si hanno quindi due approcci principali per gestire un grafo:

- **graph database**, ovvero un DBMS che gestisce in maniera persistente un grafo che viene usato per le transazioni. Si ha in questo caso la dicitura OLTP
- **graph compute engine**, ovvero tecnologie per l'analisi *off line* dei grafi. Si ha in questo caso la dicitura OLAP

I graph database supportano le operazioni CRUD (come tutti i modelli NoSQL):

- Create
- Read
- Update
- Delete

Come abbiamo visto si studiano gli attraversamenti del grafo, che sono equiparati ai *join* del modello relazionale, per fare le query in modo performante non facendo join ma attraversando e basta. Sono inoltre costruiti per essere usati nei sistemi transazionali (OLTP).
Si ha un problema grave coi grafi, motivo per cui si hanno i *non native graph storage*: **i grafi non scalano bene e non possono essere facilmente frammentati**.
Partizionare un grafo raramente è possibile, in quanto comporta un taglio

di relazioni. Se quindi si vuole scalare bisogna passare, alla fine, ad un altro modello, aggiungendo però costo di comunicazione tra il modello a grafo e il nuovo modello scelto, aggiungendo un layer software.

5.3.1 Neo4j

Abbiamo già introdotto precedentemente Neo4j. Sappiamo che è nativo sia per il *graph processing* che per il *graph storage* (sovvrendo quindi la scalabilità).
Approfondiamo quindi il **linguaggio di interrogazione di Neo4j**, chiamato **Cypher** (*OpenCypher* nella versione open).
Cypher è definito come un **pattern-matching query language**. È un linguaggio simil-umano, espressivo, dichiarativo (si dice il cosa e non il come), che consente operazioni di aggregazione, sorting e di limit (trovare ad esempio le n top key etc...) e permette di aggiornare il grafo.
Un formato in modo testuale viene rappresentato con le parentesi tonde per i nodi e le quadre per gli archi. Le relazioni orientate vengono indicate tramite "->" o "<-".

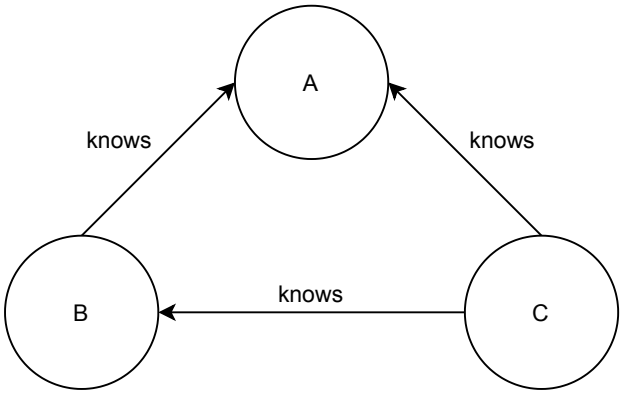
Esempio 6. *Tramite:*

(c)-[:KNOWS]->(b)-[:KNOWS]->(a), (c)-[:KNOWS]->(a)

con la "," che significa $\wedge$*.*
Equivalentemente potevamo avere:

/c)-[:KNOWS]->(b)-[:KNOWS]->(a)<-(c)-[:KNOWS]

per rappresentare:



Vediamo quindi un'interrogazione classica in Cypher, dove si cercano tutti gli utenti tali per cui conosce qualcuno a che conosce qualcuno b e che sono etichettati come "Michael", ritornando a e b:

```
MATCH(c:user)
WHERE (c)-[:KNOWS]->(b)-[:KNOWS]->(a),
      (c)-[:KNOWS]->(a), c.user="Michael"
RETURN a, b
```

Il risultato sarà una ***tabella*** *che contiene tutte le coppie a e b. Avrà dei dati ridondati a causa della dualità intrinseca di avere a che conosce b, cosa che probabilmente porta ad avere che b conosce a.*

Quindi il risultato **non sempre** è un grafo (mentre nel documentale era sempre un documento e nel relazionale una tabella) ma può anche essere altro, ad esempio una tabella. Non è quindi sempre possibile concatenare query in quanto potrei non avere un grafo in uscita.
Si hanno comunque clausole (coprendo le operazioni CRUD):

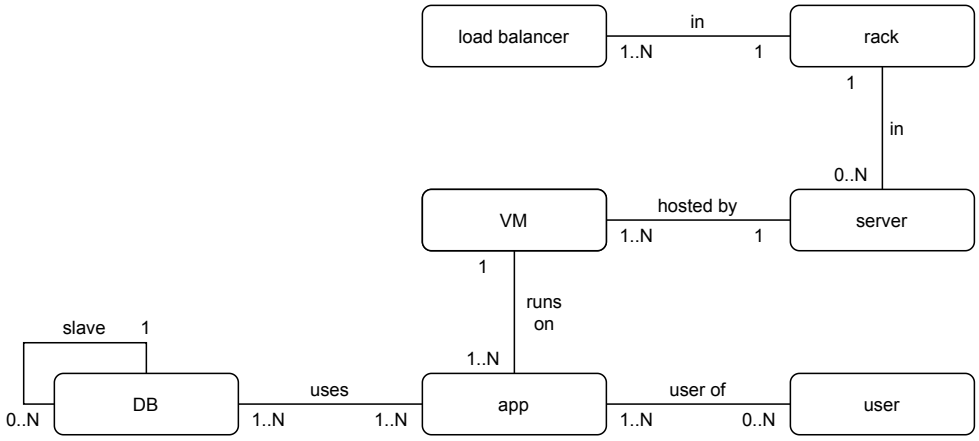
- *where* per imporre criteri di filtraggio tramite pattern-matching dei risultati
- *create/create unique*, per creare nodi e relazioni
- *delete*, per cancellare nodi, relazioni e proprietà
- *set*, per impostare dei valori alle proprietà
- *foreach*, per fare un update per elementi del grafo posti in una lista (per ogni nodo, per ogni arco etc...)
- *union*, per unire i risultati più query
- *with*, per concatenare query propagando i risultati in pipe

5.3.2 Grafi e modello relazionale

Compariamo ora il modello a grafi con il modello relazionale.
Per farlo sfruttiamo un esempio:

Esempio 7. *Vogliamo costruire un modello relazionale per la gestione di una server farm. Si ha che l'utente accede ad una applicazione che gira su una VM ed accede ad un db (primario o secondario). I server che ospitano le VM sono installati su sistemi rack controllati da un sistema di load balancing.*

Una tipica attività è quella di controllare se ogni elemento del sistema funzioni e, in caso di guasto, capire cosa non stia funzionando. Per farlo si hanno dei sensori, sia software che hardware, che monitorano le varie componenti. Modello uno schema del mio sistema collegando in una mappa concettuale le varie componenti che comunicano tra loro o che sono in relazioni (anche solo per dire che un server è in un certo rack). Si modellano così le varie istanze. Tale modellazione relazionale può avere il seguente ER:



Ogni elemento ha l'attributo on per indicare se i sensori dicono che la componente funzioni o meno, attributo che verrà usato per le query.
Qualora quindi un'utente mi dica che una certa applicazione non funziona devo risalire step by step al server, eventualmente al rack, al db etc..., avendo una quantità di join non indifferente (6 join diversi nel caso peggiore).
Una rappresentazione a grafo sarebbe molto più semplice, avendo dei semplici cammini anziché avere tutti quei join. Si avrebbe quindi un nodo per singola componente reale con le relazioni tra le varie singole componenti.
Pensiamo quindi, per esempio, alla seguente query:

Trovare gli asset (server, app, machine virtuali) in stato down per l'utente user 3

```
START user = node:users(name = "user 3")
MATCH (user)-[*1..5]-(asset)
WHERE asset.status! = "down"
RETURN DISTINCT asset
```

Parto quindi dal nodo che come nome "user 3", trova tutti quegli asset con distanza da 1 a 5 rispetto al nodo utente, arrivando quindi fino a 5 livelli di profondità. Se tale asset è in down viene restituito una sola volta.

5.3.3 Ancora su Neo4j

Tutta la teoria e la letteratura sui grafi viene applicata ai db a grafo, dai cammini minimi alle clique. Il vero problema resta la modellazione e la scelta del DBMS. Approfondiamo quindi ancora di più Neo4j.
Abbiamo già visto che non ha meccanismi nativi di frammentazione, avendo scelto di sfruttare la *index free adjacency* e la modellazione a grafo, non permettendo scalabilità. In compenso garantisce varie caratteristiche funzionali:

- permette le transazioni con tutte le proprietà ACID, con meccanismi di locking
- garantisce la disponibilità
- garantisce la recoverability

In merito alla scalabilità, nonostante le limitazioni sopra esposte, garantisce l'**alta disponibilità**. L'idea è che è possibile andare a definire un cluster di nodi Neo4j, alcuni chiamati **core server**, ovvero nodi per cui tutte le scritture sono fatte in contemporanea, essendo quindi sincronizzati. Si hanno poi gli altri nodi che sono chiamati **replica server**, che sono di sola lettura, che vengono creati tramite meccanismi di replica simili a quelli già trattati. I *replica server* vengono anche usati per reporting analysis etc.... Fatta una query essa comunque non viene distribuita ma eseguita sul nodo più vicino. Tutto questo sistema è detto **casual clustering**.
Si ha una bassa **latenza** nelle query in quanto si usa un indice per risalire al primo nodo e poi si procede coi cammini, avendo quindi che le performances non dipendono dalla grandezza del dataset ma solo da quali sono i dati richiesti dalla query.

5.4 Modelli poliglotta

In situazioni reali, ad esempio un e-commerce, posso avere i dati salvati in vari DBMS, sia relazionali che NoSQL, a seconda dei dati che contengono (magari avrei le raccomandazioni in un db a grafo con Neo4j, il catalogo prodotti in un documentale come MongoDB, le sessioni utente in un db key-value come Redis, dati finanziari e reporting in un RDBSM, per i log delle attività un modello NoSQL colonnare etc...). Dover gestire tutti questi modelli diversi è molto completo, ma sensato in base al tipo di dati. Ho quindi una sorta di **mondo poliglotta**.

5.4.1 ArangoDB

In questo mondo poliglotta complesso ci arriva in soccorso un particolare DBMS: **ArangoDB**.
Il DBA dovrebbe conoscere tutti i modelli in uso e tutti i linguaggi associati, essendo anche in grado di gestire i vari conflitti. ArangoDB introduce il concetto di **db poliglotta**, supportando i seguenti modelli:

- key-value
- documentali
- a grafo
- a ricerca full-text (come fa anche **Elastic Search**)

Rispetto ai modelli precedentemente descritti a grafo con storage non nativo, tipo **Titan**, che appunto accettano solo un linguaggio di query a grafo, il linguaggio di query di ArangoDB, chiamato **AQL (*Arango Query language*)**, garantisce la possibilità di interrogare dati in formato json (quindi documentale), a grafo, in formato key-value o per il full text search.
Ad alto livello Arango organizza i dati in database e collezioni (per fare un parallelo schemi e tabelle), dove le collezioni memorizzano documenti o similari. I documenti, come già accennato, sono documenti json di profondità arbitraria (sub-object, sub-arrays etc...), non avendo alcun limite da questo punto di vista. Nel caso semplice tali documenti sono omogenei, avendo stessi attributi e tipi, e effettuare una query AQL è molto semplice:

```
FOR c IN categories
FILTER c._key IN [ 'books', 'kitchen' ]
RETURN c

FOR c IN categories
SORT .title
RETURN { _key: c._key, title: c.title}
```

Avendo un linguaggio simile a uno di programmazione.
Abbiamo quindi, nella prima query, che selezioniamo tutti i *c* in '*categories* (che è una collezione) trovando solo le chiavi che sono o "book" o "kitchen" ritornando *c*. Nella seconda selezioniamo selezioniamo tutti i *c* in '*categories*, ordiniamo i risultati in base al titolo e ritorniamo chiave e titolo.
Posso comunque avere documenti eterogenei, con attributi diversi. Possiamo comunque interrogare usando il modello a grafo e tali query mostrano quali documenti sono collegati (direttamente o indirettamente) ad altri documenti,

specificando il cammino per ottenere il collegamento. Queste connessioni tra documenti sono chiamate **edges** che sono salvati in "edge collections". Gli *edges* hanno sempre due attributi che si riferisocno ai nodi entranti e uscenti:

- *_from*
- *_to*

Avendo sempre archi direzionati del tipo:

$$\_from \rightarrow \_to$$

anche se si possono effettuare query anche nell'ordine opposto.

Esempio 8. *Preso il documento json:*

```
{ "_key" : "sofia", "_id" : "employees/sofia"}
{ "_key" : "adam", "_id" : "employees/adam"}
{ "_key" : "sarah", "_id" : "employees/sarah"}
{ "_key" : "jon", "_id" : "employees/jon"}
```

posso avere un ***edge****, per una certa relazione, del tipo:*

```
{ "_from" : "employees/sofia", "_to" : "employees/adam"}
{ "_from" : "employees/sofia", "_to" : "employees/sarah"}
{ "_from" : "employees/sarah", "_to" : "employees/jon"}
```

(Potrei comunque continuare a interrogare il primo json come se fosse un documentale)

Rispetto a quanto visto per i modelli a grafo cambia drasticamente la scalabilità.
Si hanno vari metodi di scalabilità, avendo le modalità di deploy:

- a singola istanza
- master/slave, scritture su master e repliche su slave
- active failover
- cluster
- multiple datacenters

Approfondiamo soprattutto gli ultimi tre.

Active failover

Nel modello master/slave quando il master fallisce lo slave non può essere sostituito e quindi si è pensato al modello **active failover**, dove esistono delle istanze *single-server*, chiamate **leader**, che sono in lettura e scrittura e delle istanze, sempre *single-server*, chiamate **followers**, che non sono altro i vecchi slave, passivi e non scrivibili. Il leader invia le operazioni di scrittura ai followers che replicano in modo asincrono i dati tramite la cosiddetta **Write-Ahead Log (*WAL*)**, ovvero il leader prima salva su un file di log e poi sul disco e tale file di log, tramite replica incrementale viene aggiornato, tramite un meccanismo di replica chiamato **apache mesos**. Esiste poi almeno un **agency** che controlla le configurazioni di leader e followers e controlla che siano tutti disponibili. Qualora l'agency si accorgesse che un server leader non è più disponibile promuove, con un meccanismo elettivo, un follower a leader. Resta quindi una sorta di master/slave ma con gli agency che procedono con il controllo e la rielezione.

Cluster

Il cluster è un'evoluzione, avendo più **coordinator** che si connettono con i client. I coordinator effettuando le query e gli **agency** non si occupano di altro se non l'elezione degli stessi e della sincronizzazione dei servizi per l'intero cluster. Si hanno poi i **DB server**, di tipo slave, che, in caso di fallimento, non comporta problemi in quanto l'agency attua le solite politiche di aumento repliche, se esse sono previste, bilanciando nuovamente i dati in caso di sharding.
Lo sharding avviene in partizioni da almeno 40gb, con un meccanismo simile a quello di MongoDB: ogni server avrà associato uno shard, e ogni server, avendo ciascuno circa 3 repliche, avrà le repliche di altri server (oltre alla sua stessa replica, di cui è **leading shard**) (??????). Questo sistema permette di interrogare quasi direttamente tutto sullo stesso server (avendo molte repliche) e permette, in caso di server non funzionante, di poter stabilire che un altro server diventa il **leading shard** per un certo frammento (tramite meccanismi soliti di elezione tramite gli agency), replicando per avere comune le tre repliche tra i vari server e potendo continuare a fare interrogazioni.
Si hanno due soluzioni per la scrittura:

1. il protocollo **read once write all (ROWA)** che obbliga il coordinator a scrivere su tutte le repliche
2. il coordinatore scrive solo dove si ha la copia primaria e poi si hanno le repliche, è la soluzione **oneshard** (che era chiamato **write concern** in MongoDB)

Arango quindi scala molto bene.

Cluster to cluster

L'ultimo livello è il **cluster to cluster**, una soluzione asincrona di tipo **one way** in cui prendo un cluster e lo copio interamente in un altro, usando una coda come **kafka**. La replica è one way per cui scarico dal primary alla replica o viceversa in tempi diversi, non contemporaneamente. Non è pensato per fare una replica del single-server (userei il master/slave).

5.5 Modello key-value

È un modello abbastanza semplice dove si hanno appunto rapporti chiave-valore. I valori sono considerati come dei blob, non essendo accessibili/interpretabili direttamente dal db(??????). Si hanno poche operazioni:

- inserire un valore data la chiave
- trovare un valore data la chiave
- modificare un valore data la chiave
- cancellare una chiave e il suo valore

È inoltre possibile associare, in alcune soluzioni come **Redis**, un tipo ad un valore, per esempio *string, integer, list etc...*.
Si ha accesso veloce ai dati tramite strutture di hash e posso avere scalabilità orizzontale. I modelli key-value funzionano solitamente *in memory*, anche nel caso della condivisione dati.
Dal punto di vista della nomenclatura si ha:

relazionale	key-value
tabella	bucket
riga	key-value
id-riga	key

Tra i vari DBMS abbiamo:

- Redis
- Memcached
- Riak KV

- Hazelcast
- Ehcache

5.5.1 Redis

nato del 2009 da Salvatore Sanfilippo mentre stava lavorando su una soluzione di realtime web analytic con mysql. Nasce a causa dei limiti del modello relazionale, non riuscendo a fare le interrogazioni in realtime. Assunto da VMware continua a lavorare su Redis fino a mettesi in proprio: *Redislab* (con la parte di ricerca e sviluppo in Sicilia e la sede principale negli USA).
Redis può gestire chiavi espresse in ASCII. I valori primitivi sono le *string* ma si hanno vari container di stringhe:

- hashes
- lists
- sets
- sorted sets

Non si può sempre fare una ricerca per valore all'interno di questi elementi.
Sono stati poi aggiunti i **moduli** per fari formati di file, per memorizzare file *json* ma anche per *video in streaming*. Offre quindi blob più strutturati per memorizzare i dati, alcuni a pagamento.
Si ha una versione distribuita enterprise, a pagamento. Si possono fare repliche e avere una soluzione in cluster con un certo numero di shard.
Studiamo quindi come avviene lo sharding.
Bisogna stabilire una *policy* per capire come dividere i dati e si è scelto di dividere le chiavi in base al valore di hash. Si può, volendo, anche dividere le chiavi in base a delle RegEx.
Inoltre bisogna capire dove distribuire i dati e si hanno due soluzioni:

1. quella di default, detta **dense**, dove il meccanismo di sharding è gestito tramite un meccanismo incrementale, ovvero quando si riempie la memoria della macchina si crea un'altro shard su un'altra e così via
2. una detta **sparse**, dove, avendo già definito il numero di nodi, vado a popolarli fin da subito

Si ha l'alta disponibilità in base a due soluzioni (???):

1. una soluzione, detta **Redis replica of setup** in cui si hanno dei nodi che formano il cluster su un datacenter su cui si effettuando le operazioni e altri nodi che formano altri cluster su altri datacenter verso cui sincronizzo. È quindi un master/slave, replicando per disaster-recovery

2. una soluzione, chiamata **active-active setup**, in cui posso scrivere e leggere sui cluster presenti in due datacenters diversi (ma magari nello stesso rack, parlando di **Redis cluster with rack zone awareness**) avendo in entrambi i datacenter sia i db principali che le repliche. Magari avendo protocolli ROWA etc...

Di recente si è aggiunta la possibilità di rendere persistenti dati anche se il loro punto di forza resta la gestione *in memory* (piuttosto che andare a costruire dei sistemi di *cache* etc...).

5.6 Wide column

Studiamo ora il modello **wide column store**.
Anche se storicamente non è così il modello *wide column* può essere visto come un'evoluzione del modello *key-value* (è nato prima il modello *wide column*). L'idea è che accanto ha la chiave non ho un valore/blob ma una riga con degli attributi mono-valore. Si ha quindi una via di mezzo tra il modello *key-value*, essendo possibile fare interrogazioni su queste righe e quindi all'interno delle colonne, e il modello documentale, non riuscendo a garantire la stessa potenza descrittiva (i documenti sono comunque più espressivi essendo più strutturati, permettendo sub-array e sub-document).
Tra le implementazioni troviamo:

- **BigTable**, il primo modello *wide column* introdotto da Google
- **Hbase**
- **Cassandra**, che però viene ritenuto spesso *key-value* e anche loro stessi si definiscono *key-value*, basando la loro distribuzione sulle **distributed hashed table (*DHT*)**

5.6.1 BigTable

L'idea dietro **BigTable** è quella di una mappa multidimensionale ordinata, persistente e sparsa, ovvero si ha una mappa indicizzata tramite tre informazioni:

1. row key, di tipo *string*
2. column key, di tipo *string*
3. timestamp che caratterizzano ogni singolo valore, compreso la column key (da immagine sembra così perlomeno). Sono di tipi *int64*

All'interno si hanno poi varie **column family** con i vari dati, memorizzati come *string*.
BigTable è molto usato in ambito web grazie alla sua capacità di memorizzare contenuti HTML etc...
Grazie ai timestamp si garantisce un controllo di concorrenza basato sul multiversioning. Possiamo dire che i sistemi che implementano *wide column* con timestamp supportano transazioni ACID compliant tramite **MultiVersion Concurrency Control (*MCC o MVCC*)**.
Un insieme di righe viene definito **tablet**. Abbiamo visto che ogni riga può avere diversi **column family** e si ha che ciascun column family può avere diversi **qualifier** e per ogni qualifier più valori diversi. La libertà dello schema deriva che ogni column family può avere un numero diverso di qualifier. Si ha quindi l'approccio *key-value* per l'accesso tramite chiave e si ha un sistema di indicizzazione per migliorare le prestazioni. Ogni column family ha un nome, una *string*, è può contenere altre colonne, che a loro volta appartengono ad una column family, specificata tramite `familyName:columnName`. Un esempio nella figura 5.4. Non è semplice facile le range query a causa del meccanismo di hashing.
Il timestamp (anche se può essere usato altro) garantisce un **version number** univoco.

5.6.2 Hbase

La versione distribuita di BigTable venne rinominata **Hbase** (dove *H* sta per **Hadoop**).
In questo caso i dati sono divisi in tables e ogni table è composta di colonne che sono a loro volta raggruppate in column family (che possono avere un numero variabile di colonne, che erano i qualifier in BigTable). Ovviamente supporta il multiversioning.
Un'istanza di Hbase è formata da varie righe, identificate tramite una chiave, a cui sono associate, per ciascuna riga, varie column family, una per ogni tipo di dato. La column family viene rappresentata tramite un insieme di coppie attributo-valore e i valori possono avere timestamp diversi, specificati tramite `@timestamp` (`@ts=value`).

row key	timestamp	column "column1"	column "column2"	
"key 1"	t_{12}	"<html>..."		
	t_{11}	"<html>..."		
	t_{10}		"column2:cnnsi.com"	"CNN"
"key 2"	t_{15}		"column2:apache.com"	"APACHE"
	t_{13}		"column2:my.look.ca"	"CNN.com"
	t_6	"<html>..."		
	t_5	"<html>..."		
	t_3	"<html>..."		

Figura 5.4: Esempio di modello *wide column* implementato da BigTable

Si possono avere le stesse column family, dal punto di vista del nome, in più righe ma queste possono avere un insieme diverso di valori, garantendo che la table è **sparse**, rivelandosi utile per il mapping "uno a tanti". I dati sono tutti *bytes*. Un esempio nella figura 5.5. Il modello colonnare non presta attenzione ai *join*, che in questo caso implicherebbero scorrere l'intera tabella, ma si concentra sul **denormalizzare** ovvero a replicare le informazioni nel momento in cui si crea una column family nuova (???).
Quindi lo schema è formato dalle tables e dalle column family, **le colonne non fanno parte dello schema**.
Hbase implementa quindi le **dynamic Columns** in quanto ogni riga ha

row key	data
A	info: {'altezza': `1.8m', 'stato':'italia} ruolo: {'fiat': 'direttore', 'alfa': 'co-fondatore'}
B	info: {'altezza': `1.75m', 'stato':'italia} ruolo: {'fiat': 'direttore'@ts=2010, 'fiat': 'amministratore delegato@ts=2011, 'lancia': 'consulente'}

Figura 5.5: Esempio di istanza di Htable, dove si nota quanto detto, avendo le stesse column family ("info" e "ruolo") ma con all'interno anche attributi diversi (esempio "consulente")

differenti colonne, i quali nomi sono codificati nella riga, non facendo parte dello schema.
Il versioning è essenziale per garantire lo storico delle informazioni.
SU Hbase ho alcune operazioni fondamentali:

1. aggiungere valori
2. ricercare per valore esatto (e non in modo range-based)

Vediamo quindi come si distribuisce un cluster Hbase. Si hanno tre componenti principali (ricordando che si basa su Hadoop), avendo un'architettura master/slave:

1. un master, detto **HbaseMaster**
2. tanti region server, detti **HRegionServer**, che sono i vari slave
3. il client, detto **Hbase client**, che dialoga con il singolo master

Nel dettaglio una *region* è un sottoinsieme di righe della tabella, che vengono partizionate orizzontalmente tramite regole standard di frammentazione (partizionamento che viene fatto in automatico). Quindi un *HRegionServer* gestisce un insieme di righe di una o più tabelle e la sincronizzazione (lettura e scrittura) avviene tramite file di log. Il master coordina gli slave, assegna le region, riconosce fallimenti degli slave etc. . ..
Il client quindi fa la richiesta al master il quale cerca il dato e sviluppa l'operazione all'interno della singola region. Nel momento in cui il region server scrive i dati, attraverso il sistema distribuito chiamato **Hadoop Distributed File System (*HDFS*)**, viene fatta la replica e avviene la sincronizzazione dei dati. Potrei avere più master coordinati tra loro. Il coordinamento tra i master e i region server avviene tramite il meccanismo di **ZooKeeper**, ovvero un orchestratore software. HDFS permette di avere la memorizzazione orizzontale dei dati e tramite il **write ahead log (*WAL*)** prima scrivo sul log e poi sul disco, garantendo la ricostruibilità dei dati in caso di guasto e garantendo che prima o poi i dati saranno allineati. ZooKeeper quindi coordina master e region server per lettura/scrittura mentre HDFS consente l'allineamento e la memorizzazione su un file system distribuito.
Hbase ha un linguaggio di query poco ricco, potendo fare solo una *select* di dati con un certo valore. Si può comunque fare la ricerca sia sulle righe che su alcuni attributi (definiti precedente come "chiave"). Il sistema di ricerca è estremamente performante.

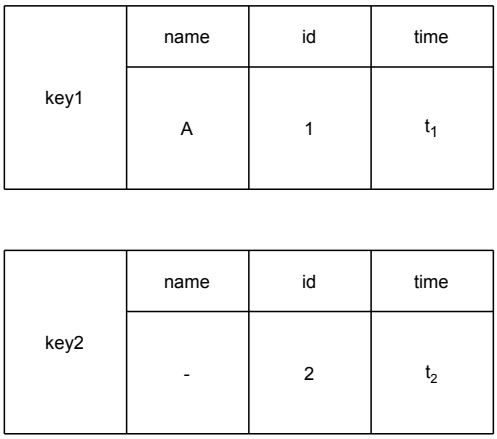


Figura 5.6: Esempio di due righe della stessa column family di Cassandra con due chiavi e tre colonne (con i tre column name "nome", "id", "time")

5.6.3 Cassandra

Come abbiamo detto **Cassandra** lo studiamo come modello *wide column* anche se loro stessi si definiscono *key-value*, essendo sostanzialmente tale ma usando concetti dei modelli *wide column*.
Cassandra è l'evoluzione open source del db NoSQL utilizzato da Facebook e risulta interessante perché implementa un sistema di distribuzione completamente diverso da quello master/slave di Hbase.
In cassandra esiste il concetto **column family**, definito all'interno di un **key space**, come insieme di coppie key-value e quindi una column family altro non è che una tabella con le coppie key-value come righe. Il concetto di column family quindi varia rispetto a quello di Hbase. Una riga è una collezione di colonne con un nome (sono quindi come le righe di una tabella relazionale). La chiave corrisponde al nome della colonna e ogni riga (che è appunto usa serie di coppie key-value definite come colonne) contiene almeno una colonna. Un esempio di modellazione è visibile alla figura 5.6. Si possono aggiungere attributi a piacere e modificarli (mentre in Hbase si creavano/modificavano le column family). Se i valori corrispondenti a tutte le column name sono uguali a "-" si specifica che essi non sono usati. Posso anche specificare di avere un singolo valore non in uso sempre con "-".

Il linguaggio di query di cassandra è chiamato **Cassandra Query language (CQL)** molto simile ad SQL. Vediamo un esempio di creazione di tabella (dove per ogni attributo si specifica il tipo), con la specifica di una primary key:

```
CREATE TABLE users(
  email varchar,
  bio varchar,
  birthday timestamp,
  active boolean,
  PRIMARY KEY (email)
)
```

(si specifica che i campi di tipo timestamp sono specificati in millisecondi).
Per intenerimento dati si procede in modo simile, anche qui, ad SQL:

```
INSERT INTO users(email, bio, birthday, active)
VALUES('example@example.com',
       'esempio',
       516513600000,
       true)
```

Inutile dire che anche per le query si ha un modello simile ad SQL dove le *select* leggono uno o più records dalle column family di Cassandra ritornando un insieme di righe:

```
SELECT * FROM users;

SELECT email FROM users WHERE active = true;
```

La particolarità di Cassandra è l'architettura.
Se BigTable usava:

- Column families
- Memtables
- SSTables

e altri DBMS come **Amazon Dynamo** dove si ha (???):

- hashing consistente per mantenere la replica
- partizionamento e replica
- routing *one-hop*

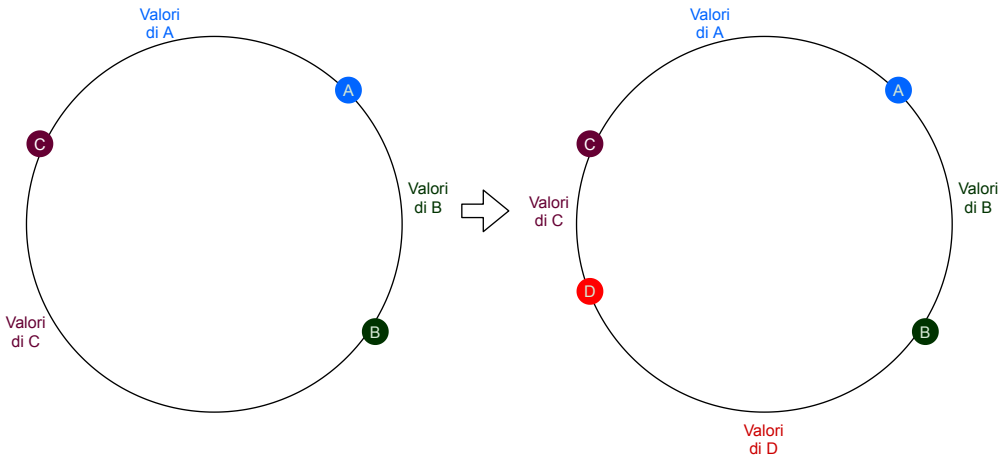


Figura 5.7: Esempio di inserimento di un nodo ("D") nell'architettura ad anello che viene posto, tramite lo studio sull'hash della chiave, tra il nodo "B" e il nodo "C". Con "valori di X" si intende che quei dati si trovano nel nodo "X"

in Cassandra i ha un'architettura distribuita *Peer-to-Peer* che si basa sostanzialmente su una modalità simile a **distributed hash tables (*DHT*)**. Cassandra implementa un meccanismo in cui si garantisce l'**elasticità** in modo **trasparente**, aggiungendo i vari nodi in modo molto efficace e performante. Per farlo si prende lo spazio di indirizzamento delle chiavi e si assegna l'organizzazione prendendo lo spazio di indirizzamento, dividendolo nei nodi disponibili. Tramite ZooKeeper si garantisce anche in questo caso la ridondanza, supportando anche la **rack awareness** (dati possono essere replicati tra diversi rack per "proteggersi" da guasti di macchine/rack). Inoltre non si ha un **single point of failure**.
Analizzando nel dettaglio il partizionamento si ha che i nodi sono divisi in modo logico in una forma ad anello, detta **ring topology**. Si usano quindi i valori hash delle chiavi, associate alle partizioni dati, per assegnare la chiave e i dati collegati ad un nodo dell'anello. L'hashing viene limitato ad un certo valore massimo per permettere la struttura ad anello (banalmente quello che si potrebbe fare anche con una semplice operazione *mod*) e si ha che i nodi con meno carico si "spostano" sull'anello per alleggerire quelli con più carico (inserendo un nodo in modo da "spezzare" i dati in carico ad un certo nodo). L'aggiunta di un nodo influisce solo sul nodo precedente e il nodo successivo. Spesso i nodi vengono replicati almeno 3 volte e viene fatto nei primi 3 nodi successivi al nodo di cui si sta facendo la replica (nell'esempio sopra, ipotizzando voler per semplicità fare solo due repliche di "A", essere sarebbero su "B" e "d"). Le repliche di un nodo vengono fatte ad ogni scrittura dello stesso.

Sapendo che ogni nodo monitora il successivo, qualora questo sia guasto, si può chiedere a quello successivo ancora, garantendo che i dati non vengono persi, il quale a sua volta replicherà su un nodo successivo all'ultimo sui cui aveva repliche per avere sempre le 3 repliche.
Per monitorare i guasti si usano i **protocolli gossip**. L'idea è appunto quella che un nodo chiede al successivo e al successivo ancora se sia vada tutto bene, ottenendo, di passaggio in passaggio, una conoscenza completa e robusta dello stato del cluster (ricordando che essendo una rete *Peer-to-Peer* non si ha alcun master). In caso di failure si procede alla riscrittura dei dati sul nodo successivo tramite il file di log.
Le operazioni di write sono quindi **write ahead log**, prima scrivo quindi sul **commit log** e poi sulla tabella persistente.
Parliamo quindi di consistenza dei dati.
Cassandra nel suo linguaggio offre la **read/write consistency** avendo delle politiche per cui si hanno letture/scritture consistenti:

- read consistency: la lettura è data per certa se un nodo risponde, politica **ONE to ALL** o, altrimenti, se un certo numero di nodi è d'accordo prima di ritornare il risultato della lettura
- write consistency: la scrittura è data per certa se un certo numero di nodi viene aggiornato prima di considerare la scrittura valida, politica **ANY to ALL**, oppure se almeno un nodo mi risponde o anche se tutti i nodi rispondono

Si ha inoltre una politica di **quorum** per il calcolo della "maggioranza" (per la quale anche possono avvenire read/write consistency) che si basa sul **replication_factor**. La richiesta di consistenza viene fatta nelle query:

```
INSERT INTO table (column1,...)
VALUES (value1,...)
USING CONSISTENCY ONE
```

Dove per esempio si specifica, con `USING CONSISTENCY ONE` che basta la conferma di un nodo per validare l'operazione. Potrei usare poi `USING CONSISTENCY QUORUM`, che si basa sul valore k del *replication_factor*, che stabilisce che servono almeno k nodi che validino l'operazione. Con `USING CONSISTENCY ALL` specifico che tutti i nodi devono confermare l'operazione.
Il problema è che se chiedo un solo nodo che verifichi l'operazione rischio di avere inconsistenza, avendo i nodi precedenti che non sono stati magari in grado di comunicare a tale nodo di aver ottenuto i dati.
Posso avere poi `USING CONSISTENCY ANY`, specificando anche il booleano *hinted_handoff_enable*, che autorizza la scrittura anche se si ha il nodo offline

in quanto un'altro nodo prende la richiesta in carico e, non appena il nodo originale torna accessibile, ne trasferisce i dati, per procedere dopo la validazione del primo nodo che risponde.
In merito alle operazioni di *delete* si ha che esse semplicemente rendono il dato non disponibile, essendo più veloce cambiare un flag piuttosto che cancellare. A causa di ciò comunque si creano nelle tabelle fisiche dei problemi di spazio, quindi periodicamente si fanno dei **merge**, ovvero ogni singolo nodo procede alla "compattazione" dei propri dati, sovrascrivendo i valori non più disponibili.
Per assicurare la sincronizzazione dei nodi e per evitare perdita di consistenza si utilizzano dei *checksum* per comparare i dati di un nodo con quelli dei successivi. Per effettuare questo controllo, nel dettaglio, vengono usati degli hash-tree detti **Merkle tree** nel seguente modo:

- vengono mandati degli snapshot dei dati ai nodi successivi
- creati e trasmetti ad ogni "compattazione" principale (non lo dice ma c'è scritto questo nella slide ????)
- se due nodi prendono uno snapshot con un intervallo pari a `TREE_STORE_TIMEOUT` allora i due snapshot sono comparati e in caso di successo i dati vengono sincronizzati

Vediamo nel dettaglio le operazioni di lettura.
Le letture inconsistenti vengono fatte usando `ANY`, posso poi specificare che mi risponda un nodo con `ONE` etc. . ., come per le scritture.
Quindi in lettura, i nodi vengono interrogati fino a quando il numero di nodi che rispondono con il valore più recente non raggiunge un livello di coerenza specificato da **ONE to ALL**. Si ha quindi che:

- se il livello di consistenza richiesto non viene raggiunto i nodi vengono aggiornati con il valore più recente che viene quindi restituito
- se viene raggiunto il livello di coerenza, il valore viene restituito e tutti i nodi che riportavano valori vecchi vengono aggiornati

Si hanno quindi, per esempio:

```
SELECT * FROM table USING CONSISTENCY ONE
```

5.7 Considerazioni finali

SI è notato come bene o male la scelta di algoritmi di gestione siano alla fine sempre gli stessi, variano invece al più le architetture (master/slave *Peer-to-Peer*) ma i vari modelli NoSQL restano comunque da un certo punto di vista (in merito ad eventuali repliche, distribuzione etc...) tutti simili tra loro. La differenza di modello invece consente la scelta di un certo sistema NoSQL in base al tipo di dato che bisogna inserire nel db. Un'altra differenza è il linguaggio usato e la facilità d'uso dello stesso (e delle altre tecnologie legate al DBMS).
La scalabilità è un'altro punto da considerare durante la scelta.