



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

IMPLEMENTAZIONE E TESTING DI UN
CLUSTER DI DATABASE SU UNA RETE DI
PARI

IMPLEMENTATION AND TESTING OF A
PEER NETWORK DATABASE CLUSTER

LINDA LUCIANO

Relatore: *Lorenzo Bettini*

Anno Accademico 2016-2017

INDICE

1	Definizione del problema	5
1.1.1	Cluster di database	6
1.1.2	Rete di pari	9
1.1.3	Sistemi di ridondanza disco (RAID)	10
1.1.4	Codice di correzione errore (Erasure Coding)	18
1.2	Software utilizzato per gli esperimenti	19
1.2.1	PosgreSQL	19
1.2.2	Pglogical	24
1.3	Hardware utilizzato per gli esperimenti	28
2	Definizione del progetto	31
2.1.1	Architettura del progetto	31
2.1.2	Simulazione di un filesystem distribuito (Dati e Metadati)	33
2.2	Considerazioni statistiche sulla ridondanza sul dato	43
2.3	Considerazioni statistiche sulla ridondanza sul metadato	46
3	Definizione del quadro sperimentale	47
3.2	Dati del quadro sperimentale	47
3.3	Lanci di configurazione	48
4	Conclusioni e possibili evoluzioni	55
4.1	Evoluzioni hardware	55
4.1.1	Utilizzo di processori dual core	55
4.1.2	Utilizzo di dischi SSD	56
4.2	Evoluzioni software	56

"Inserisci citazione"

— Inserire autore citazione

1

DEFINIZIONE DEL PROBLEMA

Nell'ambiente *Cloud* l'archiviazione corretta e lo scambio di dati è di vitale importanza. L'obiettivo è quello di ottenere una amministrazione professionale dello *storage*.

È necessario quindi esaminare la gestione e la manutenzione di un insieme di copie di dati, in modo che sia il più efficiente possibile, al fine di tutelare il dato in qualsiasi caso di perdita o di danneggiamento ai dischi.

La replicazione dati è un trasferimento dati unidirezionale da uno o più nodi che permette:

- l'aumento di affidabilità del sistema, rendendo il sistema tollerante ai guasti;
- miglioramento delle prestazioni del sistema, aumentandone la scalabilità.

Tuttavia la replicazione può causare la presenza di eccessive copie di un dato, causando perfino problemi di consistenza. La potenziale modifica di un dato replicato implica che le sue rispettive repliche siano aggiornate.

Per questo è necessario trovare un adeguato compromesso del numero di repliche, pur mantenendo garanzia, sicurezza e buone performance.

Mantenere la disponibilità del servizio realizzato, sostenere correttamente eventuali guasti dei server, rispettare la giusta partizione della rete e gestione delle disconnessioni sono le principali finalità da raggiungere. La tolleranza ai guasti garantisce dati corretti e aggiornati in caso di un eventuale guasto. Per un numero N di guasti occorre avere un numero sufficiente di copie del dato e del metadato, consentendo in questo modo di ottenere nuovamente l'oggetto perso.

In questa tesi viene analizzato qual è il modo più proficuo per eseguire la replicazione di un file in *upload*, allo scopo di distribuire i dati uniformemente.

Oltre a ciò, sono esaminati alcuni lanci di configurazione che hanno portato a una specifica scelta architettonale che garantisca la ridondanza di un dato, mantenendo buone performance.

Un file è suddiviso in *chunk* di varie dimensioni e, rispetto ai metadati, occupano gran parte del disco. I metadati rappresentano l'informazione anagrafica del dato e hanno dimensione fissa.

La replicazione dei metadati, è essenzialmente gestito da Pglogical, implementato come estensione di PostgreSQL.

È un sistema logico di replica, alternativo alla replica fisica, poichè consente di replicare i metadati in modo altamente più efficiente. Segue il modello di *publish/subscriber* e permette l'utilizzo della replica selettiva di database.

Per la copia dei dati la metodologia adoperata è più semplice: sarà sufficiente lanciare una query in parallelo che permette la scrittura dei *chunk* su due differenti dischi.

Grazie a queste configurazioni è possibile ottenere almeno una doppia ridondanza del dato in modo sicuro e corretto, mantenendo il sistema affidabile.

1.1.1 Cluster di database

Un cluster è una raccolta di componenti che garantisce scalabilità e disponibilità distribuendone i costi. Un cluster di database (SQL usa il termine cluster di catalogo) è una collezione di database gestiti da una singola istanza di un server database in esecuzione. Un'istanza è la raccolta di memoria e processi che interagiscono con un database, cioè l'insieme di file fisici che effettivamente memorizzano i dati.[1] A tal fine, è possibile creare un cluster di database per applicazioni *enterprise high-end*, memorizzando e elaborando informazioni sui nodi.

L'architettura per un cluster di database è distinta da come le responsabilità dei dati sono condivise tra i nodi di calcolo.

Seguono due dei vantaggi principali offerti dal clustering, specialmente in un ambiente di database di alto volume:

- *Fault tolerance* (tolleranza ai guasti): in caso di guasto del singolo server, il cluster offre un'alternativa, in quanto esiste più di un server o istanza per gli utenti a cui connettersi;

- *Load balancing* (bilanciamento del carico): la funzionalità di clustering è generalmente impostata per consentire agli utenti di essere assegnati automaticamente al server con il minor carico.[1]

Ci sono differenti tipi di architetture clustering che si diversificano da come vengono memorizzati i dati e allocate le risorse.

La prima modalità di clustering è conosciuta come architettura "*shared-nothing*" (SN). È un'architettura di elaborazione distribuita in cui ogni nodo/server è totalmente indipendente e autonomo, pertanto nessuno dei nodi condivide memoria o archiviazione del disco. Più generalmente, non esiste un unico punto di contesa nel sistema.[3] Il partizionamento è tale che ogni nodo possiede un sottoinsieme dei dati, ovvero ogni nodo ha accesso esclusivo su quel particolare sottoinsieme. [2]

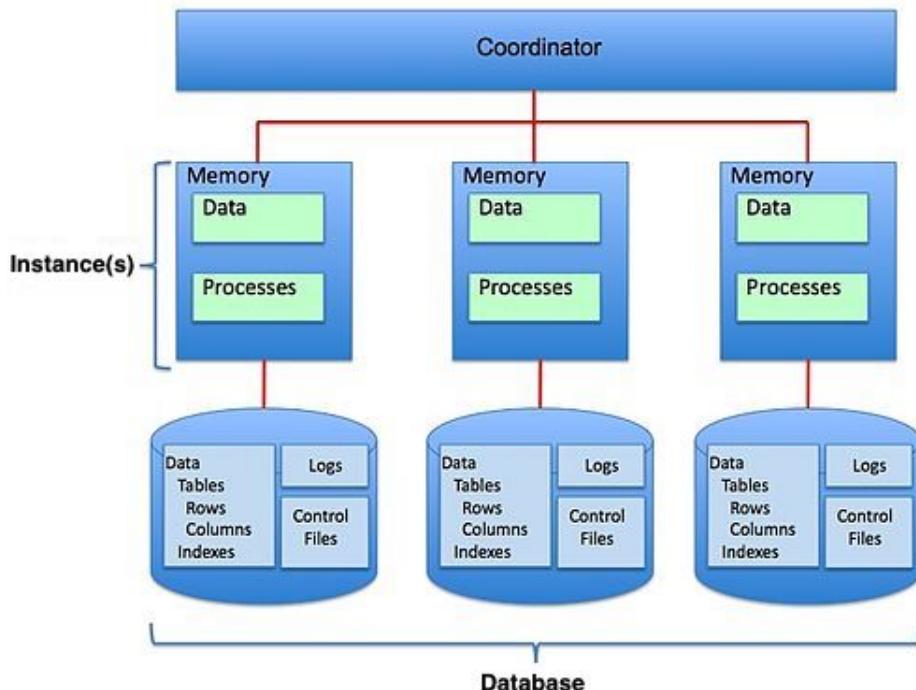


Figura 1: Architettura *shared-nothing*[3].

I vantaggi dell'architettura SN rispetto a un'entità centrale che controlla la rete (un'architettura basata su controller) riguarda l'eliminazione di qualsiasi singolo punto di guasto, consentendo funzionalità di auto-riparazione (*self-healing*) e fornendo un vantaggio nell'offrire aggiornamenti non distruttivi.[5] *Shared-nothing* è anche noto come "*database sharding*".

In generale, un sistema SN divide i suoi dati in vari nodi su database diversi o può richiedere a ciascun nodo di mantenere la propria copia dei dati dell'applicazione utilizzando un qualche tipo di protocollo di coordinamento.[3]

Si oppone a quest'ultima, l'architettura nota come "*shared-disk*" (disco condiviso), in cui tutti i dati vengono memorizzati centralmente in un unico disco e sono accessibili da tutti i nodi di cluster.[4]

In questo tipo di struttura più istanze di database vengono raggruppate in un singolo database sul disco. Nei sistemi di dischi condivisi, i blocchi (o pagine) di dati su disco possono avere un solo proprietario. L'architettura *shared-disk* è un esempio di *Synchronous multi-master*, ovvero ogni istanza del database può scrivere (cioè è un master) in modo sincrono.[4]

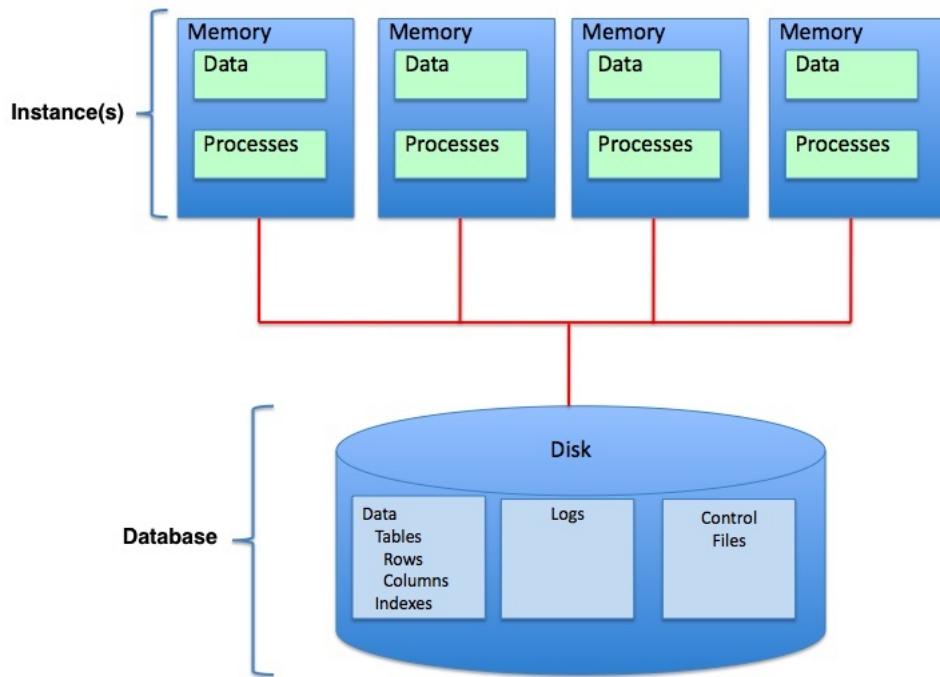


Figura 2: Architettura *shared-disk* [4].

In un'architettura SD, grandi reti di computer possono operare su un singolo set di dati senza la necessità di replicare o bloccare quel set di dati.[4] *Shared disk* ha due vantaggi:

- ogni processore ha la propria memoria;
- il bus di memoria non è un collo di bottiglia (contrariamente all'architettura *shared-everything*);

- il sistema offre un modo semplice per fornire un certo grado di tolleranza agli errori.

La distinzione tra i due tipi è diventata confusa di recente con l'introduzione di distribuzione della cache. In questa configurazione, i dati sono ancora gestiti centralmente, ma controllati da un potente "server virtuale" composto da molti server che lavorano insieme come uno.[2]

1.1.2 *Rete di pari*

Peer-to-peer networking (P2P) è un modello di comunicazione decentralizzato in cui ciascuna unità ha la stessa responsabilità per l'elaborazione dei dati e può avviare una sessione di comunicazione.

A differenza del modello *client/server*, in cui il client effettua una richiesta di servizio e il server soddisfa la richiesta, il modello di rete P2P, noto anche come *peer networking*, consente a ciascun nodo di funzionare sia come client che come server.[6]

Quando una rete P2P viene stabilita su Internet, è possibile utilizzare un server centrale per indicizzare i file oppure stabilire una rete distribuita in cui la condivisione dei file viene suddivisa tra tutti gli utenti della rete che memorizzano un determinato file. Le dimensioni della rete e i file disponibili consentono di condividere enormi quantità di dati.

Le prime reti P2P utilizzavano il *software client* e un server centrale, mentre reti successive, come BitTorrent, hanno eliminato il server centrale e dividono i compiti di condivisione tra più nodi per liberare la larghezza di banda.

Seguono i vantaggi di una rete *peer-to-peer*:

- se un dispositivo collegato interrompe la connessione, il servizio non termina a differenza del modello *client-server*;
- è possibile configurare i computer in gruppi di lavoro *peer-to-peer* per consentire la condivisione di file e altre risorse su tutti i dispositivi. *Peer networking* consente di condividere facilmente i dati in entrambe le direzioni, sia per i *download* sul computer che per gli *upload* dal computer;
- su Internet, le reti *peer-to-peer* gestiscono un volume elevato di traffico di condivisione file distribuendo il carico su più computer. Poiché non si basano esclusivamente sui server centrali, le reti P2P

possono scalare meglio e sono più resistenti delle reti *client-server* in caso di guasti o colli di bottiglia del traffico;

- le reti *peer-to-peer* sono relativamente facili da espandere. Con l'aumentare del numero di dispositivi nella rete, aumenta la potenza della rete P2P, poiché ogni computer aggiuntivo è disponibile per l'elaborazione dei dati.[7]

Il software P2P funge da server e client, il che rende le *peer networking* più vulnerabili agli attacchi remoti rispetto alle reti *client-server*. I dati corrotti possono essere condivisi su reti P2P modificando i file già presenti in rete per introdurre codice dannoso.[7]

1.1.3 Sistemi di ridondanza disco (RAID)

RAID, acronimo di *redundant array of independent disks*, insieme ridondante di dischi indipendenti (originariamente *redundant array of inexpensive disks*), è una tecnologia che permette di memorizzare dati su più dischi rigidi in un computer (o collegati ad esso) in modo da garantire una gestione sicura dei dati.[8]

I dispositivi RAID sono convenienti per sistemi che abbiano necessità di grandi quantità di dati continuamente disponibili.

Il RAID, con modalità differenti a seconda del tipo di configurazione, trae vantaggio dai principi di ridondanza dei dati e di parallelismo in modo da ottenere:

- incrementi di prestazioni (in lettura/scrittura);
 - aumenti nella capacità di memorizzazione disponibile;
 - miglioramenti nella tolleranza ai guasti, ne segue migliore affidabilità.[10]
- Il RAID rende il sistema resiliente alla perdita di uno o più hard disk, permettendo di sostituirli senza l'interruzione del servizio.

I volumi RAID vengono percepiti dal sistema operativo come una singola unità, indipendentemente dal numero di componenti che li costituiscono. Il RAID funziona mettendo i dati su più dischi e consentendo alle operazioni di *input/output* (I/O) di sovrapporsi in modo equilibrato. Poiché l'utilizzo di più dischi aumenta il tempo medio tra i guasti, memorizzare i dati ridondantemente aumenta la tolleranza agli errori.[8]

I dati vengono suddivisi in "*stripes*", ovvero in sezioni di stessa lunghezza (chiamata unità del sezionamento) e sono scritti su differenti dischi. Quando si richiede una lettura di dimensione superiore all'unità di sezionamento, diverse implementazioni di diversi sistemi RAID distribuiscono l'operazione su più dischi in parallelo, aumentando le prestazioni. Ad esempio, se abbiamo sezioni da 1 bit e un array di D dischi, le sequenze di dati lunghe almeno D bit sfruttano tutti i dischi. **DOMANDA preso tutto da wikipedia**

RAID hardware e software

Il RAID può essere implementato sia con hardware dedicato che con software specifico.

Nel primo caso si tratta di unità di controllo che gestiscono tutto autonomamente, facendo in modo che il sistema operativo veda un disco normale. Nel secondo caso, è il sistema operativo che associa i dischi e li gestisce usando una forma di ridondanza attraverso un normale controller (ATA, SCSI, Fibre Channel o altro).

Le unità di controllo RAID sono più costose di quelle normali; tuttavia, se non si creano altri tipi di problemi, hanno il vantaggio di non creare difficoltà al sistema operativo.[8]

Controllore RAID

Un controller RAID è un dispositivo hardware o un programma software utilizzato per gestire unità disco fisso (HDD) o unità SSD (*Solid State Drive*) in un computer o un array di archiviazione in modo da funzionare come unità logica.

Un controller offre un livello di astrazione tra un sistema operativo e le unità fisiche. Presenta gruppi a applicazioni e sistemi operativi come unità logiche per le quali è possibile definire schemi di protezione dei dati.

Dal momento che il controller ha la possibilità di accedere a più copie di dati su più dispositivi fisici, ha la capacità di migliorare le prestazioni e proteggere i dati in caso di *crash* di sistema.[13]

Nel RAID hardware, un controller fisico viene utilizzato per gestire l'array RAID. Il controller può assumere la forma di una scheda PCI o PCI Express (PCIe), progettata per supportare un formato di unità specifico come SATA o SCSI (alcuni controller RAID possono anche essere integrati con la scheda madre).

Un controller RAID può anche essere solo software, utilizzando le risorse hardware del sistema host. Il RAID basato su software generalmente fornisce funzionalità simili a RAID *hardware-based*, ma la sua prestazione è tipicamente inferiore a quella delle versioni hardware.[13]

Livelli RAID

La caratteristica fondamentale che identifica una configurazione RAID è, come citato in precedenza, l'array, che rappresenta il tipo di collegamento logico che c'è tra i vari dischi.

Con tale criterio viene determinato il livello RAID, ovvero la configurazione della tipologia di RAID e stabilito il numero minimo di *hard disk* che sono necessari per attivarlo. A seconda del livello RAID sono implementate diverse caratteristiche operative per ottenere maggiori prestazioni o una maggiore sicurezza dei propri dati oppure entrambe le condizioni.

Si distinguono sei livelli, da 0 a 5.

Questo sistema numerato consente di differenziare le versioni e di scegliere come diffondere i dati attraverso l'array ed è stato suddiviso in tre categorie di livelli RAID:

1. standard;
2. nidificati;
3. non standard.[8]

Successivamente sono descritti i livelli standard e alcuni livelli nidificati.

LIVELLI RAID STANDARD

- RAID 0: livello privo di ridondanza. Si occupa di unire due o più dischi, all'interno dei quali i dati vengono suddivisi equamente (tramite *striping* o sezionamento), in modo da bilanciare anche il carico di operazioni di lettura e scrittura che li riguardano. Livello che consente di realizzare un disco virtuale di grandi dimensioni, più efficiente, ma la rottura di uno dei dischi porta alla perdita di tutti i dati.[8]

Il RAID 0 è noto anche con il nome di *block striping*.

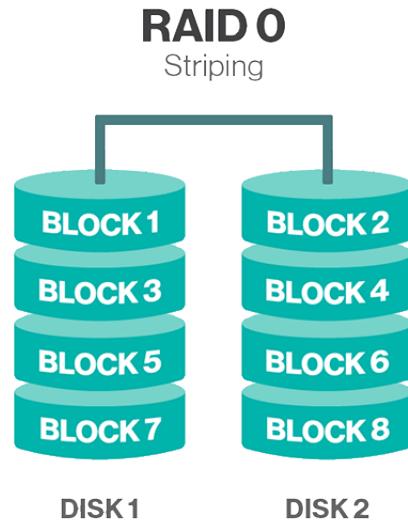


Figura 3: Sezionamento senza ridondanza. Questa configurazione ha sezionamento, ma nessuna ridondanza dei dati. Offre migliori prestazioni, ma nessuna tolleranza agli errori [8].

- RAID 1: livello che si occupa di unire assieme due o più dischi riproducendo fedelmente gli stessi dati. Questa configurazione mantiene quindi almeno una copia esatta di tutti i dati, detta "*mirror*". In questo caso, la rottura di un disco non pregiudica l'utilizzo dei dati che sono disponibili nel disco o nei dischi rimanenti. Più precisamente, l'affidabilità aumenta linearmente al numero di dischi presenti: un sistema con N dischi è in grado di resistere alla rottura di $N-1$ componenti.

La lettura delle prestazioni è migliorata poiché entrambi i dischi possono essere letti contemporaneamente. La scrittura delle prestazioni è la stessa di quella per il singolo disco.[8]

RAID 1 è conosciuto anche come *disk mirroring*. Questo tipo di RAID ha la stessa finalità della replica.

RAID 1

Mirroring

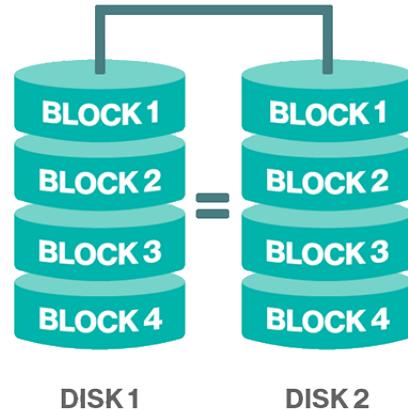


Figura 4: Replicazione. Questa configurazione è costituita da almeno due unità che duplicano la memorizzazione dei dati. Non c'è sezionamento [8].

- RAID 2: livello che divide i dati al livello di bit (invece che di blocco) e usa un *codice di Hamming* per la correzione d'errore che permette di correggere errori su singoli bit e di rilevare errori doppi. Questi dischi sono sincronizzati dal controllore, in modo tale che la testina di ciascun disco sia nella stessa posizione in ogni disco.[10]

RAID 2

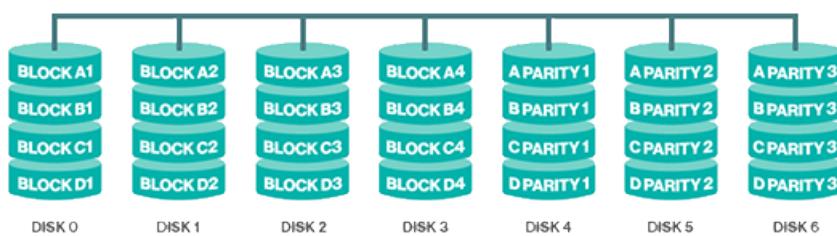


Figura 5: Sezionamento a livello di bit. Questa configurazione ha alcuni dischi che memorizzano le informazioni di errore di verifica e correzione (ECC) [8].

- RAID 3: livello che si occupa di unire assieme almeno tre o più dischi, all'interno dei quali i dati vengono suddivisi equamente, in modo da bilanciare anche il carico di operazioni di lettura e scrittura che li riguardano. Dedicano uno di questi dischi al contenimento di un sistema di codici di controllo, che permettono di ricostruire i dati nel caso in cui uno degli altri dischi si rompa. Le informazioni ECC vengono utilizzate per rilevare gli errori. Il recupero dei dati viene effettuato calcolando l'esclusiva OR (XOR) delle informazioni registrate sulle altre unità.[8]

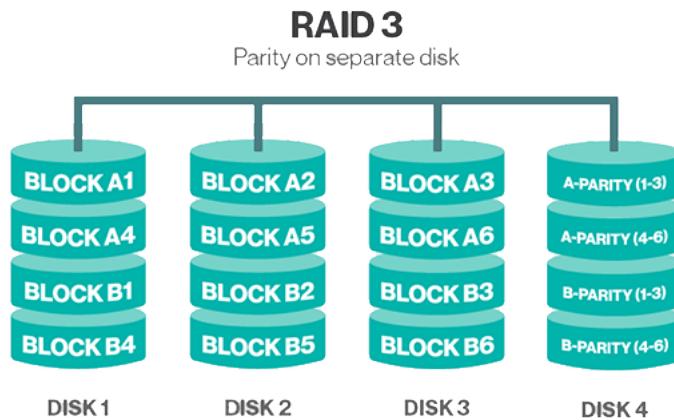


Figura 6: Sezionamento a livello di byte con disco di parità - Questa configurazione utilizza la rigatura e dedica un'unità a memorizzare informazioni di parità [8].

- RAID 4: livello simile al livello tre, con la differenza che i dati vengono distribuiti in modo più efficiente tra i dischi, ma rimane compito di un disco separato il sistema di codici di controllo che permette la ricostruzione dei dati, chiamati "blocchi di parità". Questo livello utilizza grandi sezionamenti, il che significa che è

possibile leggere i record da un'unica unità.[8]

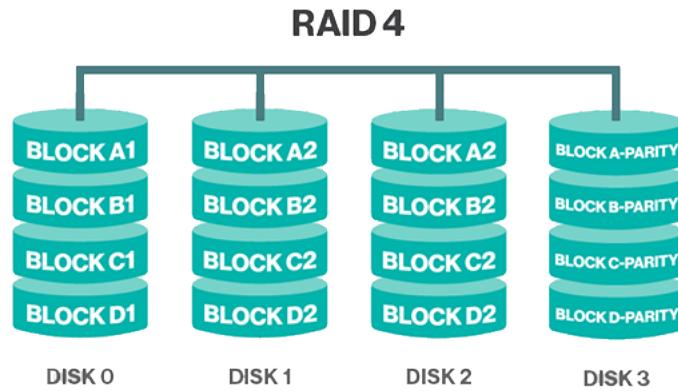


Figura 7: Sezionamento a livello di blocco con disco di parità [8].

- RAID 5: livello basato su livello di blocco con parità che risiedono su ciascuna unità. L'architettura dell'array consente alle operazioni di lettura e scrittura di coprire più unità. Ciò determina prestazioni migliori di quelle di un'unità singola, ma non altrettanto elevate di quella di un array RAID 0.[8]

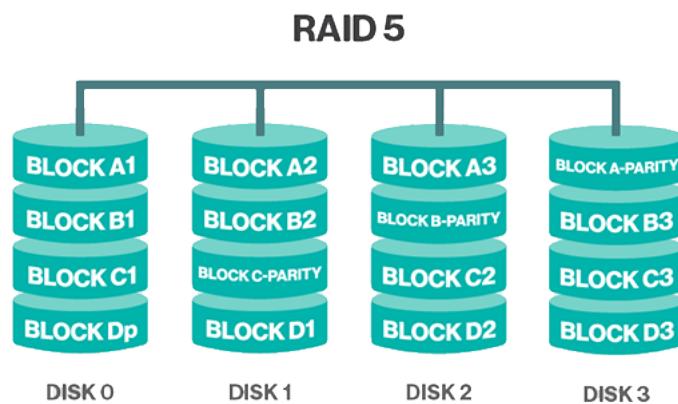


Figura 8: Sezionamento a livello di blocco con parità distribuita [8].

- RAID 6: livello simile a RAID 5, ma include un secondo schema di parità distribuito attraverso le unità nell'array. L'utilizzo di una parità aggiuntiva consente all'array di continuare a funzionare anche se due dischi non funzionano contemporaneamente. Tuttavia, questa protezione supplementare è più costosa. Le matrici RAID 6 hanno un costo superiore a gigabyte (GB) e spesso hanno prestazioni di scrittura più lente degli array RAID 5.[8]

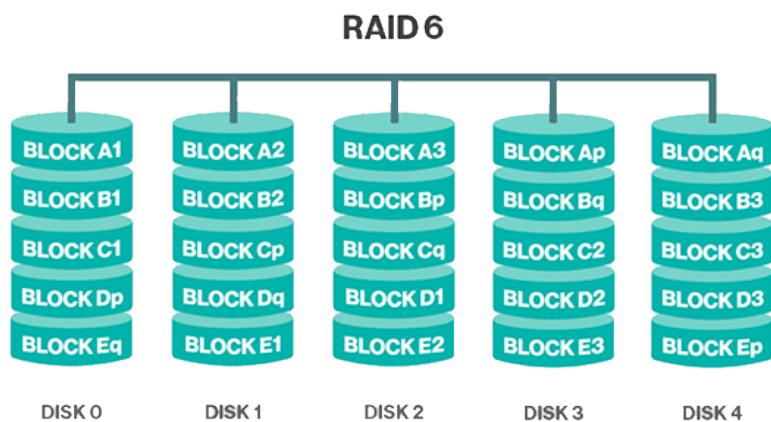


Figura 9: Sezionamento a livello di blocco con doppia parità distribuita [8].

LIVELLI RAID NIDIFICATI I livelli Annidati sono dei tipi di livelli più complessi ottenuti dalla combinazione di alcuni livelli RAID Standard. Esempi classici sono le configurazioni RAID 0+1 o 10.

- RAID10 (RAID 1+0): combinando i RAID 1 e 0, questo livello viene spesso definito RAID 10, che offre prestazioni più elevate rispetto al RAID 1, ma a un costo molto più elevato. In RAID 1+0, i dati vengono clonati e i *mirror* sono suddivisi in sezioni. [8]

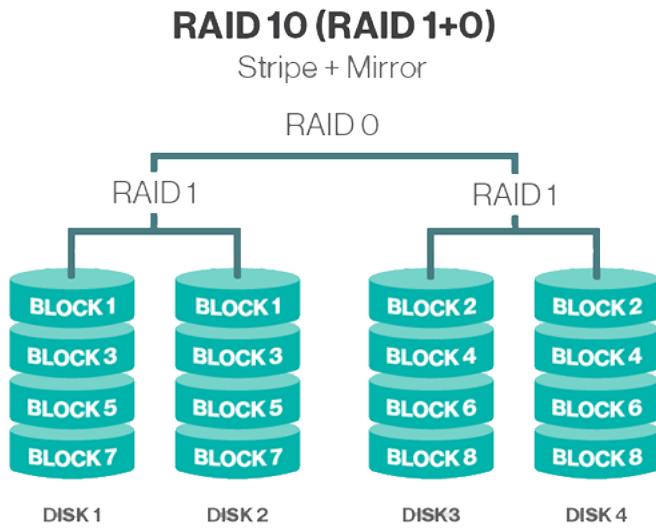


Figura 10: Livello RAID nidificato [8].

1.1.4 Codice di correzione errore (*Erasure Coding*)

La codifica di cancellazione, noto come *Erasure Coding* (EC) è un metodo di protezione dei dati, i quali vengono suddivisi in frammenti, estesi e codificati con pezzi di dati ridondanti e memorizzati su un insieme di posizioni o supporti di memorizzazione diversi.

L'obiettivo della codifica di cancellazione è quello di consentire di ricostruire i dati che vengono danneggiati utilizzando le informazioni sui dati memorizzati altrove nell'array. Lo svantaggio della codifica di cancellazione è che può essere più intenso della CPU e che può tradursi in una maggiore latenza.[11]

La codifica di cancellazione è utile con la presenza di grandi quantità di dati e tutte le applicazioni o sistemi che devono tollerare i guasti, come sistemi di array a dischi, griglie di dati, applicazioni di archiviazione distribuite. Un caso comune di utilizzo corrente per la codifica di cancellazione è in un sistema *object-based cloud storage*.[11]

Come funziona

La codifica di cancellazione crea una funzione matematica per descrivere un insieme di numeri in modo che possano essere controllati per l'accu-

ratezza e recuperati in caso di perdita. Questo è il concetto fondamentale dei metodi di codifica di cancellazione, implementati più frequentemente utilizzando i codici *Reed-Solomon* (tipo di codice lineare (ciclico) non binario di rilevazione e correzione d'errore).[11]

In termini matematici, la protezione offerta dalla codifica di cancellazione può essere rappresentata in forma semplice dalla seguente equazione:

$$n = k + m$$

dove:

- la variabile k è la quantità originale di dati o simboli;
- la variabile m indica i dati aggiuntivi o ridondanti che vengono aggiunti per fornire protezione dai guasti;
- la variabile n è il numero totale di dati creati dopo il processo di codifica di cancellazione.[11]
- la variabile r , chiamata velocità di codice, è definita nel seguente modo:

$$r = \sqrt{\frac{k}{n}}$$

Ad esempio, in una configurazione 10 di 16, o EC 10/16, sei simboli supplementari (m) saranno aggiunti ai 10 simboli di base (k). I 16 frammenti di dati (n) saranno diffusi su 16 unità, nodi o posizioni geografiche. Il file originale potrebbe essere ricostruito da 10 frammenti verificati.[11]

I codici di cancellazione, noti anche come codici di correzione degli errori di avanzamento (FEC), sono stati sviluppati più di 50 anni fa. Da quel momento sono emersi diversi tipi. In uno dei tipi più comuni, *Reed-Solomon*, i dati possono essere ricostruiti utilizzando qualsiasi combinazione di simboli k o pezzi di dati, anche se i simboli m sono persi o non sono disponibili. Ad esempio, in EC 10/16, sei unità, nodi o posizioni geografiche potrebbero essere persi o non disponibili e il file originale sarà ancora recuperabile.[11]

1.2 SOFTWARE UTILIZZATO PER GLI ESPERIMENTI

1.2.1 PosgreSQL

PostgreSQL è un potente sistema *open source* di database relazionale (DBMS, *Database Management System*) cioè è un sistema software proget-

tato per consentire la creazione e manipolazione efficiente di database, ovvero di collezioni di dati strutturati.

Ha più di 15 anni di sviluppo attivo e un'architettura collaudata che ha guadagnato una notevole reputazione per l'affidabilità, l'integrità e la salvaguardia dei dati allocati e la correttezza di archiviazione.

PostgreSQL è un sistema di gestione dei database relazionale (*Object-Relational*, acronimo ORDBMS) basato su Postgres, sviluppato presso l'Università della California presso il Dipartimento di Informatica di Berkeley.[12]

Funziona su tutti i principali sistemi operativi, tra cui Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), e Windows.[12]

Supporta gran parte dello standard SQL e offre molte funzionalità moderne:

- *queries* complesse;
- foreign keys;
- triggers;
- views aggiornabili;
- integrità transazionale;
- controllo della concorrenza multiversione.

Seguono le caratteristiche principali di PostgreSQL; le funzionalità introdotte nella recente versione 9.1 :

- elevata aderenza agli standard SQL;
- architettura *client-server* con una gamma completa di driver e di client;
- progettato in modo altamente concorrente, evitando che i processi in scrittura blocchino i processi in lettura;
- altamente configurabile ed estendibile, consentendo svariati tipi di applicazioni;
- elevate scalabilità e prestazioni, unite a un ampio spettro di possibilità per tarare la configurazione;
- sofisticato ottimizzatore delle query;

- supporto completo per Java, Python, Perl, PHP e molti altri linguaggi, sia per le procedure interne al server di database che per l'accesso da parte di client;
- elevata affidabilità, con una vasta serie di caratteristiche per durabilità e alta disponibilità.[13]

PostgreSQL è progettato per essere estensibile, poiché è possibile definire i propri tipi di dati, tipi di indici e lingue funzionali. Offre un ricco set di strumenti per gli sviluppatori in modo da gestire l'accesso simultaneo ai dati. Inoltre vi è la possibilità di ottimizzarlo per soddisfare le proprie esigenze, tramite uno sviluppo di un plugin personalizzato.

Un database di classe enterprise, PostgreSQL vanta funzionalità sofisticate come il controllo della concorrenza multiversione, il ripristino in tempo reale (*point in time recovery*), *tablespaces*, replica asincrona, transazioni nidificate (*savepoints*), backup in linea/a caldo, un sofisticato query planner/optimizer, ed il *write ahead logging* per una maggiore tolleranza ai guasti.

È estremamente scalabile sia nella quantità pura di dati che può gestire sia nel numero di utenti concorrenti che può ospitare.[12]

Alcuni limiti generali di PostgreSQL sono inclusi nei punti riportati di seguito:

Dimensione massima del database	Illimitato
Dimensione massima Tabella	32 TB
Dimensione massima Riga	1,6 TB
Dimensione Massima campo	1 GB
Numero Massimo Righe per tabella	Illimitato
Numero Massimo colonne per la tabella	250 - 1600
Numero Massimo indici per la tabella	Illimitato[13]

Controllo concorrenza multiversione (MVCC)

MVCC (*Multi-Version Concurrency Control*) è una tecnica avanzata per migliorare le prestazioni del database in un ambiente multiutente, mantenendo la coerenza dei dati.

A differenza della maggior parte degli altri sistemi di database che utilizzano i *lock* per il controllo della concorrenza, Postgres mantiene la

coerenza dei dati utilizzando questo modello.

Ciò significa che durante l'interrogazione di un database ogni transazione vede un'istantanea di dati (una versione del database), indipendentemente dallo stato corrente dei dati sottostanti. Questo protegge la transazione dalla visualizzazione di dati incoerenti che potrebbero essere causati da altri eventuali aggiornamenti simultanei delle transazioni sulle stesse righe di dati, fornendo l'isolamento della transazione per ciascuna sessione del database.[12]

La differenza principale tra i modelli multiversione e di blocco è che nei blocchi MVCC acquisiti per la query (lettura) i dati non sono in conflitto con i blocchi acquisiti per la scrittura dei dati e quindi la lettura non blocca mai la scrittura e la scrittura non blocca mai la lettura.

Caratteristiche PostgreSQL

Mentre PostgreSQL ha un catalogo di sistema completamente relazionale che supporta più schemi per database, il suo catalogo è accessibile anche attraverso lo schema di informazioni definito nello standard SQL.

Le funzionalità di integrità dei dati includono le chiavi primarie (combinate), le chiavi estranee con aggiornamenti e cancellazioni a cascata, i controlli dei vincoli, i vincoli unici e non i vincoli nullo.

Ha anche una serie di estensioni e funzionalità avanzate. PostgreSQL supporta indici composti, unici, parziali e funzionali che possono utilizzare qualsiasi metodo di archiviazione *B-tree*, *R-tree*, *hash* o *GiST*.[12]

Altre funzionalità avanzate includono l'ereditarietà delle tabelle, i sistemi di regole e gli eventi del database. L'ereditarietà di tabelle mette un orientamento orientato all'oggetto sulla creazione della tabella, consentendo ai progettisti di database di derivare nuove tabelle da altre tabelle, trattandole come classi di base.

Di fatto, PostgreSQL supporta sia l'ereditarietà singola che quella multipla.

Il sistema di regole, chiamato anche il sistema di riscrittura delle query, consente al progettista di creare regole che identificano operazioni specifiche per una determinata tabella o vista e le trasformano dinamicamente in operazioni alternative quando vengono elaborate.

Il sistema eventi è un sistema di comunicazione interprocesso in cui i messaggi e gli eventi possono essere trasmessi tra i client utilizzando

i comandi `LISTEN` e `NOTIFY`, consentendo sia la semplice comunicazione *peer-to-peer* sia un coordinamento avanzato sugli eventi del database. Poiché le notifiche possono essere rilasciate da *trigger* e procedure salvate, i client PostgreSQL possono monitorare eventi di database come gli aggiornamenti, gli `insert` o le eliminazioni di tabella quando vengono eseguiti.[12]

Elevata personalizzazione

PostgreSQL esegue procedure memorizzate in più di una dozzina di linguaggi di programmazione, tra cui Java, Perl, Python, Ruby, Tcl, C/C++ e il proprio PL/pgSQL, simile a PL/SQL di Oracle.

I *trigger* e le procedure salvate possono essere scritti in C e caricati nel database come libreria, permettendo una grande flessibilità nell'estensione delle sue funzionalità. Allo stesso modo, PostgreSQL include un *framework* che consente agli sviluppatori di definire e creare i propri tipi di dati personalizzati insieme a funzioni di supporto e operatori che definiscono il loro comportamento.

Il codice sorgente di PostgreSQL è disponibile sotto una licenza libera open source: la licenza PostgreSQL. Questa licenza dà la libertà di utilizzare, modificare e distribuire PostgreSQL in qualsiasi forma, sorgente aperta o chiusa.[12]

Non esiste un unico formato per tutti i software di replica. È necessario capire le proprie esigenze e come si adattino diversi approcci.

La replica delle modifiche dello schema è un problema frequentemente discusso e solo pochi sistemi di database forniscono le estensioni necessarie per implementarlo. PostgreSQL non fornisce la possibilità di definire i trigger richiamati sulle modifiche dello schema, quindi un modo trasparente per replicare le modifiche dello schema non è possibile senza un sostanziale lavoro nel sistema centrale di PostgreSQL.

Il sistema di replica Pglogical avrà un meccanismo per eseguire script SQL in modo controllato come parte del processo di replica.

Write-Ahead Logging (WAL)

Write-Ahead Logging (WAL) è un metodo standard per garantire l'integrità dei dati. Il concetto centrale di WAL è che le modifiche ai file di dati (dove risiedono tabelle e indici) devono essere scritte solo dopo che tali

modifiche sono state registrate, ovvero dopo che i record di registro che descrivono le modifiche sono stati scaricati nella memoria permanente. Se viene seguita questa procedura, non abbiamo bisogno di svuotare le pagine di dati sul disco su ogni *commit* di transazione, perché sappiamo che in caso di *crash* saremo in grado di recuperare il database usando il log: eventuali modifiche che non sono state applicate alle pagine di dati possono essere rifatte dai record del registro. Questo è il recupero *roll-forward*, noto anche come REDO.

Poiché WAL ripristina il contenuto del file di database dopo un arresto anomalo, i *filesystem* registrati non sono necessari per l'archiviazione affidabile dei file di dati o dei file WAL.

L'utilizzo di WAL determina un numero notevolmente ridotto di scritture su disco, poiché è necessario scaricare il file di registro sul disco per garantire che una transazione venga eseguita, piuttosto che ogni file di dati modificato dalla transazione.

Il file di registro viene scritto in modo sequenziale e pertanto il costo della sincronizzazione del log è molto inferiore al costo dello svuotamento delle pagine di dati. Ciò è particolarmente vero per i server che gestiscono molte piccole transazioni toccando diverse parti dell'archivio dati.[12]

1.2.2 Pglogical

Pglogical è un sistema logico di replica implementato come estensione di PostgreSQL. Completamente integrato, non richiede alcun triggers o programmi esterni. Questa alternativa alla replica fisica è un metodo altamente efficiente per replicare i dati utilizzando un modello di *publish/subscriber* per la replica selettiva.[14]

Vantaggi

I vantaggi offerti da Pglogical sono i seguenti:

- replica sincrona;
- replica ritardata;
- risoluzione dei conflitti configurabili;
- capacità di convertire lo *standby* fisico in una replica logica;
- le sequenze possono essere replicate;

- nessun *trigger*; questo significa ridurre il carico di scrittura sul *Provider*;
- nessuna re-esecuzione di SQL significa overhead e latenza ridotti per il sottoscrittore;
- non è necessario annullare le query per consentire alla replica di continuare la riproduzione;
- il sottoscrittore (*subscriber*) può avere diversi utenti e protezione, indici diversi, impostazioni di parametri diversi;
- replica solo un database o un sottoinsieme di tabelle, noto come set di replica (*Replication Sets*);
- replica in versioni o architetture di PostgreSQL, consentendo aggiornamenti a bassa o zero-downtime;
- più server a monte in un singolo *subscriber* per l'accumulo di cambiamenti.[14]

Casi di uso

I diagrammi che seguono descrivono i gestori di database delle funzioni che sono eseguibili con Pglogical:

Migrare e aggiornare PostgreSQL con tempi di inattività quasi a zero:

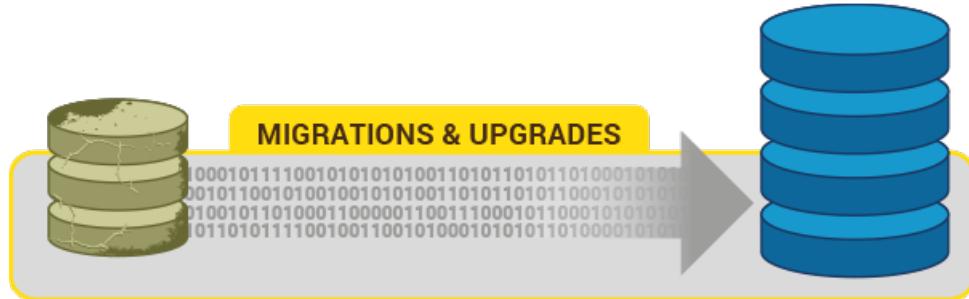


Figura 11: Migrazione e aggiornamenti PostgreSQL [14].

Accumulare le modifiche provenienti da server di database scartati in un data *warehouse*:

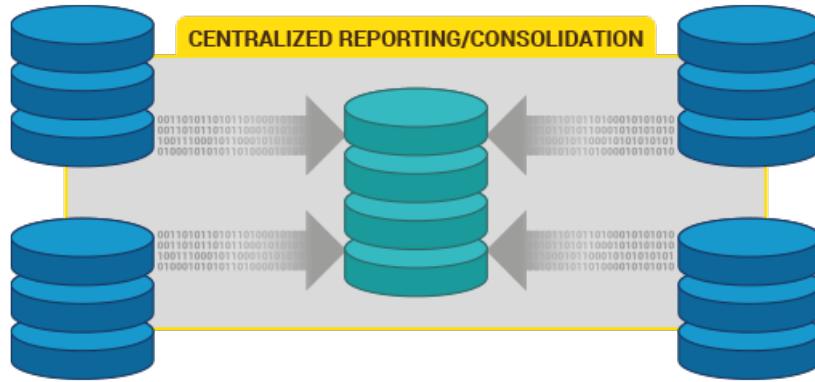


Figura 12: Aggregazione [14].

Copiare tutti o una selezione di tabelle di database ad altri nodi di un cluster:

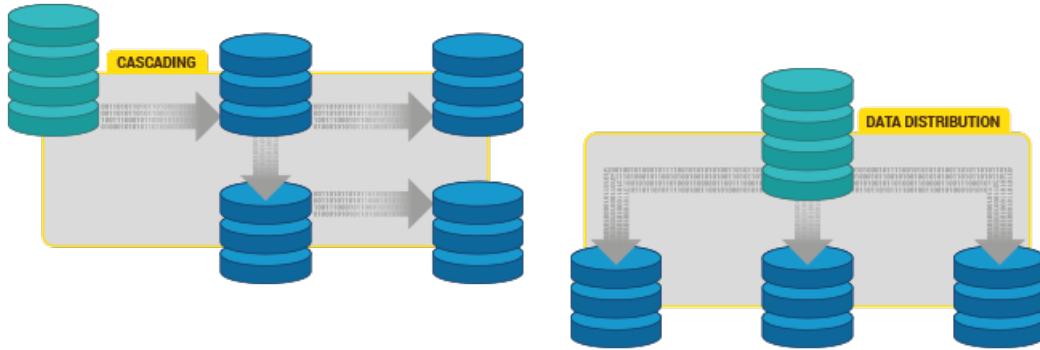


Figura 13: A cascata e distribuzione dati [14].

Le modifiche del database in tempo reale ad altri sistemi:

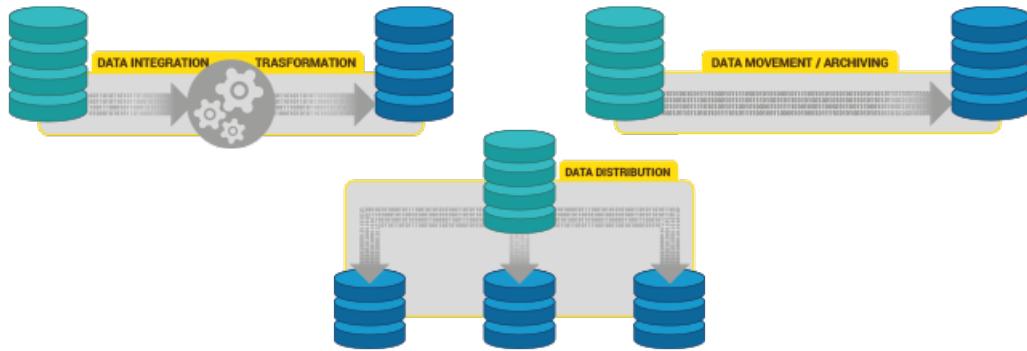


Figura 14: A cascata e distribuzione dati [14].

Come funziona pglogical?

Pglogical utilizza le funzioni di decodifica logica aggiunte da 2ndQuadrant (e disponibili da PostgreSQL 9.4).

Pglogical funziona ancora più veloce con PostgreSQL 9.5 e successive, con bassi *overhead* su entrambi i *provider* e i *subscribers*.

Pglogical si basa molto sulle caratteristiche introdotte nell'ambito dello sviluppo BDR (*Bi-directional replication*), tra cui:

- decodifica logica;
- slot di replica;
- origini di replica;
- impegnano timestamp;
- messaggi WAL logici.[14]

Pglogical non fornisce funzionalità complete di replica *multi-master* e un supporto di modifica dello schema coerente, come fa la BDR. Pglogical incorpora le funzionalità di BDR per produrre una soluzione più semplice da utilizzare per la replica unidirezionale, utilizzabile da più utenti per una vasta gamma di casi di utilizzo.

Lo sviluppo di BDR è utilizzato per scenari che richiedono piena capacità *multi-master*, riutilizzando gran parte del codice da Pglogical.[14]

Utilizziamo i seguenti termini per descrivere i flussi di dati tra host:

- Nodi: istanze del database PostgreSQL
- *Provider/Subscribers*: ruoli presi dai nodi
- *Replication set* o set di replica: una raccolta di tabelle che identificano i dati da replicare.

I casi d'uso supportati sono:

- replica completa del database;
- replica selettiva di insiemi di tabelle mediante set di replicate;
- replica selettiva delle righe della tabella sul lato del *publisher* o del sottoscrittore (*row_filter*);
- replica selettiva delle colonne della tabella sul lato del *master*.[14]

1.3 HARDWARE UTILIZZATO PER GLI ESPERIMENTI

Il lavoro presentato in questa tesi analizza l'esecuzione di test su un determinato tipo di hardware.

Le reti sono orientate in due LAN per avere alta disponibilità del dato (*high-ability*). È possibile contattare lo stesso microserver su due differenti porte Ethernet.

Una semi-unità che supporta fino a 6 microserver e 6 hard disk. Ha alimentazione dedicata a gruppi di 6 e ventole di raffreddamento.



Figura 15: Sei microserver e sei hard disk.

Ci sono due semi-unità per ogni piano *rack*, inserite come illustrato dalla seguente figura:



Figura 16: Due semi-unità.

Segue un'illustrazione di un *rack*. Alla sommità ci sono due *switch* di rete:



Figura 17: Un rack.

2

DEFINIZIONE DEL PROGETTO

2.1.1 *Architettura del progetto*

Concetto Provider/Subscriber

Il modello di *provider/subscriber* definisce un flusso unidirezionale di informazioni da un oggetto *provider* a un numero qualsiasi di oggetti *subscribers*. Il concetto è che il *provider*, anche chiamato *publish*, ha informazioni o eventi utili che devono essere comunicati ad altri oggetti, ovvero tutti i suoi *subscriber*, che useranno tali informazioni per eseguire azioni aggiuntive o rimanere sincronizzati con il fornitore.

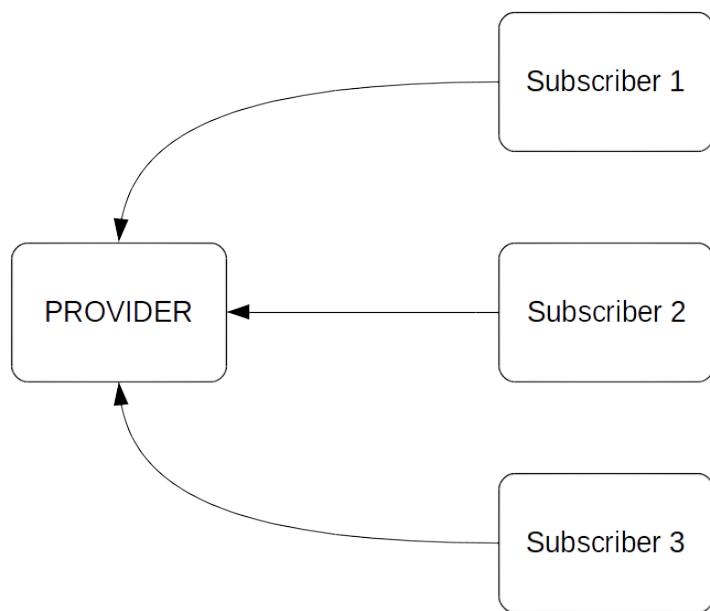


Figura 18: Semplice esempio di un Provider e tre Subscriber abbonati

Come anticipato nel capitolo precedente, la struttura base del sistema di replicazione Pglogical utilizza il modello *publish/subscriber* sopra descritto. Al *provider* sono "abbonati" uno o più nodi *subscribers*. Ogni nodo che riceve i dati di replica da una fonte, quindi *provider*, può essere configurato per essere in grado di inoltrare tali dati agli altri nodi sottoscritti a sè. Utilizzando la replica in cascata, ogni nodo *subscriber* è in contemporanea mittente e destinatario. Più nello specifico, ogni sottoscrittore è anche *provider* di altri *subscribers*.

Ci sono tre idee distinte dietro questa capacità:

1. La scalabilità: un database, in particolare il *publish* che riceve tutte le transazioni di aggiornamento dalle applicazioni client, ha solo una capacità limitata di soddisfare le query dei nodi sottoscritti durante il processo di replica.
2. Limitare la larghezza di banda network richiesta per un sito di backup mantenendo la possibilità di avere più slave nella posizione remota.
3. Essere in grado di configurare scenari di *failover*: in una configurazione da master a slave multipli, è improbabile che tutti i nodi slave siano esattamente nello stesso stato di sincronizzazione quando il master fallisce. Per garantire che uno slave possa essere promosso al master è necessario che tutti i sistemi rimanenti possano concordare lo stato dei dati. Poiché non è possibile eseguire il *rollback* di una transazione confermata, questo stato è indubbiamente lo stato di sincronizzazione più recente di tutti i nodi slave rimanenti.

Aggiunto alle funzionalità di PostgreSQL, che ci permette di replicare un intero database, il sistema di replicazione Pglogical può essere configurato per replicare in modo selettivo le righe di una tabella su entrambi i lati *publisher/subscriber*.

Le sequenze e le tabelle sono raggruppate in modo logico dentro un set di replica (*replication set*).

Ogni set corrisponde ai dati da replicare. È composto da una serie di flussi di dati di replica, che definiremo con R_n (dove n rappresenta la n -esimo set di replica), contenente un gruppo di oggetti da replicare indipendenti da altri oggetti provenienti dallo stesso master. In ogni caso, tutte le tabelle che hanno relazioni che potrebbero essere espresse come vincoli di chiavi esterne e tutte le sequenze utilizzate per generare numeri di serie in queste tabelle dovrebbero essere contenute in uno stesso set.

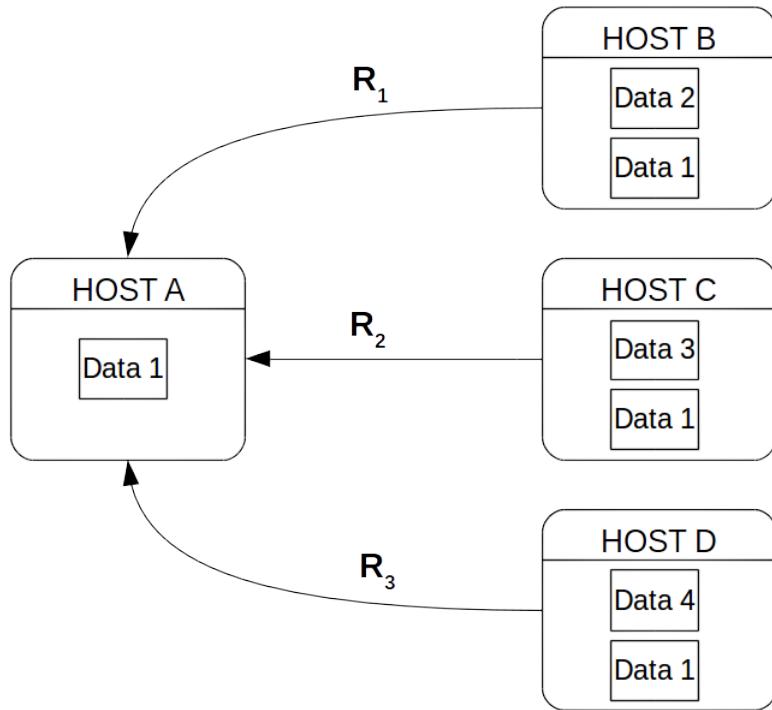


Figura 19: Set di replica o (*replication set*)

La figura illustra un semplice esempio di una configurazione di replica. Il set di replica è composto dal flusso di dati R₁, R₂ e R₃. Questo scenario raffigura quattro host (nel seguente caso quattro nodi) in cui il **NODO A** ha il ruolo di *publish* e i restanti sono i suoi sottoscrittori.

Data 1 rappresenta il flusso di dati nativi di A da replicare. Ciascun *subscriber*, in aggiunta ai propri nativi (Data 2, Data 3 e Data 4), ha esattamente i dati originali del **NODO A**, in quanto abbonati.

In questo modo se fallisce una scrittura di nuovi dati sul *provider*, l'**HOST B**, in quanto suo successivo, può essere promosso come *master* della sottoscrizione di replica.

Simulazione di un filesystem distribuito (Dati e Metadati)

Ogni file viene inserito dentro un *bucket*, che è un insieme di record all'interno di un database, ovvero un oggetto, contenente un insieme di campi o elementi, ciascuno dei quali è identificato da un nome univoco e da un tipo di dato.

Di ciascun file vengono scritte due informazioni:

1. file intero diviso in chunk,
2. una parte di metadato.

È necessario replicare i dati/metadati sulle varie *board* in modo tale che, in casi di *fault*, guasti o perdite, sia possibile ottenere nuovamente il dato originale.

Ci domandiamo quindi: in che modo possiamo ottenere maggiore affidabilità? Quando è scritto un file, per ogni *chunk* e per ogni metadato, viene generato un numero pseudocasuale che definisce un ID. Ogni *board* gestisce un range di ID. L'ID ottenuto è passato a una mappa che restituisce la posizione della *board* a cui appartiene; più precisamente il suo indirizzo IP. Di conseguenza sono identificate anche le *board* precedenti e successive a essa.

Per ottenere una garanzia della replica del dato, è indispensabile replicarlo almeno due volte su *board* differenti. Il dato deve essere quindi scritto sulla *board* a cui appartiene e sul suo successore. Inizialmente una stessa *board* gestiva due range diversi tra loro. Ciascuna *board* conteneva perciò due nodi e quindi due database.

Prendiamo come esempio la BOARD 01, che per semplificare chiameremo B 01 01, dove il primo numero rappresenta lo *chassis* e il secondo identifica la *board*.

La rappresentazione della mappa avrà una forma simile a quella che segue:

BOARD	range ID
B 01 01	0-10
B 01 02	11-20
B 01 03	21-30
B 01 01	31-40
B 01 02	41-50
B 01 03	51-60

Da questo scenario emerge che la BOARD 01 gestisce gli ID compresi tra 0-10 e 51-60.

Il problema sta nel fatto che se la BOARD 01 muore, il successivo e quindi il *subscriber*, sarà in entrambi i casi la BOARD 02.

Ciò è fonte di debolezza; è fondamentale che i dati siano replicati su *board* diverse. Questo implica che ogni nodo di una *board* deve avere nella

mappa differenti successori, allo scopo di garantire una distribuzione dei dati più uniforme.

La gestione delle repliche è coordinata direttamente da PostgreSQL. Quest'ultimo consente la creazione e manipolazione efficiente di database, tuttavia ha la limitazione di replicare in modo totale i dati presenti in un host; altro motivo per cui non è possibile avere una struttura come quella raffigurata precedentemente.

La soluzione è fornita da Pglogical, che permette la replica selettiva di righe o tabelle, rendendo in questo modo agevole la copia su più *board*: una riga di una tabella può essere replicata sulla BOARD 02 e un'altra su una diversa *board*.

Nello scenario descritto precedentemente, in cui sono configurate due nodi per *board*, contemporaneamente posso avere due connessioni (ossia due client differenti eseguono due transazioni parallele in simultanea), che richiedono la scrittura su due tabelle della stessa *board*. Ad esempio: due client vogliono scrivere un dato con ID 8 e 58 con due transazioni parallele; i client si aspettano una risposta di successo una volta avvenuta la scrittura sulla BOARD 01 e sulla sua successiva (configurabile dal parametro `synchronous_commit` di PostgreSQL). Questo non è possibile, poichè Postgres, come detto, replica un intero database. L'occupazione del disco dei metadati rispetto a quello dei dati è decisamente minore, infatti il grosso dell'occupazione del disco sono i *chunk* del file. Di fatto i metadati hanno grandezza fissa; l'unico metadato che cresce è la lista di un file presente in un *bucket*. I metadati possono cambiare a runtime: c'è la necessità di gestire la replica tramite Pglogical. I dati, invece, quando sono scritti non cambiano nel corso del tempo, poichè hanno dimensione fissa.

Per questo motivo è stata fatta la seguente distinzione:

- la mappa che coordina i *chunk* di un file, quindi i dati, gestisce più nodi per *board*,
- la mappa dei metadati, gestisce un solo nodo per *board*, un solo database, quindi una sola replica (sincrona).

Siamo arrivati alla conclusione che i database per i dati e i metadati devono essere separati.

Ogni *board* ha otto database di dati, che corrispondono a otto nodi, e uno solo di metadati.

Scrivere la ridondanza dei dati è di conseguenza più semplice:

1. viene calcolato l'ID del *chunk* e il rispettivo contenuto;

2. una funzione, a cui viene passato l'ID ottenuto, restituisce dalla mappa la *board* con il range di cui fa parte e il successivo host. Nel caso in cui una *board* è morta, c'è un meccanismo per cui viene restituita quella successiva ancora (fino ad arrivare a un massimo di 5 *board*, che corrisponde alla massima catena di replica);
3. sono aperte due connessioni;
4. il *frontend* si connette a due Postgres, lanciando una query in parallelo, che permette la scrittura dei *chunk* su due host differenti;
5. sono chiuse le connessioni.

figura con due host e query in parallelo Se tutte le *board* sono morte, non si effettua alcuna scrittura. Di fatto se quattro *board* sono già ho solo una *board* per scrivere il dato, non rispettando la necessità di replicarlo almeno due volte.

Tuttavia la probabilità di avere quattro *board* consecutive morte è decisamente bassa.

Di conseguenza le repliche sono utilizzate solo per quanto riguarda i metadati.

Se la BOARD 01 è offline per un periodo di tempo, i metadati che devono essere replicati, sono scritti sugli host successivi; il primo *subscriber* della BOARD 01 diventa appunto *provider* di quel set di replica. Quando torna online la BOARD 01 non è aggiornata con gli ultimi dati replicati.

Sarebbe utile un meccanismo di versionamento; tuttavia è troppo complesso.

La replica selettiva di Pglogical consente la giusta soluzione. In questo modo la replica è coordinata direttamente da PostgreSQL. Il *frontend* si connette a un host, dopodiché è il database che si replica con le funzionalità offerte da Postgres.

Segue un esempio di come un metadato è rappresentato nella mappa:

2a01:84a0:1001:a001::2:2;70;80;70

- 2a01:84a0:1001:a001 identifica la posizione del host, quindi l'indirizzo IP della *board*;
- la *board* è la 2:2 (B 02 02): *chassis* 02 e *board* 02;
- 70;80 è il range gestito dal host individuato;

- l'ultimo numero, 70, rappresenta il nome identificativo del database.

Consegue la rappresentazione del host preso in considerazione:

2:2	70
•	<u>70</u>
•	60
•	50
•	40
•	30

Figura 20: BOARD 02 02": 2a01:84a0:1001:a001::2:2;70;80;70".

Prendiamo in considerazione le *board* successive all'host appena visto:

```
2a01:84a0:1001:a001::2:2;70;80;70
2a01:84a0:1001:a001::2:3;80;90;80
2a01:84a0:1001:a001::2:4;90;a0;90
2a01:84a0:1001:a001::2:5;a0;b0;a0
2a01:84a0:1001:a001::2:6;b0;c0;b0
```

La BOARD 02 02 è *provider* dei suoi quattro host successivi (di conseguenza anche i predecessori, che non sono presenti in questa mappa d'esempio).

Illustriamo il set di replicazione sottoscritta:

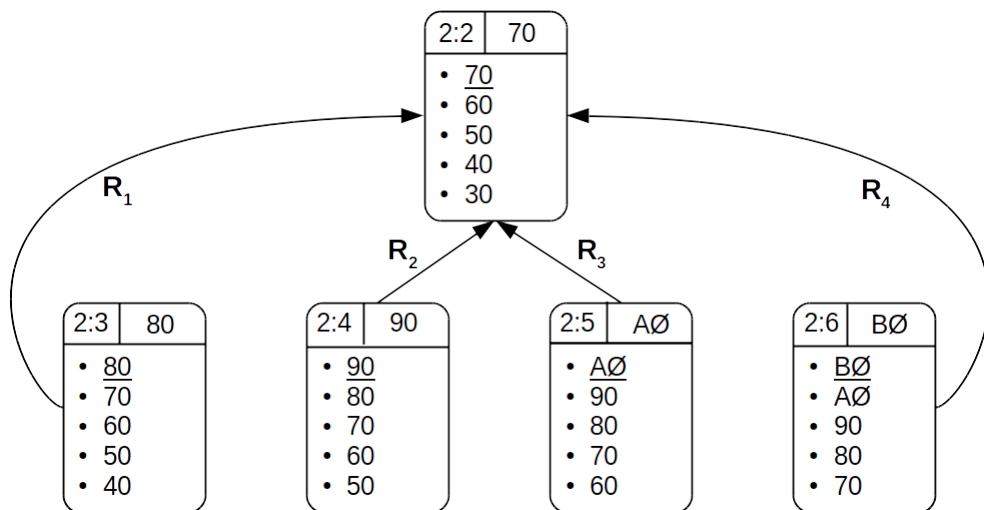


Figura 21: Set di replica della BOARD 02 02" visto come *provider*.

È stato configurato che il metadato viene replicato su quattro host differenti sottoscritti alla BOARD 02.

R_1 replica i metadati con ID 70, 60, 50, 40, R_2 70, 60, 50, R_3 70, 60 e R_4 70.

A sua volta, l'host preso in considerazione, funziona come *subscriber* a un altro set di replica; come segue:

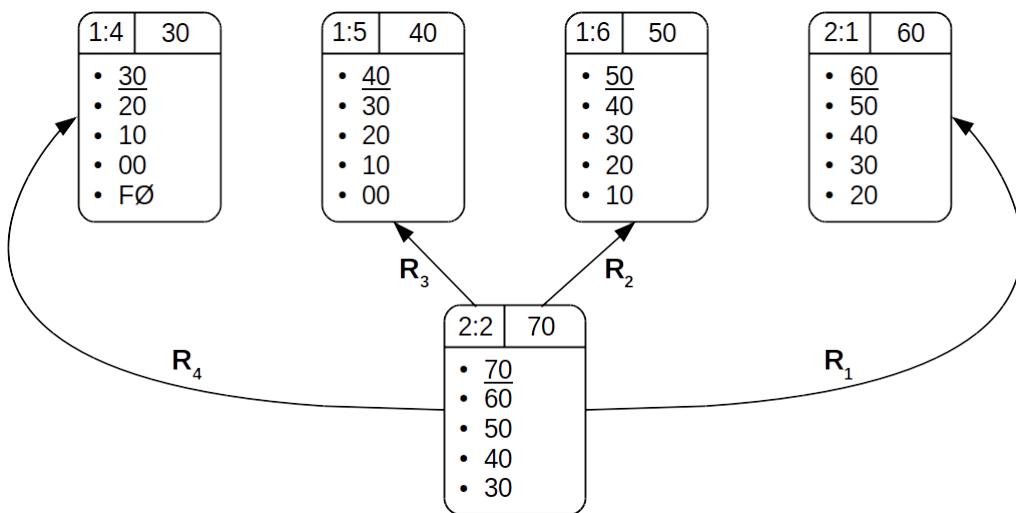


Figura 22: Set di replica della BOARD 02 02" visto come *subscriber*.

In questo modo abbiamo la certezza che il metadato sia replicato su tutti i nodi (quindi su tutte le *board*) a lui "abbonate". In contemporanea ogni database funge da *subscriber* di altri nodi.

Native subscriptions

Fino a questo punto è stato descritto come funzionano le repliche.

Consideriamo il caso in cui la BOARD 02 02 sia offline per un periodo di tempo. Se il client prova a connettersi al database della BOARD 02 per scrivere un metadato che appartiene al range 70-80, vede che la *board* non risponde. Il metadato viene quindi scritto sulla *board* immediatamente successiva alla 02 nella mappa; di fatto la B 02 03 diventa il nuovo *master* e il metadato viene replicato quattro volte, invece di cinque.

A un certo punto la BOARD 02 torna disponibile, di conseguenza tutti i metadati che appartengono a quel range possono essere scritti normalmente e replicati come illustrato nella *figura 1.17*. Questo avviene però solo nel caso in cui la *board* è aggiornata.

Il problema sta nel fatto che non è certo che l'host 02 sia sincronizzato correttamente con gli altri nodi, poiché gli eventuali nuovi metadati scritti si trovano nel database della BOARD 03.

Pglogical permette di gestire le repliche tra un nodo *master* e una serie di *slaves*. Nel nostro caso, è configurato per coordinare tanti nodi *master*.

Per renderlo *multi-master* sono state realizzate le repliche native (*native subscriptions*) che hanno direzione opposta alle repliche appena descritte. Quando la BOARD 02 diventa nuovamente online, può ottenere i nuovi metadati scritti sul nodo successivo attraverso una fase di sincronizzazione.

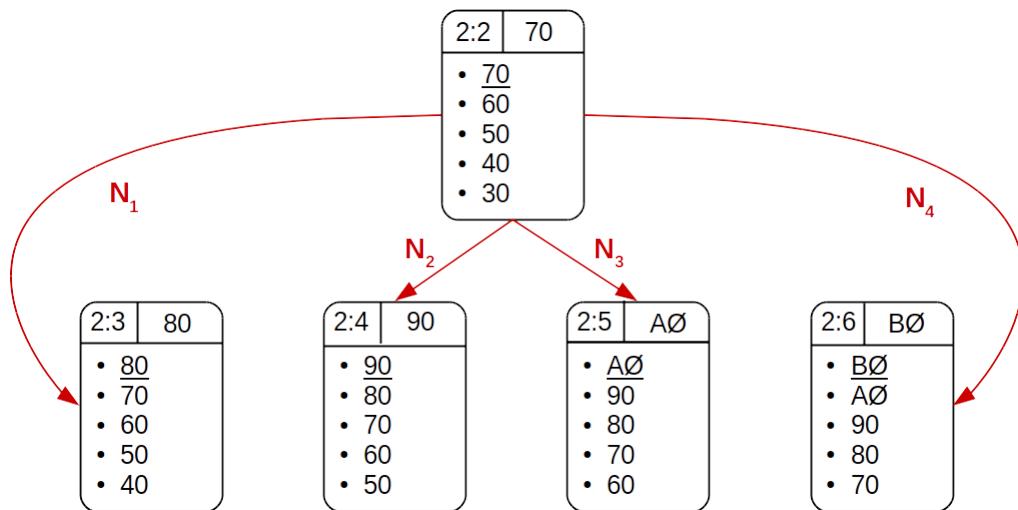


Figura 23: *Native subscriptions* della BOARD 02 "visto come provider".

Di conseguenza:

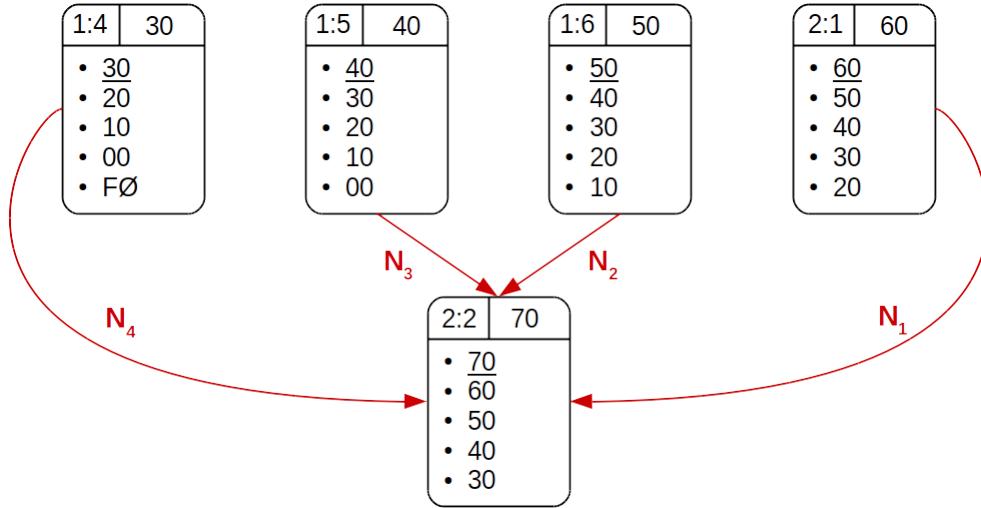


Figura 24: *Native subscriptions* della BOARD 02 02" visto come *subscriber*.

Replica sincrona e asincrona

Quando la scrittura di un dato è andata a buon fine, viene confermata al client con un messaggio di risposta di successo. Come abbiamo detto, per una questione di affidabilità, deve essere garantita la replicazione di un dato almeno due volte. Ciò è permesso da un importante funzionalità di Postgres: il `synchronous_commit`.

Ci sono due tecniche di repliche che esamineremo:

- replica asincrona
- replica sincrona

La replica asincrona è un approccio *store and forward*. È la modalità di archiviazione dei dati in cui i dati non vengono immediatamente sottoposti a backup; di fatto non aspetta che lo storage primario confermi la completa scrittura del dato sul disco. La replica asincrona ha come obiettivo di eseguire la copia di un dato in un determinato periodo di tempo prestabilito. Più precisamente, il client ha la conferma di avvenuto successo quando il dato da replicare è stato scritto sul WAL. Questo metodo si traduce in un sistema con buone prestazioni e minori requisiti di larghezza di banda, poiché i dati non vengono replicati in backup in tempo reale. Ciò non garantisce però che il dato sia immediatamente

disponibile, di conseguenza, questo tipo di approccio dovrebbe essere usato per dati o informazioni meno sensibili che hanno tolleranza alla perdita. Nel caso in cui, in una minima frazione di tempo, muore una *board*, il dato è tracciabile solo sul WAL. La sua latenza di rete e la tolleranza della larghezza di banda lo rendono adatto per la replica a lunga distanza.

La replica sincrona viene utilizzata principalmente per le applicazioni transazionali di fascia alta che richiedono il *failover* istantaneo in caso di guasto del nodo primario. Questa tecnica è preferibile per le applicazioni con obiettivi a basso tempo di recupero che non possono tollerare la perdita di dati.

Il seguente approccio ha i suoi svantaggi. La replica sincrona è più costosa di altre forme di replica dei dati, introduce la latenza che rallenta l'applicazione principale e funziona solo a brevi distanze. Il vantaggio risiede nel fatto che garantisce la scrittura della replica su un'altra *board*, comunicando al client che la copia è avvenuta con successo.

La differenza principale tra la replica sincrona e la replica asincrona è il modo in cui i dati vengono scritti nella replica. La maggior parte dei prodotti di replica sincrona scrive i dati nello storage primario e nella replica contemporaneamente. In quanto tale, la copia primaria e la replica dovrebbero rimanere sempre sincronizzati.

Al contrario, i prodotti di replica asincrona scrivono prima i dati nella memoria primaria e quindi copiano i dati nella replica. Sebbene il processo di replica possa verificarsi quasi in tempo reale, è più comune che la replica si verifichi su base pianificata.

Il nostro scopo è di trovare un giusto compromesso tra garanzia e velocità della replicazione di dati.

Parametri del file di configurazione di PostgreSQL

Analizzeremo ora i parametri che possono essere impostati sul *master* che deve inviare i dati di replica ai suoi *slaves* "abbonati".

La replica del metadato su almeno due *board* è realizzabile utilizzando l'attendibilità della tecnica di replica sincrona, ottenuta tramite PostgreSQL, con la configurazione del parametro `synchronous_commit`.

È uno dei parametri più importanti con una quantità di opzioni sopra la media. Specifica se il `commit` della transazione attende che i record WAL

vengano scritti sul disco prima che il comando restituisca un'indicazione "riuscita" al client.

I valori validi sono `on`, `remote_write`, `local` e `off`. Quando una *board* è *offline*, può verificarsi un ritardo tra il momento in cui viene segnalato il successo al client e quando la transazione è realmente garantita per essere sicura contro un arresto anomalo del server.

L'impostazione di questo parametro su `off` non crea alcun rischio di incoerenza del database: un sistema operativo o un arresto anomalo del database potrebbe causare la perdita di alcune transazioni presunte recenti. Pertanto, disattivare `synchronous_commit` può essere un'alternativa utile quando le prestazioni sono più importanti della certezza esatta sulla durata di una transazione.[13] Nel nostro caso il `synchronous_commit` è impostato a `on`. Ciò garantisce che il `commit` delle transazioni attenda sempre che i dati vengano realmente scaricati nel log delle transazioni (ovvero nel WAL) assicurandosi che la transazione sia realmente persistente. Più nello specifico, ciò garantisce che la transazione non vada persa a meno che sia il *master* che lo *slave* subiscano un danneggiamento della memoria del database. Nella modalità di replica dello streaming sincrono anche la replica deve fare lo stesso.

Il `synchronous_standby_names` specifica l'elenco delle sottoscrizioni di replica che sono in grado di supportare la replica sincrona. È stato impostato come segue:

```
synchronous_standby_names = '1 (r1, r2, r3, r4)'
```

In qualsiasi momento ci sarà una replica sincrona; le transazioni in attesa di `commit` saranno autorizzate a procedere dopo che questo server di *standby* confermerà il ricevimento dei loro dati. Lo *standby* sincrono è il primo nominato in questo elenco. Altri server di *standby* che appaiono in seguito rappresentano le potenziali repliche sincrone; nel caso in cui `r1` non possa permettere la replica, disconnettendosi per qualsiasi motivo, verrà sostituito con il successivo *standby* presente in questo elenco, che diventerà la nuova replica sincrona.

Impostare una sola replica sincrona può essere il giusto compromesso tra affidabilità e velocità di prestazioni. Il metadato, infatti, sarà copiato sicuramente due volte su host differenti: quello nativo e un sicuramente un suo *subscriber* tramite la replica sincrona.

Questi parametri possono essere impostati solo nel file di configurazione di PostgreSQL `postgresql.conf`.

CONSIDERAZIONI STATISTICHE SULLA RIDONDANZA SUL DATO

L'occupazione dei dati del disco è maggiore rispetto a quella dei metadati, in quanto un dato rappresenta il contenuto del file stesso.

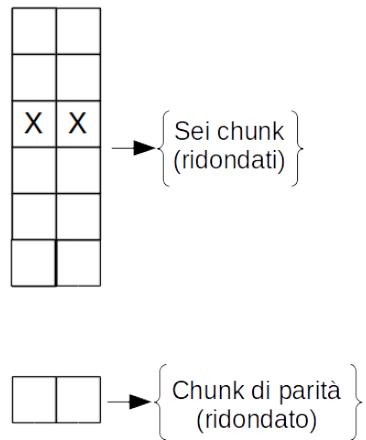
Durante l'*upload* il file viene suddiviso in *chunk* di varie dimensioni. Ogni *chunk* è ridondato due volte su due diversi host. Oltre la ridondanza, tramite lo XOR byte a byte di sei *chunk* viene costruito un *chunk* chiamato di parità, ridondato anch'esso due volte. Nel caso sia perso un *chunk* appartenente al set di parità è possibile ricostruirlo tramite il *chunk* di parità, eseguendo l'operazione inversa.

L'utilizzo di algoritmi di parità sono applicati solo a file di svariati *chunk*. Di fatto se è caricato un file di solo un *chunk* di grandezza d , l'occupazione sarà $4 \cdot d$. Se il file è di 6 *chunk* l'occupazione è di $14d$, di cui $6 \cdot 2 \cdot d$ sono ottenuti dalla ridondanza dei 6 *chunk* e $2 \cdot d$ dalla ridondanza di parità.

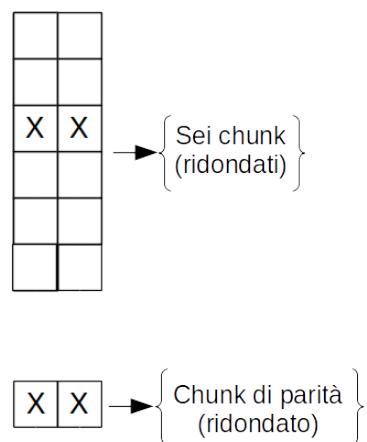
È sconveniente di conseguenza usare questo metodo di parità per file di solo un *chunk*, poiché il rapporto tra dati uploadati e dati salvati è di 4 rispetto al rapporto di 2,3 nel caso di file di chunk.

Ciascun *chunk* è ridondato su dischi differenti.
Per causare la perdita di un dato, quanti dischi possono essere danneggiati?

La seguente figura illustra il caso in cui è perso un *chunk*, quindi anche la sua ridondanza. Questo scenario garantisce tuttavia la possibilità di ricostruire il dato, mantendo di conseguenza la sua sicurezza:

Figura 25: Perdita di un *chunk*

Seguono i due scenari in cui può accadere che un dato non possa essere recuperato. Se viene perso un *chunk* e il *chunk* di parità (quindi la sua ridondanza) non è realizzabile ricreare il dato (figura 25). Ugualmente se sono persi due *chunk* del set di parità (figura 26).

Figura 26: Perdita di un *chunk* e del *chunk* di parità

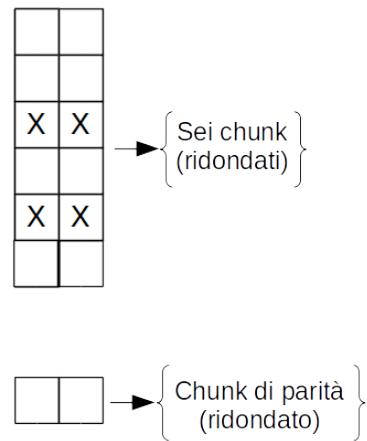


Figura 27: Perdita di due *chunk*

Nel peggior dei casi la perdita di 4 dischi causa l'impossibilità di ricostruire un dato. La probabilità che si rompano 4 dischi su 228 è:

$$\frac{1}{228} \cdot \frac{1}{227} \cdot \frac{1}{226} \cdot \frac{1}{225}$$

Tuttavia ancora non corrisponde al caso peggiore. Questo calcolo non tiene conto che i dischi si guastano con una probabilità di un certo tipo; solitamente i dischi si rompono con probabilità uniforme del 2%.

Affinchè questo accada è necessario considerare il seguente evento distribuito nel tempo.

Ad esempio consideriamo che in un anno si rompono 2 dischi su 100. Se viene valutato un giorno ottengo:

$$\frac{100}{365} \cdot 0,02$$

che corrisponde alla probabilità che mi si rompa il disco quel preciso giorno.

Consideriamo un target di 3 ore per ricostruire il dato; calcoliamo la probabilità che un disco si rompa in queste tre ore:

$$\frac{\frac{100}{365} \cdot 0,02}{24} \cdot 3$$

Se il disco si rompesse dopo le 3 ore considerate, il dato verrebbe ricostruito quindi la catena di probabilità che rappresenta la rottura di 4 dischi si annullerebbe. Affinchè ci sia la perdita di un dato deve verificarsi la rottura di 4 dischi in 3 ore.

Fissato un tempo, nel nostro caso un giorno, la probabilità che si danneggino tutti e quattro i dischi è:

$$\frac{1}{228} \cdot \frac{1}{227} \cdot \frac{1}{226} \cdot \frac{1}{225} \cdot \frac{\frac{100}{365} \cdot 0,02}{24} \cdot 3$$

La probabilità che non si possa ricostruire un chunk in tre ore è quindi un numero veramente basso.

Un danneggiamento di un file avviene anche quando è perso anche solo un chunk. La probabilità congiunta che tutti i file siano presenti e non siano andati persi rappresenta la probabilità di avere tutto il file integro.

Questo varia sui file di piccole dimensioni e sui file di maggiori grandezza è una probabilità più bassa.

La considerazione è stata fatta nel caso peggiore. In realtà la probabilità è ulteriormente più bassa, in quanto ci sono altre probabilità sfavorevoli che hanno a loro volta probabilità maggiore.

La perdita di un dato è di conseguenza realmente difficile.

CONSIDERAZIONI STATISTICHE SULLA RIDONDANZA SUL METADATO

Il metadato è di piccole dimensioni, poiché rappresenta l'informazione anagrafica del dato. Per questo motivo non è conveniente utilizzare metodi di parità per ricostruire l'informazione anche se è perso in parte. Per il metadato è sufficiente creare quattro repliche, garantendo ugualmente la stessa sicurezza.

La perdita del metadato si verifica soltanto se tutte e 5 le *board* contenente l'informazione muoiono.

3

DEFINIZIONE DEL QUADRO SPERIMENTALE

Al principio i dati dovevano essere trattati con la stessa configurazione architetturale dei metadati, utilizzando il modello di *publsh/provider* e le sottoscrizioni sincrone e asincrone.

I lanci di configurazione che seguono hanno condotto verso un tipo di architettura differente per gestire la ridondanza del dato.

Dai risultati dei test è comprensibile la preferenza di un'altra scelta: utilizzare la stessa replicazione del metadato non è conveniente.

Inizialmente, è stato possibile notare che utilizzando Postgres, la scrittura di un dato su un database, apriva una connessione a un altro nodo inviandogli la copia del dato in modo sincrono. Questo causava lente prestazioni.

La libreria utilizzata scritta a codice, per cui veniva ridondata soltanto una copia, causava eccessivi rallentamenti.

Ciò ha portato a un cambiamento del codice, ottenendo in questo modo la configurazione attuale: il dato viene scritto in due database e le risposte di successo o eventuale fallimento, sono gestite direttamente da codice; è restituito un rollback se la ridondanza non è effettuabile, in caso contrario è ritornato un commit soltanto se entrambi i database hanno scritto la replicazione del dato.

3.2 DATI DEL QUADRO SPERIMENTALE

I dati sono stati ricavati dai log di *nginx*.

Nginx è un *web server/reverse proxy* leggero ad alte prestazioni e fornisce in modo rapido i contenuti statici con un utilizzo efficiente delle risorse del sistema.

Ciò è interessante in quanto rende la misura maggiormente obiettiva.

Di fatto i risultati non sono ottenuti da uno strumento "imparziale", al contrario da terze parti, quale nginx che non sfavorisce né favorisce.

Dai log di nginx sono state prese in considerazione le PUT dei *chunk*. In particolar modo è stato necessario controllare il tempo di quando ogni pacchetto entra all'interno dello *storage* a quando esce (tempo dato in ms). Questo tempo è stato successivamente raggruppato al decimo di secondo.

I dati sono di fatto il numero dei pacchetti, ossia la frequenza di PUT registrate da nginx, e il tempo impiegato da ogni dato.

3.3 LANCI DI CONFIGURAZIONE

Per stabilire questa decisione architettonale sono stati esaminati vari lanci con configurazioni diverse.

In parte dei casi è stato influente, altre sono state peggiorative, fino a che non abbiamo ottenuto il risultato finale.

Esaminiamo i principali lanci di configurazioni che ci hanno portato a queste riflessioni.

Lancio di configurazione 1

Nel primo lancio è partecipe una singola istanza Postgres.

Non è presente nessuna copia di ridondanza. Questo rappresenta il caso ottimale, indubbiamente il migliore; tuttavia non ha vantaggi in quanto non ridonda i dati.

In aggiunta sono state aumentate il numero di connessioni a 60 (invece che a 40 come in tutti gli altri lanci di configurazione) arrivando in questo modo a ottenere 133Mbit/sec.

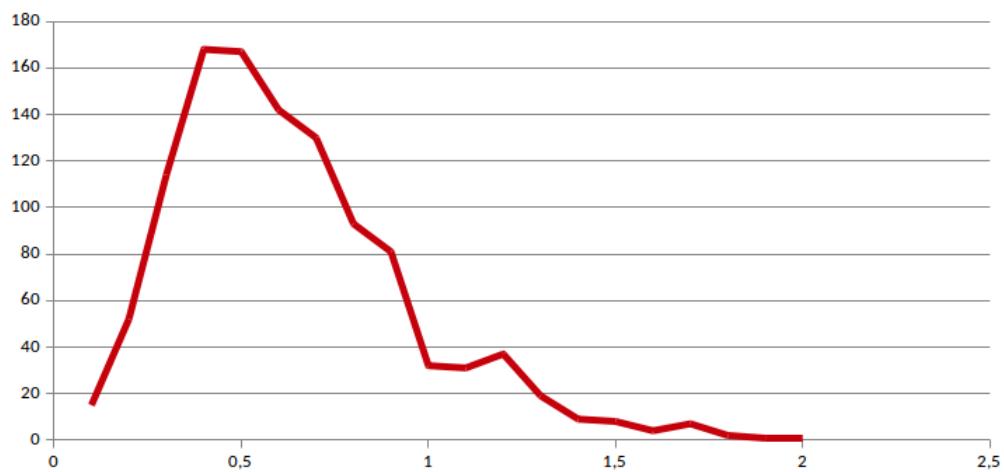


Figura 28: Singola istanza Postgres. Nessuna copia di ridondanza.

Lancio di configurazione 2

Questo lancio è stato utile in quanto è stato possibile scoprire che le connessioni non erano parallele ma sequenziali.

Le chiamate avvenivano per mezzo di una libreria scritta che però non rendeva le connessioni parallele. Così facendo le connessioni per scrivere il dato erano sequenziali. Le connessioni erano di fatto aperte soltanto su richiesta e questo provoca un ritardo dei tempi essenziale.

Sono state configurate due sottoscrizioni sincrone e questo ha portato ad ottenere senza dubbi il caso peggiore.

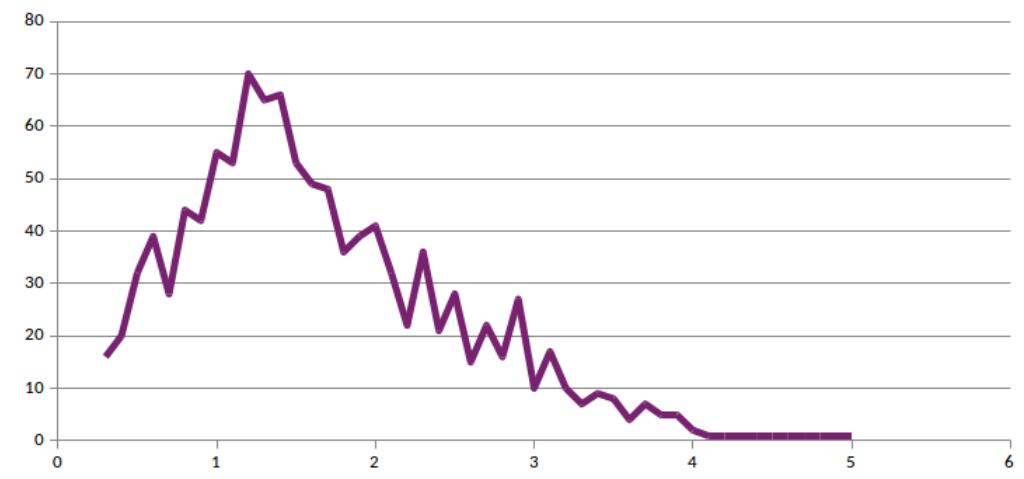


Figura 29: Introdotte le *foreign-table*. Scoperta del non parallelismo delle connessioni.

Lancio di configurazione 3

In questo lancio sono state utilizzate più istanze Postgres, al fine di avere sottoscrizioni sincrone, altrimenti avremo una sola sottoscrizione sincrona e tutte le altre asincrone.

In questo scenario sono mandati più di una copia per chunk usando sottoscrizioni asincrone.

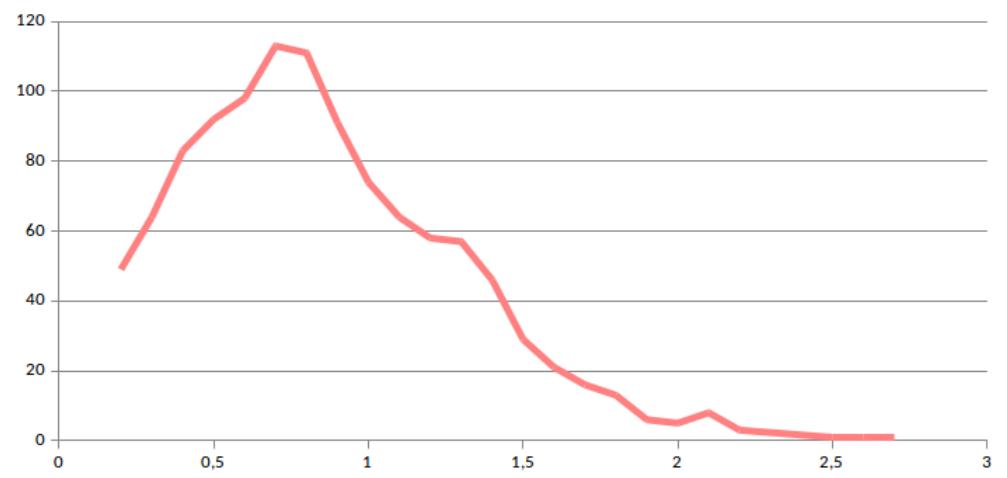


Figura 30: Più copie per chunk con sottoscrizioni asincrone.

Il risultato ottenuto si trova in una posizione centrale rispetto ai due casi limite. Tuttavia la campana formata è ancora troppo larga per poter esser soddisfatti del risultato.

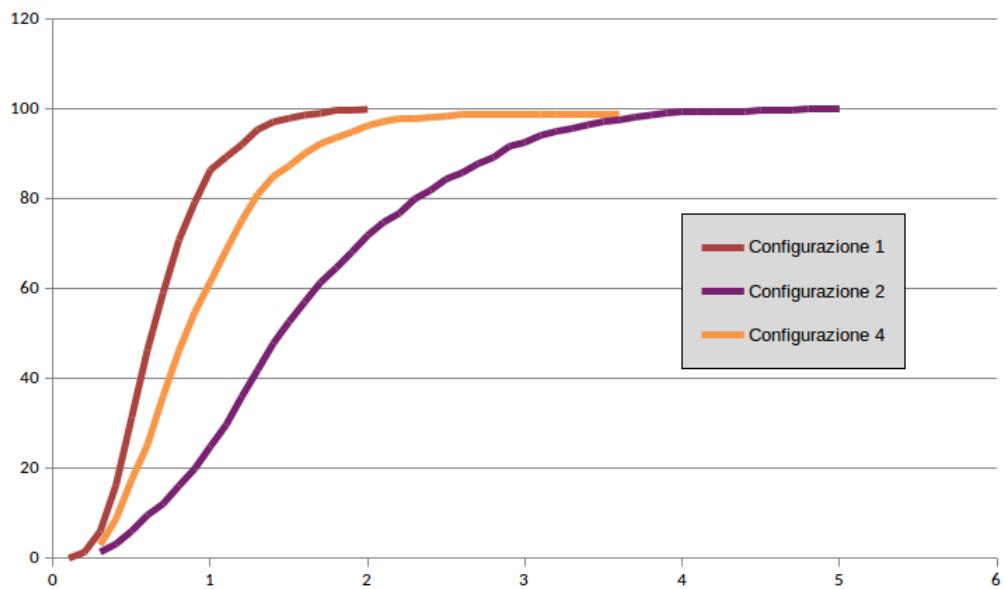


Figura 31: Risultato dei quattro lanci di configurazione

Lancio di configurazione 4

Nella seguente configurazione sono state tolte le *foreign-table*.

Le connessioni sono aperte in serie, mentre le PUT sono in parallelo. Le connessioni non incidono eccessivamente.

Iniziano a rilevarsi miglioramenti, in quanto è presente un primo avvicinamento ai tempi della configurazione 1, ossia quella ottimale.

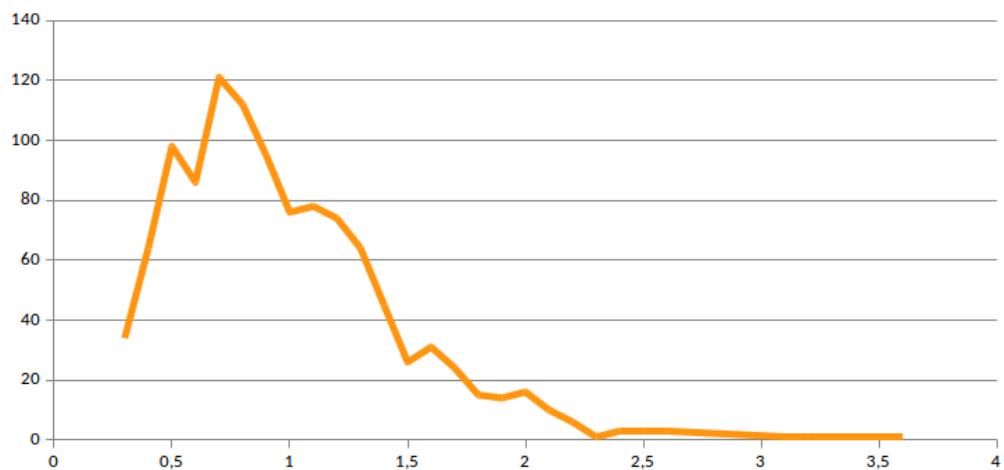


Figura 32: Tolte le *foreign-table*. Connessioni in serie e PUT in parallelo.

I miglioramenti sono visibili dal confronto dei lanci di configurazione.

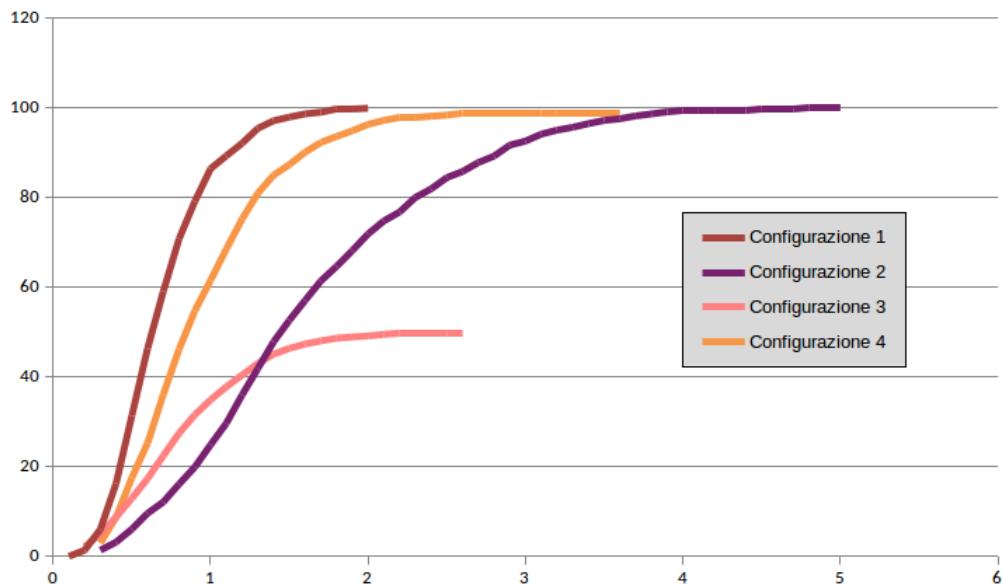


Figura 33: Risultato dei quattro lanci di configurazione

Lancio di configurazione 5

In questo ultimo lancio sono state tolte definitivamente le *foreign-table*. Le PUT sono in parallelo.

In aggiunta sono stati disabilitati i log di Postgres e le connessioni sono aperte in parallelo. Più nello specifico, si aprono alcune connessioni in parallelo che scrivono sul database, senza attendere messaggi di successo su tutti gli altri database.

Si nota che è il risultato migliore ottenuto fino ad ora, anche se non totalmente impeccabile. Questo scenario è un compromesso ragionevole del caso ottimale, che tuttavia non garantisce la ridondanza del dato.

È possibile notare dalla curva che il sistema rallenta sotto 80%, peggiorando solo per un 20% dei casi.

Ciò permette di concludere che la strategia di avere un solo database è vantaggioso per un punto di vista di performance.

Otteniamo in questo modo un'affidabile sicurezza in buoni tempi, arrivando a 126Mbit/sec.

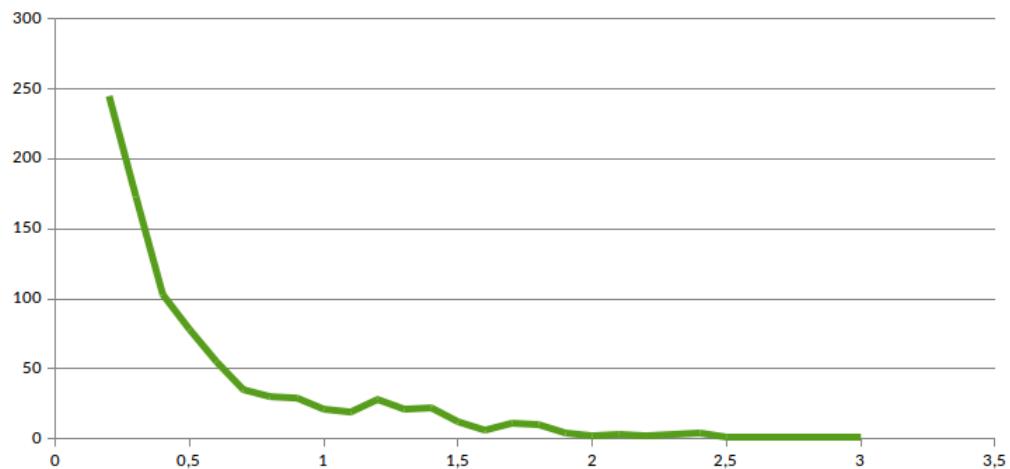


Figura 34: Connessioni in parallelo e PUT in parallelo.)

Il confronto totale ottenuto dai lanci di configurazione appena esaminati, restituisce il seguente grafico:

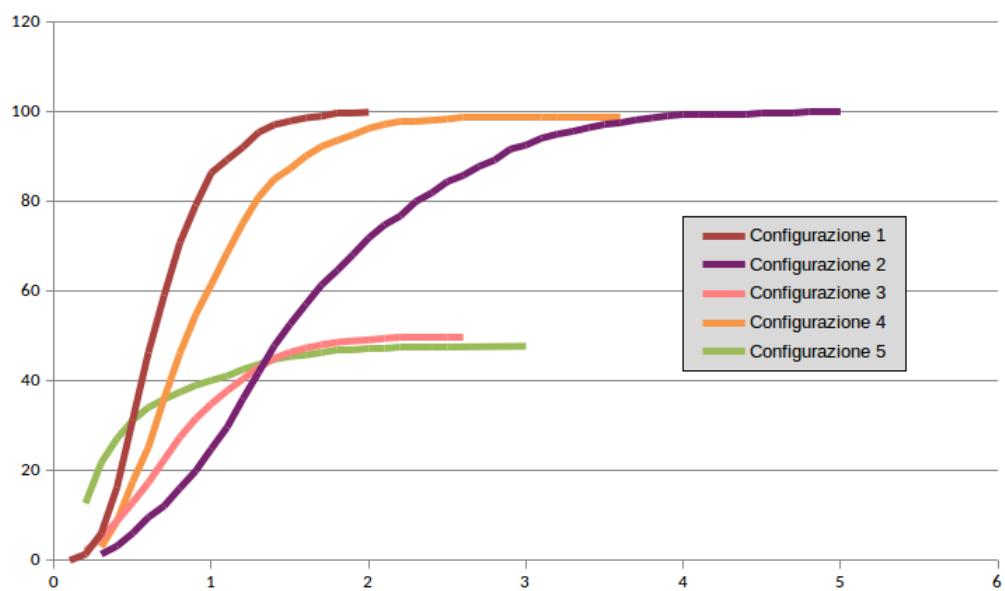


Figura 35: Scenario globale dei lanci di configurazione)

Segue un grafico che rappresenta la frequenza cumulata dello scenario globale, che corrisponde alla somma dei grafici precedentemente ottenuti su base 100. In sostanza è pari alla percentuale.

Più è larga la campana descritta dalle linee, maggiormente la distribuzione è distesa.

Quando la campana è stretta, significa che quel comportamento è indice di un fattore rilevante.

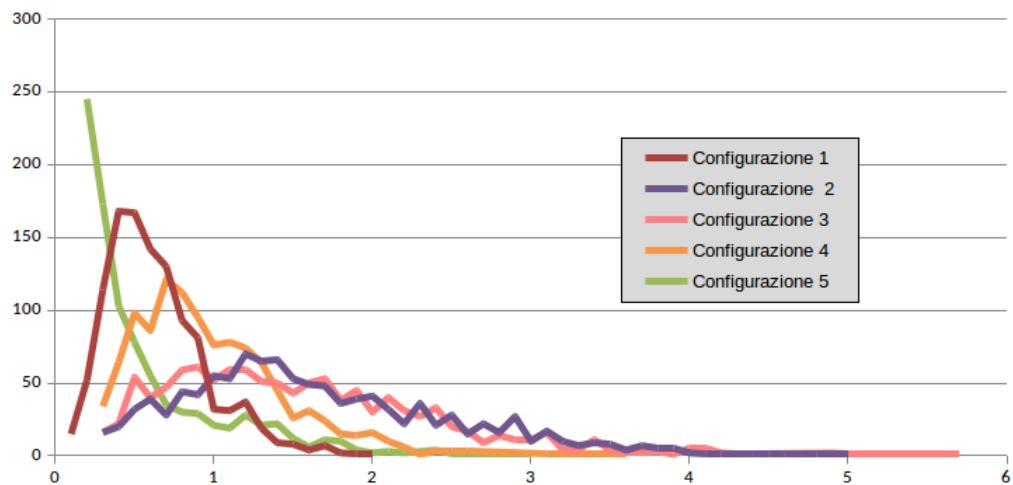


Figura 36: Frequenza cumulata dello scenario globale)

4

CONCLUSIONI E POSSIBILI EVOLUZIONI

4.1 EVOLUZIONI HARDWARE

4.1.1 Utilizzo di processori dual core

I processi coinvolti per la replicazione dei dati e dei metadati sono due:

1. il database Postgres;
2. il *frontend nodejs* che scrive sul database.

Usando un processo a *single-core*, i due processi si contendono il processore e si constringono a svariati *context-switch* di memoria.

Il *context-switch* è fonte di *overhead* non indifferente, generato da:

- tempo impiegato a salvare in memoria lo stato;
- blocco e riattivazione delle *pipeline* di calcolo del processore;
- svuotamento e ripopolamento delle cache.

È di conseguenza consigliabile evitare di effettuare un numero eccessivo di *context-switch*.

Poichè i prezzi si stanno riducendo, una possibile evoluzione è quella di usare un *dual-core*, in modo che ogni processo abbia un *core* dedicato e non siano eseguiti numerosi *context-switch*. Questo tipo di architettura consente di aumentare la potenza di calcolo di una CPU senza aumentare la frequenza di lavoro.

Ciò potrebbe consentire numerosi vantaggi, tra cui miglioramenti per quanto riguardano le prestazioni.

4.1.2 Utilizzo di dischi SSD

Per gli SSD (*solid-state disk*), viste le prestazioni attuali, non sono attesi eccessivi miglioramenti, se non in termini di minori consumi.

Esaminiamo i principali fattori di vantaggio dell'utilizzo di dischi SSD. Quando il costo di *storage*, ossia il costo a GB, sarà lo stesso degli *hard disk*, l'utilizzo degli SSD consentira di diminuire il tasso di guasto.

L'unità a stato solido è di fatto più resistente, dal momento che non contengono parti in movimento (come il motore e il disco magnetico degli HD).

Non ci sarà alcuna perdita del disco, di conseguenza sarà possibile ridurre le ridondanze di dati. L'utilizzo della parità, ad esempio, non sarà più conveniente, tuttavia diverrà fonte di eccessiva occupazione del disco.

Il costo degli SSD incide sul costo di servizio.

I costi di servizio sono quelli che sono pagati dai clienti per avere uno *storage* dei loro dati.

Sarebbe possibile diminuire i costi se ci fosse la possibilità di usare meno *storage* per storare ogni dato degli utenti.

Per mantenere nello *storage* 1 GB, sono occupati in realtà 2,3 GB, in modo da garantire la sicurezza del dato in caso di perdita.

Con l'utilizzo degli SSD sarebbe possibile occupare solo 2 GB per ogni GB del cliente, concendendo un 30% di spazio in meno che però ci permette di diminuire il costo a GB.

L'utilizzo di questo tipo di hardware influisce sui costi di esercizio, ovvero il valore che è spesa dall'azienda anche qualora non ci sia alcun cliente che usufruisce dello spazio, come il consumo elettrico necessario. Il costo di esercizio si può diminuire riducendo i consumi. Se l'uso degli SSD permettesse un calo dei consumi, da un punto di vista di costi, sarebbe un ulteriore vantaggio. Ci sarebbe un garantito taglio dei costi.

Tuttavia attualmente il costo degli SSD è estremamente alto e, per questo, non è conveniente.

4.2 EVOLUZIONI SOFTWARE

In questa tesi è stato esaminato come la replicazione dei dati possa essere gestito da Postgres e dai cluster di database.

Una possibile evoluzione software è di non limitarsi più a offrire un servizio RESTful API basato su *object storage*, ma in qualcosa di più evoluto. Nello specifico, potrebbe essere un software che contenga un linguaggio di query, permettendo di aggregare i dati in un altro modo. Cominciare così a dare degli strumenti di analisi sui dati direttamente sullo *storage* invece di dare soltanto la possibilità di accedere ai dati per caricarli o scaricarli dallo spazio virtuale.

Questo tipo di servizio è sempre più richiesto.

BIBLIOGRAFIA

- [1] Techopedia - *Definition - What does Clustering mean?* (Cited on pages 6 and 7.)
- [2] Techopedia - *Techopedia explains Clustering* (Cited on pages 7 and 9.)
- [3] Wikipedia, the free encyclopedia - *Shared nothing architecture* (Cited on pages 7 and 8.)
- [4] Wikipedia, the free encyclopedia - *Shared disk architecture* (Cited on page 8.)
- [5] Dave Wright - *The Advantages of a Shared Nothing Architecture for Truly Non-Disruptive Upgrades* solidfire.com. 2014-09-17. Retrieved 2015-04-21 (Cited on page 7.)
- [6] SearchNetworking - *peer-to-peer (P2P)* (Cited on page 9.)
- [7] Lifeware - *Introduction to Peer-to-Peer Networks* (Cited on page 10.)
- [8] SearchStorage - *RAID (redundant array of independent disks)* (Cited on pages 10, 11, 12, 13, 14, 15, 16, 17, and 18.)
- [9] SearchStorage - *RAID controller* (Cited on pages 11, 12, 21, and 42.)
- [10] Derek Vadala - *Managing RAID on Linux*, O' Reilly, 2002 (Cited on pages 10 and 14.)
- [11] SearchStorage - *erasure coding* (Cited on pages 18 and 19.)
- [12] PosgreSQL - *Documentation* (Cited on pages 20, 21, 22, 23, and 24.)
- [13] 2ndQuadrant Ltd - *PostgreSQL* (Cited on pages 11, 12, 21, and 42.)
- [14] 2ndQuadrant Ltd - *pglogical* (Cited on pages 24, 25, 26, 27, and 28.)