Allegato Relazione Ing. Scarcello
Analisi delle soluzioni di persistenza No-Sql

Indice

I Analisi principali database NoSQL 6

1 Dynamo DB 7
- 1.1 Data model . . . 7
- 1.2 Query model . . . 8
- 1.3 Shading . . . 8
- 1.4 Replication . . . 10
- 1.5 Consistency . . . 10
- 1.6 Architecture . . . 13
- 1.7 Membership e Failure . . . 14
- 1.8 Implementation . . . 14
- 1.9 Conclusioni . . . 15

2 Google Bigtable/Apache HBase 17
- 2.1 Data Model . . . 17
- 2.2 Query Model . . . 19
- 2.3 Shading . . . 19
- 2.4 Replication . . . 21
- 2.5 Consistency . . . 21
- 2.6 Architecture . . . 22
- 2.7 Membership e Failure . . . 27
- 2.8 Implementation . . . 27
- 2.9 Apache HBase . . . 27
- 2.10 Hypertable . . . 28
- 2.11 Apache Cassandra . . . 29
- 2.12 Conclusioni . . . 29

3 Apache Cassandra 32
- 3.1 Data Model . . . 32
- 3.2 Query Model . . . 36
- 3.3 Sharding . . . 36
- 3.4 Replication . . . 38
- 3.5 Consistency . . . 39
- 3.6 Architecture . . . 42
- 3.7 Membership e Failure . . . 43
- 3.8 Implementation . . . 43

3.9 Conclusioni 43

4 Mongo DB 46
4.1 Data model 46
4.2 Query Model 49
4.3 Sharding 55
4.4 Replication 58
4.5 Consistency 58
4.6 Architecture 61
4.7 Membership e Failure 62
4.8 Implementation 63
4.9 Conclusioni 63

5 BigMemory 66
5.0.1 BigMemory Go & Max 69
5.1 Data Model 70
5.2 Query Model 71
5.3 Sharding 71
5.4 Replication 71
5.5 Consistency 71
5.6 Architecture 72
5.7 Membership e Failure 76
5.8 Implementation 77
5.9 Conclusioni 77

6 Hazelcast 79
6.1 Data Model 79
6.2 Query Model 80
6.3 Sharding 80
6.4 Replication 81
6.5 Consistency 81
6.6 Architecture 82
6.7 Membership e Failure 82
6.8 Implementation 83
6.9 Conclusioni 83

II Analisi principali datastore JCR 85

7 Apache Jackrabbit 85
7.1 Il clustering 85

8 JBoss ModeShape 91
8.1 Caratteristiche chiave 92
8.2 Clustering 93

III Comparazione database NoSQL 98
8.3 MongoDB (2.2) 98
8.4 Redis (2.8) 99
8.5 Riak (1.2) 100

8.6 CouchDB (1.2) 100
8.7 HBase (0.92.0) 101
8.8 Cassandra (1,2) 102
8.9 Hypertable (0.9.6.5) 102
8.10 Accumulo (1.4) 103
8.11 Neo4j (1.5M02) 103
8.12 ElasticSearch (0.20.1) 104
8.13 Couchbase (ex-Membase) (2.0) 105
8.14 VoltDB (2.8.4.1) 105
8.15 Kyoto Tycoon (0.9.56) 106
8.16 Scalaris (0.5) 106

Elenco delle figure

1 Dynamo: Data Model . . . 8
2 Dynamo: Consistent Hashing originale . . . 9
3 Dynamo: Consistent Hashing variante . . . 9
4 Dynamo: Replication over Consistent Hashing . . . 10
5 Dynamo: MultiVersion Concurrency Control . . . 11
6 Dynamo: Vector Clock . . . 11
7 Dynamo: Replication Factor N . . . 12
8 Dynamo: Replication Write and Read Factor . . . 12
9 Dynamo: Architecture . . . 13
10 BigTable: Row/Column DataModel . . . 18
11 BigTable: DataModel . . . 18
12 BigTable: Sharding tramite Tablet . . . 20
13 BigTable: Gerarchia Tablet . . . 20
14 BigTable: Architettura . . . 21
15 BigTable: Architettura . . . 23
16 BigTable: Google File System . . . 23
17 BigTable: Chubby Service . . . 24
18 BigTable: SSTable . . . 24
19 BigTable: Architettura Tablet . . . 25
20 BigTable: Tablet Metadati . . . 26
21 HBase: architettura . . . 28
22 Hypertable: architettura . . . 29
23 Cassandra: Key-Value Storage . . . 33
24 Cassandra: Data Model . . . 33
25 Cassandra: Column Family . . . 34
26 Cassandra: Super Column Family . . . 34
27 Cassandra: Keyspace . . . 35
28 Cassandra: Sharding via Consistent Hash . . . 37
29 Cassandra: Multiple Datacenter Sharding . . . 37
30 Cassandra: Rack Unaware . . . 38
31 Cassandra: Rack Aware . . . 38
32 Cassandra: Datacenter Shard . . . 39
33 Cassandra: Direct and Background Read Requests . . . 41
34 Cassandra: Architecture . . . 42
35 MongoDB: Document Model . . . 46
36 MongoDB: BSON Format . . . 47
37 MongoDB: Collection Model . . . 47
38 MongoDB: Embedded Data Model . . . 48
39 MongoDB: Normalized Data Model . . . 49
40 MongoDB: Query Criteria . . . 50
41 MongoDB: Projection . . . 50
42 MongoDB: Find Method . . . 51
43 MongoDB: Find to SQL . . . 51
44 MongoDB: Aggregation pipeline . . . 51
45 MongoDB: MapReduce . . . 52
46 MongoDB: Indexing . . . 53
47 MongoDB: Indexing . . . 53
48 MongoDB: Single Field Index . . . 54

49 MongoDB: Compound Index 54
50 MongoDB: Compound Index 55
51 MongoDB: Sharding 55
52 MongoDB: Range Based Sharding 56
53 MongoDB: Hash Based Sharding 56
54 MongoDB: Splitting 57
55 MongoDB:Shard Migration 57
56 MongoDB: Replication 58
57 MongoDB: Unacknowledged 59
58 MongoDB: Acknowledged 60
59 MongoDB: Journaled 60
60 MongoDB: Replica Acknowledged 61
61 MongoDB: Architecture 62
62 MongoDB: Primary Set Failure 63
63 Terracotta: Ehcache 67
64 Terracota: BigMemory Max 70
65 Single Server - No persistance of shared data 73
66 Reliable Single Server - Shared Data Persisted 74
67 Available and Reliable - Failover and Persistence 75
68 Available, Reliable and Scalable - Failover, Persistence and Capacity 76
69 ModShape: Architecture 92
70 ModShape: Local Configuration 94
71 ModShape: Clustering Configuration 1 94
72 ModShape: Clustering Configuration 2 95
73 ModShape: Clustering Configuration 3 96
74 ModShape: Clustering Configuration 4 96
75 ModShape: Clustering Configuration 5 97

Parte I

Analisi principali database NoSQL

In commercio esistono decine di database NoSQL ognuna delle quali presenta soluzioni diverse. **Dynamo**, **S3** e **SimpleDB** sono portati avanti da Amazon e sono esempi di database key-value. **BigTable** di Google e il suo derivato open source **HBase** di Apache sono esempi di database a colonne con implementazione di MapReduce, un algoritmo di esecuzione di query in grandi di ottenere risultati anche su enormi mole di dati. Sempre a modello di dati a colonna, **Cassandra** di Apache rappresenta un esemepio di database shared nothing utilizzato da Facebook per lo storage dei messaggi privati. Infine **ChurchDB** e sopratutto **MongoDB** rappresentano esempi di database document-based. MongoDB viene da molti considerato come la soluzione NoSQL che può replicare il successo di MySQL. Sul fronte dei NoSQL basati su un middleware clustering si hanno le due soluzioni principali **BigMemory** e **Hazelcast** che implementano una mappa distribuita come soluzione NoSQL key-value.

Per una migliore visione delle differenze tra i database, ogni soluzione verrà analizzata negli aspetti teorici principali.

Data model come vengono salvati i dati e come viene gestito l'accesso concorrente.

Query model le modalità di ricerca e modifica dei dati memorizzati.

Shading/replication in generale come viene gestita la distribuzione dei dati.

consistency quale livello di consistenza viene garantito.

Architecture le scelte architetturali sul sistema distribuito e sulla merbership.

Failure come vengono gestiti i fallimenti.

1 Dynamo DB

Sviluppato da Amazon[1] per i suoi servizi, Dynamo è uno dei più famosi database NoSQL ed ha influenzato molte altre implementazioni, in particolare per quelli orientati al modello chiave-valore semplice. Lo sviluppo ha seguito un contesto con alcune caratteristiche ben chiare.

- L'infrastuttura è composta da decine di migliaia di server localizzati in una rete di livello mondiale.
- L'esecuzione con parziali fallimenti è una modalità di operazione praticamente standard.
- Un interno stretto livello di agreement (SLA) riguardo prestazioni, affidabilità ed efficienza deve essere rispettato dal 99% della distribuzione.
- Tra le tre componenti, l'affidabilità ha priorità assoluta perché anche la minima interruzione ha notevoli conseguenze finanziarie e impatta anche la fiducia del cliente.
- Per supportare una crescita continua, il sistema deve essere altamente scalabile sul numero di server.

Dynamo è utilizzato per gestire lo stato dei servizi che hanno requisiti molto elevati di affidabilità e bisogno di uno stretto controllo sui compromessi tra disponibilità, coerenza, economicità e prestazioni.

1.1 Data model

Si è notato come molti servizi (come la gestione dei prodotti di un catalogo, il carrello di un utente, la gestione delle sessioni) hanno bisogno solo della chiave per essere utilizzati e gran parte della potenza semantica dei database relazionali non è utilizzata. Per tale ragione Dynamo, nonostante tutti i vantaggi delle propietà ACID, preferisce utilizzare un modello dati del tipo chiave-valore semplice i cui i valori sono memorizzati in forma di oggetti binari (BLOB). Le operazioni sono limitate a un singolo oggetto senza eventuali referenze e più in generale operazioni che operano su un insieme di oggetti. Anche spazio gerarchico come un filesystem distribuito non è supportato in Dynamo.

[1] Dynamo: Amazonś Highly Available Key-value Store

Figura 1: Dynamo: Data Model

{ Id = 101 ProductName = "Book 101 Title" ISBN = "111-1111111111" Authors = ["Author 1","Author 2"] Price = -2 Dimensions = "8.5 x 11.0 x 0.5" PageCount = 500 InPublication = 1 ProductCategory = "Book" }	{ Id = 202 ProductName = "21-Bicycle 202" Description = "202 description" BicycleType = "Road" Brand = "Brand-Company A" Price = 200 Gender = "M" Color = ["Green", "Black"] ProductCategory = "Bike" }
{ Id = 201 ProductName = "18-Bicycle 201" Description = "201 description" BicycleType = "Road" Brand = "Brand-Company A" Price = 100 Gender = "M" Color = ["Red", "Black"] ProductCategory = "Bike" }	

Il miglior beneficio di questo modello è la possibilità di rendere le operazioni di lettura e scrittura un singolo dato il più performante possibile. E dynamo riesce a offrire prestazioni notevoli nei suoi servizi.

1.2 Query model

L'interfaccia che Dynamo offre ai client segue la visione semplice del modello di dati. Sono sostanzialmente due le possibili operazioni.

get(key) ritorna l'oggetto indicizzato dalla chiave passata come parametro oppure, in presenza di conflitti) a lista di oggetti memorizzati con la stessa chiave ma aventi versioni diverse. Inoltre la viene ritornato uno speciale oggetto che identifica il contesto, ovvero un insieme di informazioni di sistema sull'ambiente in l'oggetto è memorizzato.

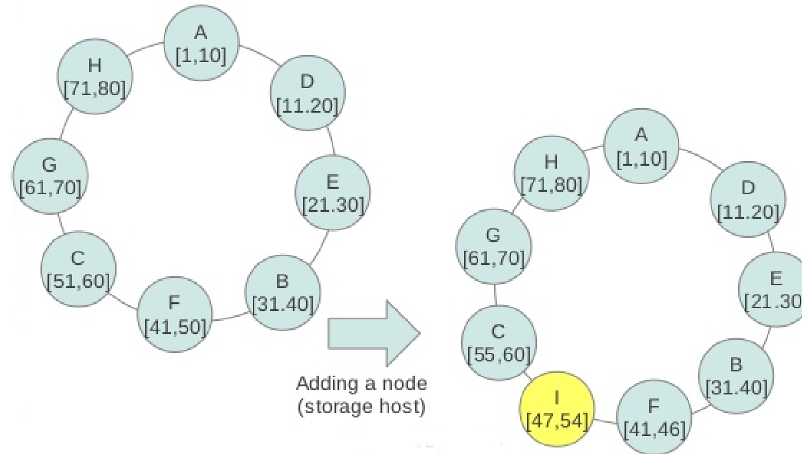
put(key, context, object) salva un oggetto sul database con una chiave associata. Il caso di aggiornamento, ilcontesto deve essere lo stesso di quello ricevuto da una operazione di get e serve a Dynamo èper gestire il controllo di versione.

Dynamo tratta gli oggetti come un "*array opaco di byte*", lasciando totalmente il compito all'applicazione sovrastante il conferirne semantica.

1.3 Shading

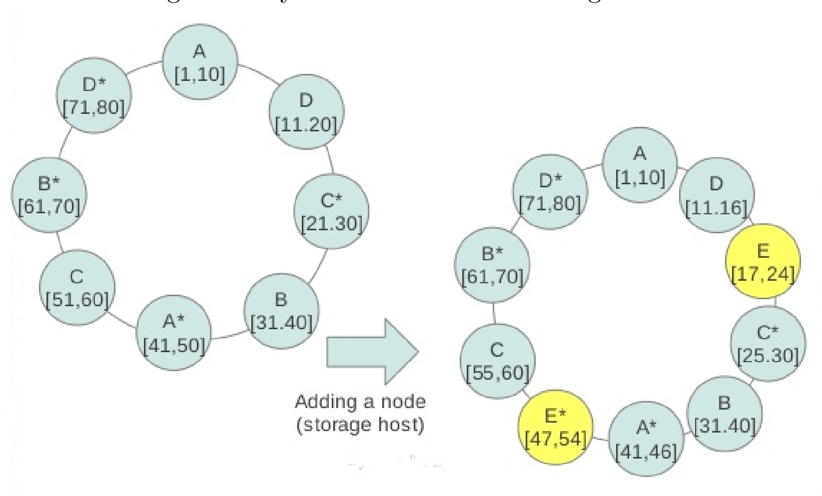
Per garantire elevata scalabilità, Dynamo utilizza una variante del consistent hasing come tecnica di partizionamento. La tecnica base non permette un buon bilanciamento del carico a causa della struttura ad anello, inoltre non permette di gestire server con potenza di calcolo diversa.

Figura 2: Dynamo: Consistent Hashing originale



La variante proposta da Amazon permette al suo database di mantenere un miglior equilibrio nel carico di dati tramite un partizionamento dinamico che avviene all'aggiunta di nodi virtuali.

Figura 3: Dynamo: Consistent Hashing variante

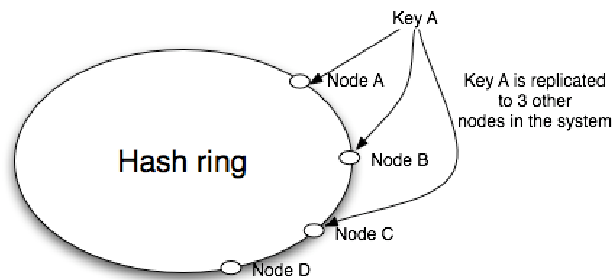


Dynamo applica il concetto di **virtual host**: ogni nodo è suddiviso in un numero di nodi virtuali che vengono associati in maniera casuale in una posizione all'interno dello spazio dei possibili valori di delle chiavi (**key-space**). Ogni nodo è responsabile di tutte le chiavi dalla propria posizione fino a quella del predecessore. In base alla potenza hardware di un nodo fisico è possibile configurare sopra di esso un numero più o meno alto di nodi virtuali al fine di ottenere un miglior uso delle risorse fisiche.

1.4 Replication

Poiché Dynamo nasce per lavorare in un contesto dove la presenza di fallimenti fisici delle macchine è "*standard*", la gestione della disponibilità e dell'affidabilità è garantita in maniera robusta attraverso la replicazione implementata sopra lo schema del consistent hashing. Ogni oggetto è replicato un numero configurabile di volte (tipicamente N=3). Ogni nodo è responsabile della porzione di valori delle chiavi dalla propria posizione fino a quella del predecessore

Figura 4: Dynamo: Replication over Consistent Hashing

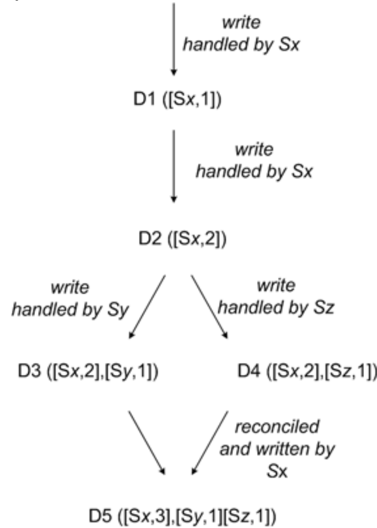


Ogni volta che un nodo deve memorizzare un oggetto ha anche il compito di inoltrare un messaggio di replicazione agli N-1 successori in senso orario. In altre parole, in presenza di replicaizone, ogni nodo memorizza tutte le chiave dalla sua posizione a quella dei suoi N predecessori. Ogni nodo ha conoscenza dell'insieme dei nodi che gestiscono una determinata chiave su di esso generano una lista di prioritaria detta **preference list**.

1.5 Consistency

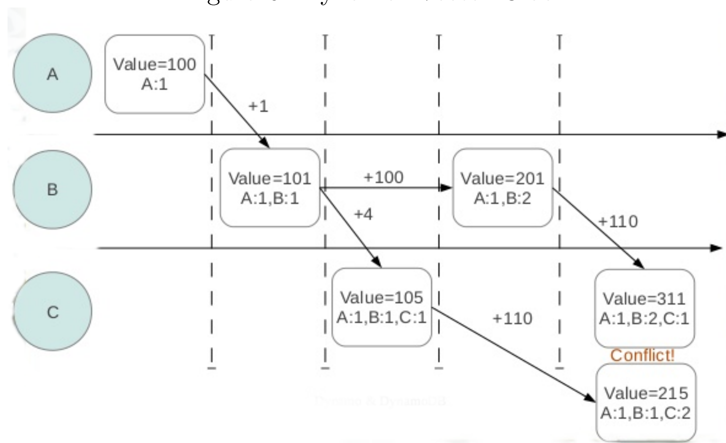
Dynamo è stato progettato per garantire una consistenza eventuale di tipo ottimistico in cui le operazioni di scrittura vengono completate immediatamente senza aspettare che tutti i nodi designati come replica abbiano ricevuto e completato l'aggiornamento. Letture sequenziali potrebbero ritornare valori differenti se effettuate su nodi replica diversi e per questo Dynamo crea sempre una nuova immutabile versione di un oggetto ad ogni scrittura (MVCC).

Figura 5: Dynamo: MultiVersion Concurrency Control



Poiché il database ha una visione "grezza" dei dati è in grado di fare una riconciliazione puramente sintattica delle diverse versioni di un determinato oggetto attraverso l'uso dei vector clock. Quando un client esegue una operazione di put di un oggetto con una determinata chiave, deve inserire il contesto (che ha al suo interno i vector clock) dell'ultima operazione di get effettuata sulla stessa chiave.

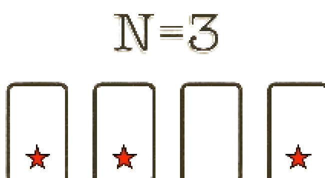
Figura 6: Dynamo: Vector Clock



Quando un nodo ricevere una richiesta su una determinata chiave, questa può essere inoltrata utilizzando la preference list, la lista prioritaria di nodi che hanno la responsabilità su una determinata chiave. Ad ogni richiesta vengono

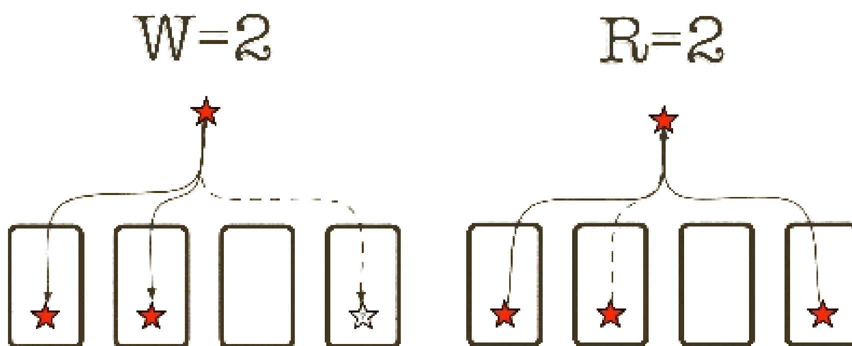
considerati un numero configurabile N di nodi attivi ignorando quelli che sono in fallimento.

Figura 7: Dynamo: Replication Factor N



Per garantire una visione consistenza del risultato si utilizza un protocollo basato su quello che viene detto **sloppy quorum**. In particolare, durante una scrittura, il nodo, detto coordinatore, crea in locale un nuovo vector clock per la nuova versione ed in seguito effettua un inoltro ai nodi responsabili della chiave. L'operazione viene considerata completata se una soglia $W - 1$ di nodi hanno terminato con successo l'aggiornamento. Nelle lettura è il nodo stesso che riceve la richiesta ad essere coordinatore e invierà direttamente messaggi a N nodi attivi presenti nella lista preferenze e si metterà in attesa di una soglia $R - 1$ di risposta.

Figura 8: Dynamo: Replication Write and Read Factor



In presenza di valori diversi (osservabili attraverso i vector clock) verrano tutti ritornati al client sotto forma di lista. di oggetti. Il client, invece, ha una conoscenza completa sulla struttura degli oggetti e sul il contesto reale in sui si utilizzano. Per cui è compito dell'applicazione che utilizza Dynamo occuparsi della consistenza e gestire esplicitamente il conclitto attraverso una riconciliazione semantica. Per tale ragione l'operazione di get può tornare una lista di oggetti, mentre quella di put riceve sempre un solo valore da poter scrivere.

Per garantire una buona probabilità di risposta consistente, la somma dei valori di soglia R e W dovrebbe superare il numero di nodi contattati dalla preference list.

$$R + W > N \tag{1}$$

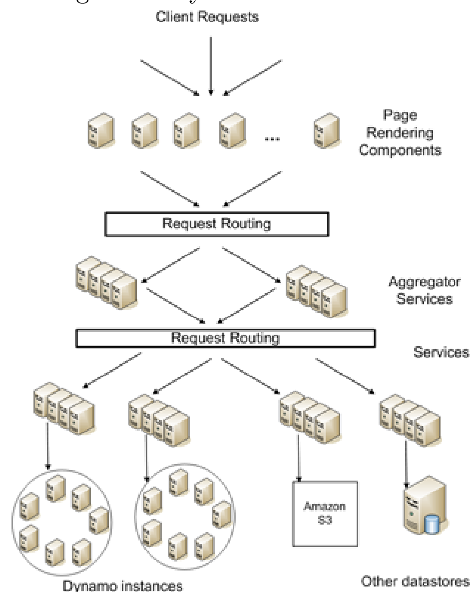
In genere, al fine di garantire tempi di risposta bassi nelle letture solitamente si attende la read di uno, massimo due nodi attivi ($R = 1$ o $R = 2$), mentre sulle write si preferisce garantire una soglia pari a tutti i nodi attivi contattati ($W = N$). Nei casi in cui questa proprietà ($R + W > N$) non viene rispettata si aumenta la probabilità di inconsistenza a causa del fatto che le write non attendono un numero sufficentemente elevato di repliche con rischi anche in termini di affidabilità e durabilità.

In generale Dynamo affronta il problema del Teorema CAP lasciando al client la possibilità di decidere i livelli di consistenza, disponibilità, affrdabilità e prestazioni desiderate configurando i valori di R, W ed N.

1.6 Architecture

Dynamo offre un'architettura **shared nothing**, ovvero senza una master o uno strato di file system condiviso. La topologia della rete è ad anello in cui ogni nodo può ricevere una richieste su una qualunque chiave per poi inoltrarla al nodo opportuno. In questo modo è possibile evitare colli di bottiglia o single point of failure. Per individuare un nodo sul quale fare una richiesta il client può contattare un load balancer atto ad inoltrare le richieste dentro l'anello in maniera da garantire un bilanciamento del carico, oppure utilizzare una libreria ad-hoc che riflette il partizionamento e può determinare il nodo da contattare attraverso dei calcoli in locale al client.

Figura 9: Dynamo: Architecture



1.7 Membership e Failure

Dynamo offre un meccanismo esplicito per l'aggiunta rimozione attraverso una command-line o da un'interfaccia browser che genera un messaggio inviato ad un qualunque nodo della rete che si occuperà, attraverso un protocollo GOSSIP, di notificare la variazione topologica a tutti gli altri nodi. Il sistema provvederà poi a ricalcolare il key-space in baso all'aggiunta o rimozione dei nodi virtuali associati alla macchina fisica.

Lo **sloppy quorum** offre un a modalità di disponibilità del sistema anche in presenza di fallimenti. Le operazioni read e write in arrivo su un nodo vengono inoltrare su N nodi attivi che non sono necessariamente i primi N nodi secondo l'ordine del key-space.

Se un nodo va in crash durante la write di un oggetto entra in gioco una seconda misura di risposta ai fallimenti temporanei nota come **hinted handoff**: il nodo coordinatore replica la richiesta di write fallita su un altro nodo solitamente non responsabile della chiave associate in modo da garantire comunque la presenza di almeno N repliche. Quando il nodo fallito tornerà attivo, sarà il nodo che ha ricevuto la seconda richiesta ad inoltrarla al vero destinatario cancellando il valore dal database locale non appena ne riceve il riscontro.

Esistono alcuni scenari in cui sono possibili fallimenti permanenti non più ercuperabili attraverso le tecniche normali. Ad esempio un nodo potrebbe fallire prima che riesce ad inoltrare alle repliche. Per gestire queste d altre situazioni che potrebbero inficiare la durabilità, Dynamo ha implementato un protocollo anti-entropia per la sincronizzazione delle repliche. La tecnica si basa sugli alberi **Merkle** per individuare inconsistenze tra le repliche e determinare i dati da sincronizzare. In particolare un Merkle è un albero dove le foglie sono i calori delle singole chiavi e i nodi di livello più alto sono risultato di una funzione hash applicata ai rispettivi figli. In particolare ogni nodo mantiene un albero Merkle per ogni key-range di cui è responsabile. Un semplice controllo sui valori hash dei rispettivi alberi permette di determinare differenze tra due repliche in maniera estremamente veloce.

1.8 Implementation

Ogni componente di Dynamo, sia esso per la gestione delle richieste, del fallimento o della merbership è stato scritto in Java. Il supporto alla persistenza è garantito dall'integrazione con i più comuni database in commercio come MySQL o Berkeley Database Transactional DataStore.

Il componente responsabile di richieste di coordinamento è implementato sopra un livello di messaggistica basato su eventi in cui la pipeline di elaborazione messaggi viene suddiviso in più fasi. La comunicazione di basso livello avviene tramite il package standard Java NIO.

Quando un nodo qualunque riceve una richiesta ne diventa il coordinatore inoltrando la richiesta ai nodi responsabili ed aspettando la risposta nella logica a quorum. Per gestire questa situazione crea, per ogni richiesta, una struttura di

macchina a stati finiti. Per ottimizzare i tempi ed alleggerire il carico sui server Dynamo offre la possibilità all'applicazione client di poter essa stessa creare, mantenere e gestire localmente la macchina a stati finita della richiesta. In questo modo è il client stesso a diventare coordinatore dei nodi nella preference list.

I nodi di Dynamo supportano anche una gestione in-memory. Le richieste write, che normalmente sono lente, sono sempre scritte nella RAM volatile e saranno rese persistenti da un processo parallelo che in maniera asincrona salva la cache su disco. In fase di lettura sia la memoria che il disco sono usati. Per ridurre il rischio di perdere dati no salvati su disco il coordinatore designa una delle macchine ad eseguire una write direttamente su disco. La latenza non è inficiata dalla scrittura su disco perché il coordinatore attende sempre W-1 riscontri.

Tra i componenti implentati, Dynamo ne offre uno per il monitoraggio della rete sotto tutti gli aspetti.

1.9 Conclusioni

In generale Dynamo rappresenta un ottimo esempio di come un sistema di storage con il modello di dati più semplice possibile possa essere adattato per un contesto estremo come quello dei servizi di Amazon. Le tecniche sviluppate sono state da esempio per numeroi database e servizi successivi.

Problema	Tecnica	Vantaggi
Partizionamento	• Consistent Hashing • Virtual Host	Scalabilità orizzontale
Consistenza e Isolamento	• Vector Clock • Read reconciliation	La dimensione delle versioni non riduce il rate di scrittura
Gestione fallimenti temporanei	• Sloppy Quorum • Hinted Handoff	Elevata garanzia di disponibilità e durabilità in presenza di temporanei fallimenti delle repliche
Gestione fallimenti permanenti	• Merkle-Tree • Anti-entropy protocol	Sincronizzazione delle repliche in background

Membership	GOSSIP protocol	Evita la presenza di un registro centralizzato

In una presentazione Ippolito [2] riassunse i vantaggi di Dynamo.

Vantaggi:

- Assenza di colli di bottiglia su un nodo master.
- Ottimo bilanciamento carico e gestione risorse hardware.
- Elevate prestazioni e disponibilità nelle write.
- Espicità configurabilità del rapporto consistenza/prestazioni.
- Semplicità nelle API esposte.

Svantaggi:

- Proprietario di Amazon.
- Applicazione client intelligente.
- Nessuna compressione.
- Nessun supporto a interrogazioni basate su colonne.
- Limiti del modello Chiave-Valore.

[2] Ippolito, Bob (2009) - Drop ACID and think about data

2 Google Bigtable/Apache HBase

Ideato da Google, BigTable rappresenta una delle implementazioni più famose di database ColumnFamily. Costruito sopra il file system distribuito GFS è usato in molti servizi con diverse esigenze: alcuni richiedono basse latenze per garantire all'utente risposta in tempo reale ed altre più orientate all'elevato volume di produzione.

BigTable resta proprietario dell'azienda, ma la sua tecnolgia ha ispirato varie implementazioni open source tra le quali la più famose sono Hypertable e soprattutto HBase, di Apache che si basa sul file system Hadoop.

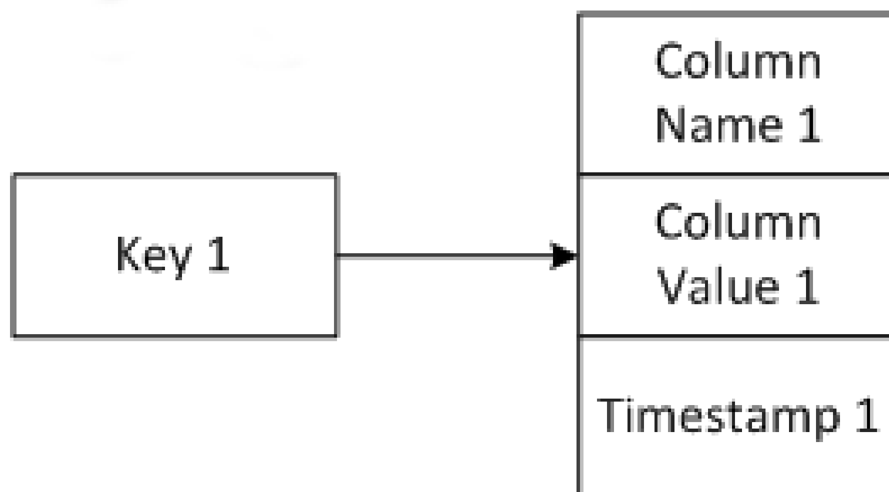
Parte delle idee hanno ispirato anche Cassandra, altro database basato su famiglie di colonne che viene spesso visto come una via di mezzo tra BigTable e Dynamo.

2.1 Data Model

Il modello di dati di BitTable può essere visto come una evoluzione del più semplice key-value di Amazon Dynamo. La visione di dati a coppie chiave-valore come mattone per lo sviluppo di applicazioni è troppo semplicistica. L'idea è di offrire un modello più ricco che supportasse un concetto di informazione semi-strutturata, ma rimanendo talmente semplice da poterne avere una rappresentazione su file e garantendo una trasparenza tale da poter offrire una facile configurabilità degli aspetti importanti del sistema.

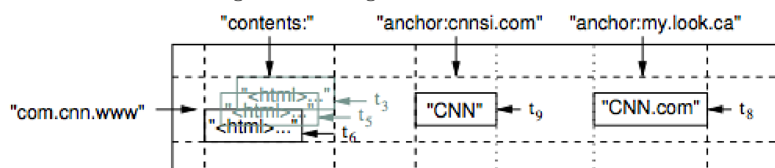
Formalmente la struttura dei dati fornita da BigTable è descritta come una "*tabella sparsa, distribuita, persistente, multidimensionale e ordinata*" in cui i valori sono memorizzati come array di byte non interpretati e indicizzati per via della tripletta chiave di riga (**row-key**), chiave di colonna (**column-key**) e un marcatore temporale (**timestamp**).

Figura 10: BigTable: Row/Column DataModel



Le chiavi di riga in BigTable hanno una dimensiona massima di 64KB, sono mantenute ordinate e sono usate per la strategia di partizionamento su diverse macchine. Le column-key hanno una sintassi del tipo "**family:qualifier**". Il prefisso è obbligatorio ed è noto come **ColumnFamiliy** (**CF**), che permette di raggruppare colonne in sotto liste. Questo concetto è il vero cuore del modello di dati che conferisce a BigTable molte potenzialità. In figura si mostra un esempio di modellazione a colunne.

Figura 11: BigTable: DataModel



Ogni famiglia permette di indicizzare dati di uno stesso tipo o comunque molto simile. Basando il controllo di accesso sulle column family, è possibile avere colonne con diversi privilegi per diverse applicazioni che accedono alla stessa riga. BigTable è stato progettato affinché ogni riga possa avere un numero teoricamente illimitato di column family rendendo il database flessibile anche in assenza di uno schema. È consigliano di mantenere basso e stabile nel tempo il numero di famiglie per ogni tabella. Questo perché si vuole assumere che l'applicazione conosca le possibili famiglia di una riga e che sia sempre specificata prima di poterne effettuare una scrittura. Nonostante il modello dei dati sia flessibile e garantisca facile aggiornamenti al cambiare della struttura dei dati è importante manentenere costante la visione dei dati dell'applicazione con quella del database.

I timestamp invece sono degli interi da 64bit che usati da BigTable per identificare differenti versioni di uno stato dato. Il valore può essere generato automaticamente oppure fornito da client stesso, ma deve essere mantenuta la proprietà di univocità.

2.2 Query Model

BigTable fornisce una libreria scritta in C++ per interrogare un database che espongono un interfaccia "*scanner* "che permette di definire filtri sulle chiavi, colonne e timestamp e di poter iterare i risultati.

Le operazioni fornite dalla API BigTable sono:

Read selezione delle righe con filtri su colonne e timestamp in un ottica simile alle proiezioni del modello relazionale. È possibile iterare suia su righe che su colonne, ma l'ordinamento è garantito solo sulle prime.

Write for rows creazione, modifica e cancellazione dei valori delle colonne di una particolare riga. È possibile eseguire operazioni scritture in blocco su più righe.

Write for table and column family crewzione o cancellazione di tabelle o famiglie di colonne.

Administrative operazioni di modifica di meta informazioni come i diritti di accesso.

Server-Side code esecuzione di codice scritto in Sawzall (linguaggio di Google orientato all'utilizzo di dati) per operazioni quali filtri basati su espressioni arbitrarie, aggregazioni su complessi algoritmi e altre trasformazioni di alto livello. Operazioni di modifica dei dati non sono permesse.

MapReduce supporto al framework MapReduce.

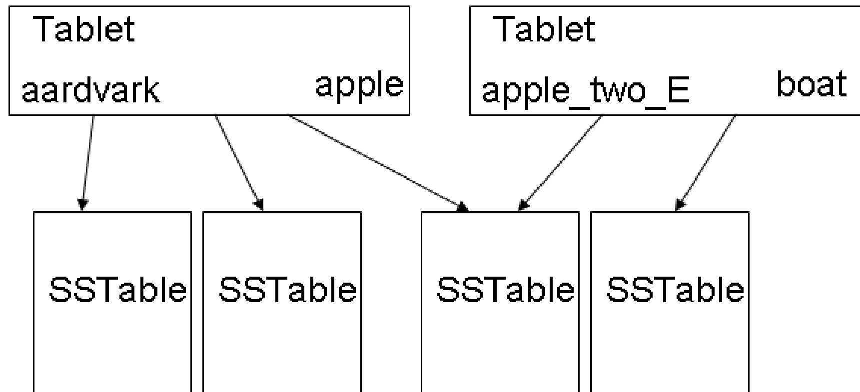
In generale ogni query restringe l'insieme di righe sulle quale lo scanner deve andare a cercare la locazione fisica. È anche possibile eseguire ricerche su intervalli di chiave basate su prefissi. In ogni caso i risultati sono sempre ordinati per chiave, mentre non è possibile avere garantito un ordine per colonne.

BigTable fornisce anche un integrazione per il framework MapReduce potendo definire le funzioni di map e reduce.

2.3 Shading

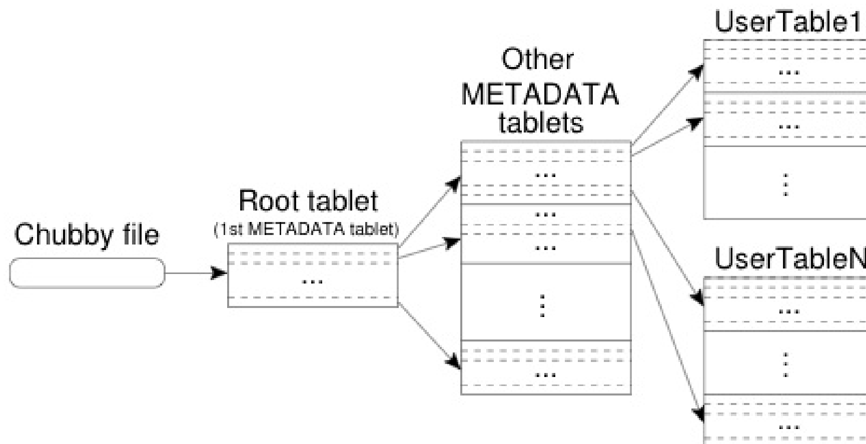
In BigTable il partizionamento è basato sul concetto di **tablet**: il blocco unitario utile per la distribuzione ed il bilanciamento del carico. Una tabella viene quindi divisa in vari blocchi di righe che vengono distribuiti su diversi nodi, detti **tablet server**.

Figura 12: BigTable: Sharding tramite Tablet



Ogni tablet è assegnato ad un solo server in maniera dinamica e i server che possono entrare ed uscire dalla rete senza blocchi dell'esecuzione di BigTable. È, quindi, compito del framework stesso gestire l'assegnamento e la localizzazione dei tablet in modo da dare la possibilità alla libreria client di poter individuare quale server contattare per un determinato blocco di chiavi. Infatti, tutte le fasi del ciclo di vita di un tablet, creazione, cancellazione o assegnamento sono gestite da un nodo master. Un tablet può avere al massimo un server che lo gestisce e possono esserci lassi di tempo in cui non è assegnato a nessuno nodo e di conseguenza non è raggiungibile dal client. In BigTable il partizionamento è dinamico in quanto i tablet possono essere accorpati o divisi.

Figura 13: BigTable: Gerarchia Tablet



La localizzazione dei tablet ha una struttura gerarchia a tre livelli. Il primo livello consiste in un file con le informazioni sulla **root metadata**, una tabella mantenuta

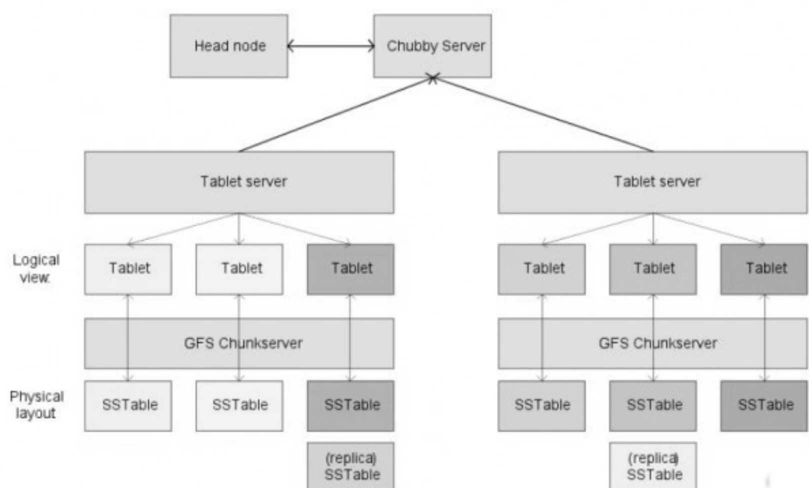
in-memory a partire dalla quale è possibile raggiungere un numero arbitrario di altre tabelle metadata a partire dalle quali si hanno le effettive informazioni per l'identificazione delle cosiddette **user table**, tabelle create dalle applicazioni. Poiché le chiavi sono mentenute in ordine ascendete, la coppia tabella e la sua ultima ultima chiave è sufficiente per identificare un tablet.

2.4 Replication

BIgTable non gestisce direttamente la replicazione perché impone che ogni tablet sia assegnato ad un solo server, ma utilizza il file system distribuito GFS che garantisce una replicazione dei tablet sotto forma di file noti come **SSTable**.

Per migliorare le prestazioni delle read è possibile definire un doppio livello di cache a livellod i tablet server. Una cache di tipo chiave-valore ritornata da un oggetto offerto dalla API SSTable noto come **scan cache** che permette di velocizzare l'accesso ai dati letti più frequentemente. Una seconda cache, la **block cache**, è utilizzata per quelle applicazioni che tendono a leggere dati vicini a quelli che leggono di recente.

Figura 14: BigTable: Architettura



2.5 Consistency

Con una strategia di partizionamento basa su blocchi di chiave garantisce che tutti gli attributi duna singola riga sono sempre presenti sulla stessa macchina. Questo permette a BigTable di fornire una consistenza forte a discapito della disponibilità in presenza di fallimenti su un server.

Le operazioni su una singola riga sono atomiche ed è possibile garantire anche blocchidi operazioni transazioni su di esse. Al contrario nessuna forma di relazione è garantita, per cui transazioni su più righe devono essere gestite lato applicazione client.

2.6 Architecture

BigTable utilizza una tipo di architettura orientata a un master che ha il compito di assegnare i tablet ai singoli server che lavorano sopra uno strato di memorizzazione distribuita su file abbinato a un servizio di lock basato su consenso.

I componenti principali in gioco in Bigtable sono.

Google File System lo strato di basso livello per gestire la persistenza delle informazioni e dei log commit in un formato di file. Ha anche il compito di gestire replicazione e affidabilità in maniera trasparente al database.

Memtable/SSTable Le informazioni sono la fusione da i dati più recenti tenuti in-memory e quelli più vecchi salvati su un formato file immutabile e reso persistente dal file dal GFS.

Tablet Server gestisce i dati partizionati in blocchi di chiavi ordinati, detti tablet.

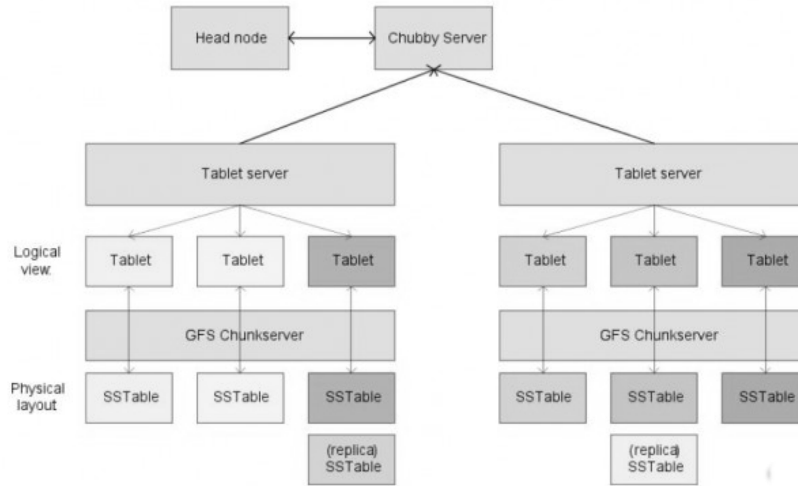
Master Server gestisce tablet e il loro assegnamento ai server con bilanciamento del carico. Gestisce cambianti dello schema con creazione di tabelle e famiglie di colonne. Ha un moduile di gargage collector per la gestione dei file obsoleti presenti nel GFS. Ha sempre una seconda macchina pronta a sostiure il master.

Chubby Service servizio di lock distribuito ad elevata disponibilità. Assicura la presenza di un unico master nella sistema con un algoritmo del consenso. Memorizza le informazioni per la fase di avvio di BigTable. Gestisce la membership scoprendo la presenza di nuovi tablet server ed indentificando il fallimento di quelli presenti. Memorizza le informazioni di schema e la gestione delle restrizioni d'accesso sia per le tabelle che per le column family.

Client library libreria fornita per far interagire l'applicazione con BigTable. Ha il compito di individuare quali tablet sono responsabili delle chiavi richieste dal client ed inoltrare direttamente su essi le richieste.

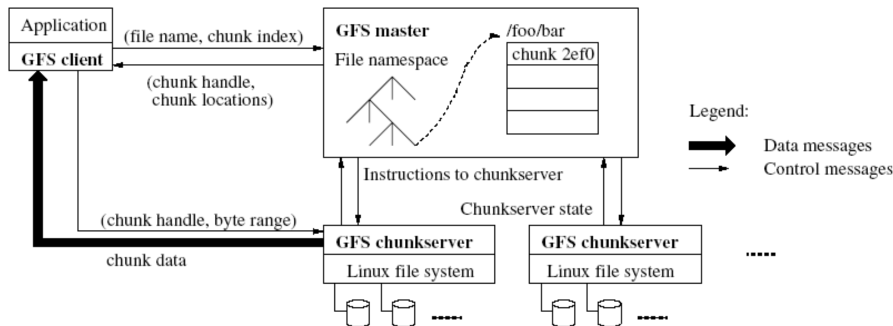
La figura in seguito ne mostra la gerarchia.

Figura 15: BigTable: Architettura



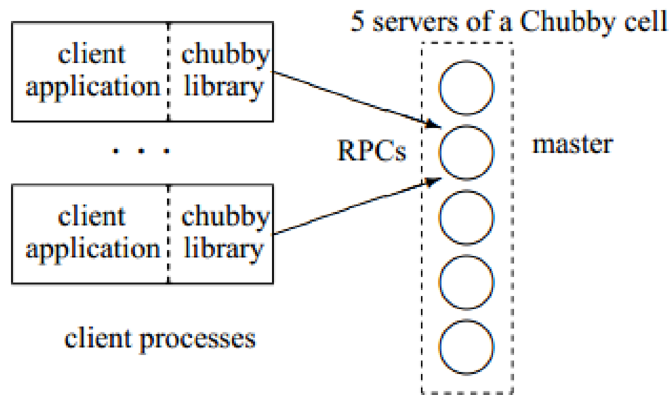
Lo strato più basso usato da BigTable per memorizzare in maniera persistente sia dei log e sia dei dati è il **Google File System** (**GFS**).

Figura 16: BigTable: Google File System



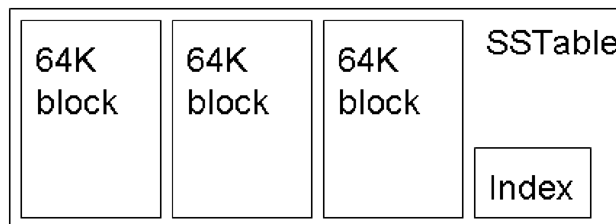
In abbinamento a esso è presente un servizio di lock distribuito, il **Chubby Service**, un cluster di cinque macchine replica che, attraverso un algoritmo di elezione, designano una di essere come unica macchina primaria che risponde alle richieste degli altri componenti del sistema. Per garantire una consistenza forte il cluster utilizza l'algoritmo di consenso Paxos. Il servizio espone un'interfaccia a file-system semplificato per gestire directory e file di piccole dimensioni su determinati namespace. Le applicazioni client, incluse le istanze BigTable, possono aprire una sessione, chiuderla o aggiornarla per mantenerla viva. Il servizio di Chubby è essenziale per l'esecuzione di BigTable e una sua caduta è un punto di fallimento del sistema per cui la presenza di cinque macchine è considerato un requisito essenziale.

Figura 17: BigTable: Chubby Service



Le operazioni di write vengono sia scritte immediatamente sul GFS tramite un commit log per renderle rieseguibili e contemporaneamente vengono rese visibili tramite uno spazio in-memory gestito attraverso un componente software noto detto **memtable**. Raggiunta la sua sua massima dimensione, la memtable viene congelata e tutti suoi dati sono salvati sul file system distribuito nel formato **SSTable**, una tabella persistente immutabile e ordinata dove e le chiavi e i valori sono sequenze non strutturate di byte. Internamente un SSTable è composto da una sequenza di blocchi di dimensione prefissata in (tipicamente 64k) fase di configurazione.

Figura 18: BigTable: SSTable

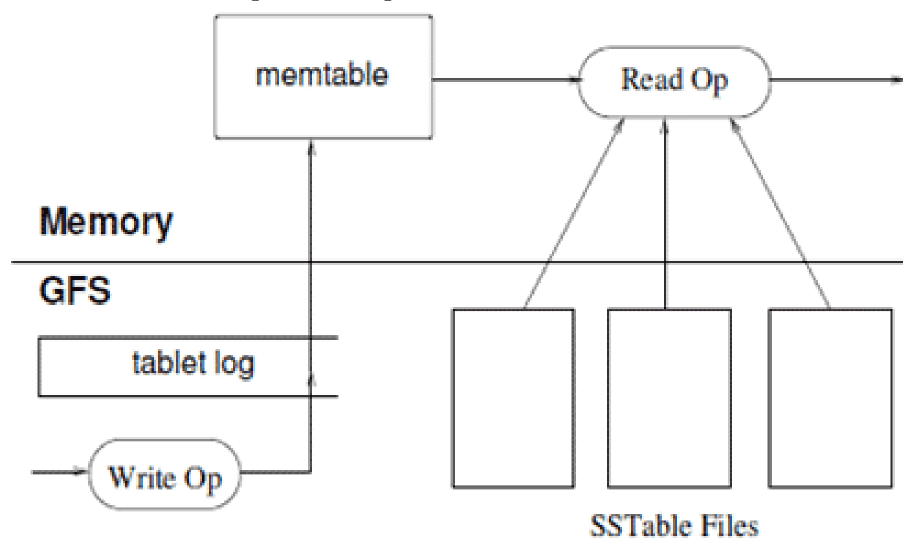


Quando il file viene aperto, gli indici di questi blocchi sono caricati in memoria e su di essi si esegue la ricerca binaria, garantita dall'ordine delle chiavi, in modo da eseguire l'accesso al disco direttamente su blocco che contiene una determinata chiave. Il processo di salvataggio su una SSTable è detto **minor compaction** durante il quale una nuova memtable viene creata per i successivi aggiornamenti. All'aumentare delle operazioni di write il numero di memtable congelate aumenta e di conseguenza anche il numero di SSTable rischiando di degradare le prestazioni. Per tale ragione, in maniera asincrona, BigTable esegue quella che è nota come **merge compaction**, una fusione tra SSTable attraverso la tecnica Copy-on-Write. Un caso speciale di fusione è la **major compaction**: la produzione di una sola

SSTable. Questa tecnica viene eseguita regolarmente da BigTable per gestire i dati cancellati e liberando risorse assicurando la loro effettiva cancellazione dal sistema. Durante tutte le operazioni sulle memtable il sistema non rende disponibili i dati coinvolti nelle operazioni con conseguente riduzione della disponibilità.

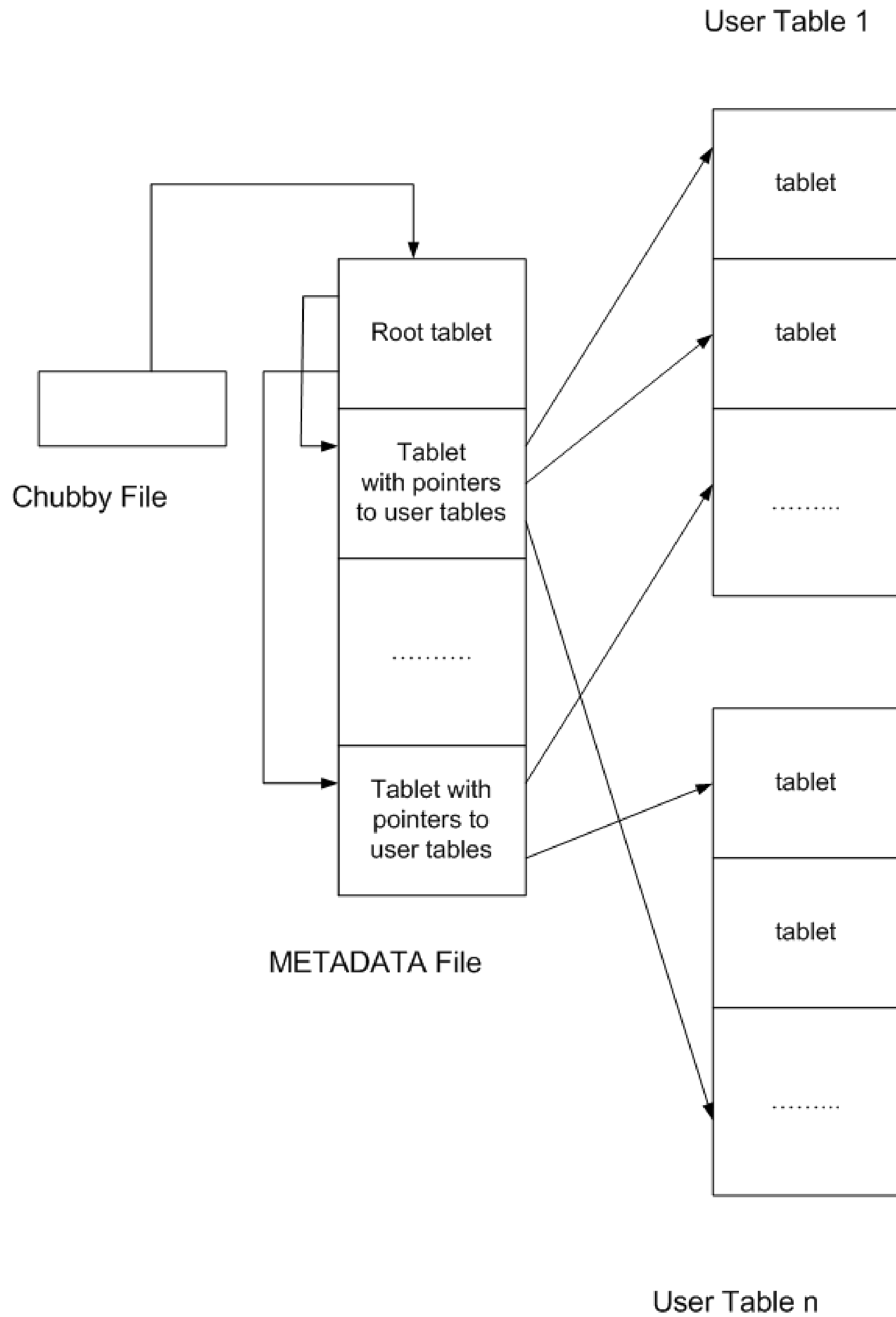
Le operazioni di read sono eseguite su una vista accorpata dei dati presenti sia nelle SSTable che nelle memtable. Nella terminologia BigTable, tale vista è definita **tablet** e rappresenta un concetto fondamentale nell'architettura globale.

Figura 19: BigTable: Architettura Tablet



Un tablet è un insieme di dati raggruppati per intervallo di chiavi e rappresenta l'unità minima sulla quale eseguire partizionamento e distribuzione dei dati con conseguente bilanciamento del carico. Ogni tablet ha un solo nodo server che lo gestisce e per questo è detto **tablet server**. Durante il corso della sua vita un tablet potrebbe avere periodi di transizione dove non è ancora assegnato a un tablet riducendo la disponiblità del sistema nella terminologia CAP. Tutte le informazioni sulla localizzazione dei tablet sono presenti un file di metadati nello spazio del servizio Chubby che ne garantisce un accesso esclusivo e consistente.

Figura 20: BigTable: Tablet Metadati



L'assegnazione è dinamica ed è gestita da un **master server** che può seguire logiche di bilanciamento del carico, gestione membership e reazioni ai fallimenti in maniera trasparente.

All'interno dell'architettura BigTable il **client** ha il compito di snellire il master dal dover smistare le richieste su determinati tablet utilizzando delle informazioni mantenute in locale. Nell'esecuzione delle interrogazioni sul database è il client stesso che smista le richieste direttamente sui singoli tablet server evitando colli di bottiglia sul master.

2.7 Membership e Failure

Quando il tablet server si avvia crea un file con un nome unico in una directory predefinita presente nello spazio Chubby e ne acquisisce il lock esclusivo. Il master contatta periodicamente i server per accertarsi che abbiano ancora il lock sul loro file. Se non arriva risposta il master contatta direttamente il servizio Chubby per verificare se il lock è ancora attivo. Il caso di risposta negativa, il master considera fallito il server cancellando il rispettivo file e spostando, nel file di metadata, i tablet associati in uno spazio non assegnato. Se il server torna attivo smetterà di servire richieste perché si accorge di aver perso il lock sul file. Se un tablet server viene spento dall'amministrazione sarà esso stesso a rilasciare il lock in modo da portare il più velocemente possibile il master a riassegnare i tablet.

Quando è il master ad avviarsi deve creare un file speciale nello spazio Chubby ed acquisirne il lock eslusivo. Il nome e la posizione del file sono stabiliti a priori per garantire la presenza di un unico master nel sistema. Se il master non riesce più ad accedere al lock del file è esso stesso che esce dal sistema non potendo più garantire una corretta gestione dei tablet. È lo stesso servizio a eseguire un algoritmo di elelzione di un nuovo master. Si può dire che Chubby rappresenta il single point of failure di BigTable.

2.8 Implementation

BigTable è un progetto, scritto in C++, propretario di Google ed utilizzato per gran parte dei suoi servizi. Esistono alcune implementazioni Open Source che replicano le idee esposte per progetti open source: **Apache HBase** scritto in Java e **Hypertable** implementato in C++.

2.9 Apache HBase

HBase è in realtà un componente di un progetto più generale che vuole offrire blocchi software per il processamento di dati su larga scala: **Apache Hadoop**. HBase implementa in Java le logiche del servizio BigTable, ma tra i moduli di Hadoop si ha anche l'implementazione delle idee di MapReduce, dei linguaggi di scripting server side **Pig** e **Hive** simili a Sawzall, di un servizio di lock distribuito **ZooKeeper** corrispettivo di Chubby.

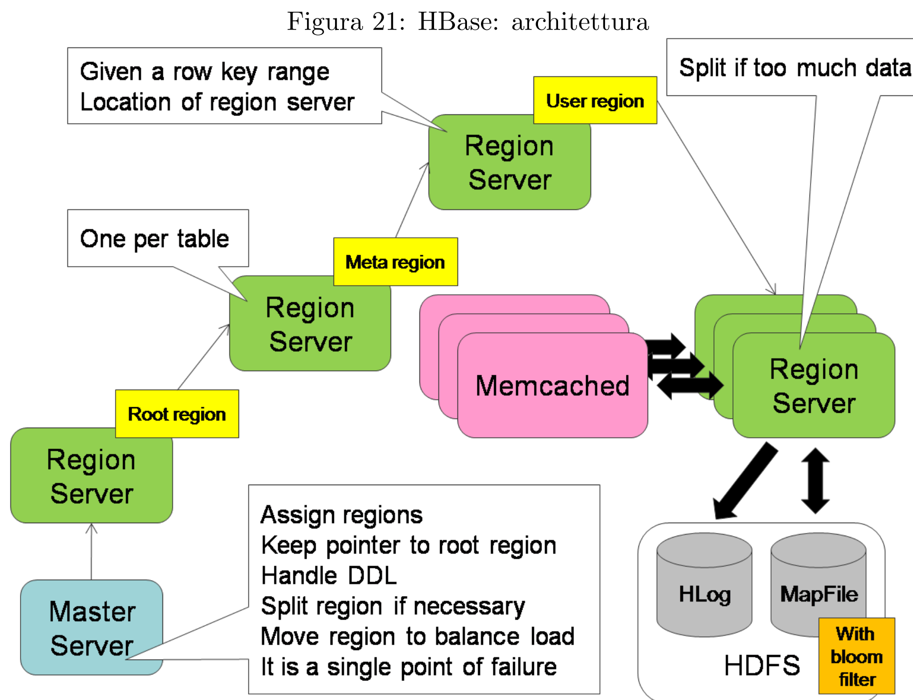
Servizio	Google	Apache Hadoop

Distributed FileSystem	Google FileSystem	Hadoop FileSystem
MapReduce Execution	Google MapReduce	Hadoop MapReduce
Distributed Coordinator	Google Chubby	Hadoop ZooKeeper
Server Side Scripting	Google Sawzall	Hadoop Pig & Hive
Distributed Database	Google BigTable	Hadoop HBase

2.10 Hypertable

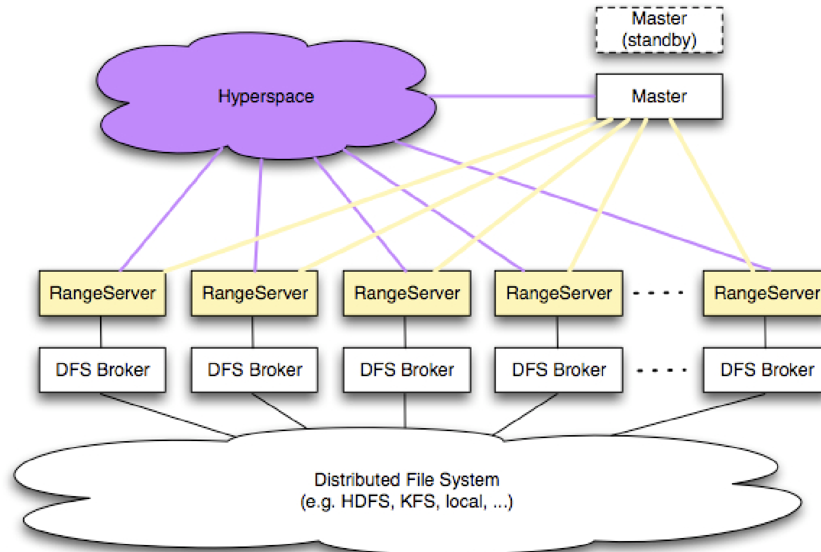
Diversamente da HBase, Hypertable non è legato a nessun file system distribuito, ma introduce uno livello software intermedio, il **Distributed File System broker**, che è stato ideato per lavorare con le implementazioni già esistenti tra cui anche HDFS. Il corrispettivo del Chubby è **hyperspace**.

Figura 21: HBase: architettura



Hypertable utilizza un protocollo di comunicazione basato su **Apache Thrift**, che permette una integrazione con gran parte dei linguaggi utilizzati (Java, PHP, Perl, Python, Ruby. . .).

Figura 22: Hypertable: architettura



2.11 Apache Cassandra

Un'altro noto database, **Apache Cassandra**, ha ereditato il modello di dati ColumnFamily, ma diversamente da HBase e Hypertable, lo ha fatto su un architettura diversa, senza alcun file system distribuito. Inoltre cassandra ha evoluto il concetto di famiglia introducendo un concetto di super famiglia, **Super ColumnFamily**, per implementare una gerarchia tra le colonne.

2.12 Conclusioni

L'introduzione di BigTable nel mondo dei NoSQL ha portato molte novità che sono state riprese anche da altri software. Il modello a famiglie di colonne rappresenta il vero punto di forza del database di Google che lo ha reso più flessibile rispetto ad una visione a chiave valore semplice. In termini del Teorema CAP, BigTable preferisce introdurre una consistenza forte nella singola write a costo di perdere in disponibilità del sistema e ciò lo colloca nella categoria CP.

Problema	Tecnica	Vantaggi
Partizionamento	• Tablet-based • Chubby metadata	Scalabilità orizzontale con bilanciamento dinamico
Consistenza e Isolamento	• Tablet single server association • Commit log on GFS	Massima consistenza sulle scritture con supporto ai redo basato su log
Gestione fallimenti temporanei	• Chubby file lock	Individuazione automatica dei fallimenti da parte del master
Gestione fallimenti permanenti	• Tablet reassocation • Commit log recostruction	Massima garanzia dovuta ai commit log
Membership	• Chubby service	Ingresso uscita dinamici

i vantaggi di BigTable e dei suoi cloni riguardano principalmente il modello di dati.

Vantaggi:

- Modello di dati semplice e flessibile.
- Concetto di ColumnFamily di colonne potente.
- Controllo accessi potente e basato su ColumnFamily.
- Range e filtered query efficienti.
- Supporto a MapReduce.
- Elevata consistenza delle write.
- Replicazione garantita dallo strato di file system.
- Elevate prestazioni tramite gestione in-memory.
- Assenza di colli di bottiglia sul nodo master.
- Bilanciamento dei dati dinamico.
- Gestione membership e failure dinamici.

Svantaggi:

- Proprietario di Google.
- Applicazione client deve gestire le richeiste ai server.
- Nessuna gestione di relazioni tra righe.
- Disponibilità non sempre garantita.
- Single point of failure nel serivizio di lock.

3 Apache Cassandra

Inizialmente sviluppato da Facebook, Cassandra venne continuato dalla comunità di Apache nel 2008 e divenne open source. Da quel momento anche altre compagnia iniziarono ad usarlo, tra le quali Twitter e Digg.

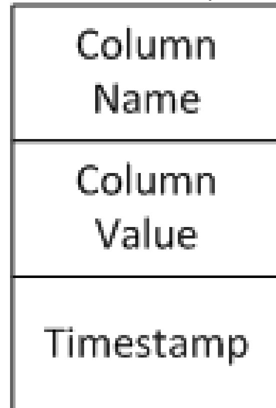
Il caso d'uso con cui venne concepito fu la gestione della posta privata di Facebook. Il popolare social network richiede una mole di dati enorme ed in continua crescita per cui un requisito fondamentale nell'implementazione del database è certamente l'elevata scalabilità in grado di supportare anche centinaia di nodi dislocati in posizioni geograficamente molto distanti tra loro. Il volume di operazioni di write da eseguire non deve impattare l'efficienza nelle read. Inoltre, un temporaneo disservizio per una società come Facebook può avere significativi impatti in termini economici, per cui è necessario garantire un'elevata disponibilità in un sistema dalle dimensioni talmente elevate da rendere standard l'operare anche in presenza di qulache fallimento interno deve essere una condizione standard nodo non funzionante è praticamente standard. Serve dunque evitare il single point of failure.

In letteratura Cassandra via spesso visto come una via di mezzo tra Amazon Dynamo nella scalabilità estrema dei dati, ma mantenendo il modello di dati più potente e flessiblie ideato da BigTable.

3.1 Data Model

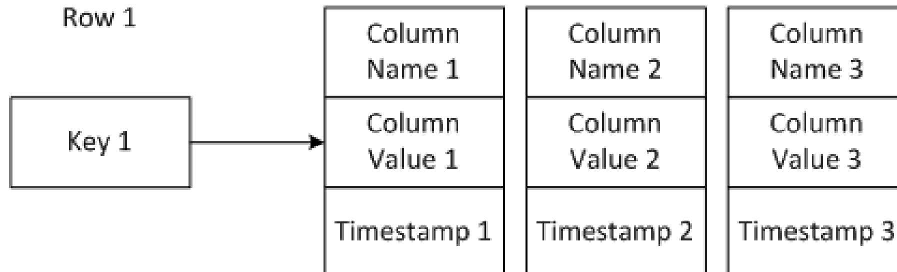
Poiché Cassandra eredita il modello di dati di BigTable, viene spesso usata la stessa definizione di "*tabella sparsa, distribuita, persistente, multidimensionale e ordinata*". Così come accade per il celebre database di Google ogni valore viene fisicamente memorizzato attraverso attraverso una tupla composta da una chiave riga (**row-key**), una chiave colonna (**column-key**) e la colonna (**column**) composta dalla coppia valore (**value**) vista come sequenza non strutturata di byte e marcatore temporale di 64bit (**timestamp**).

Figura 23: Cassandra: Key-Value Storage



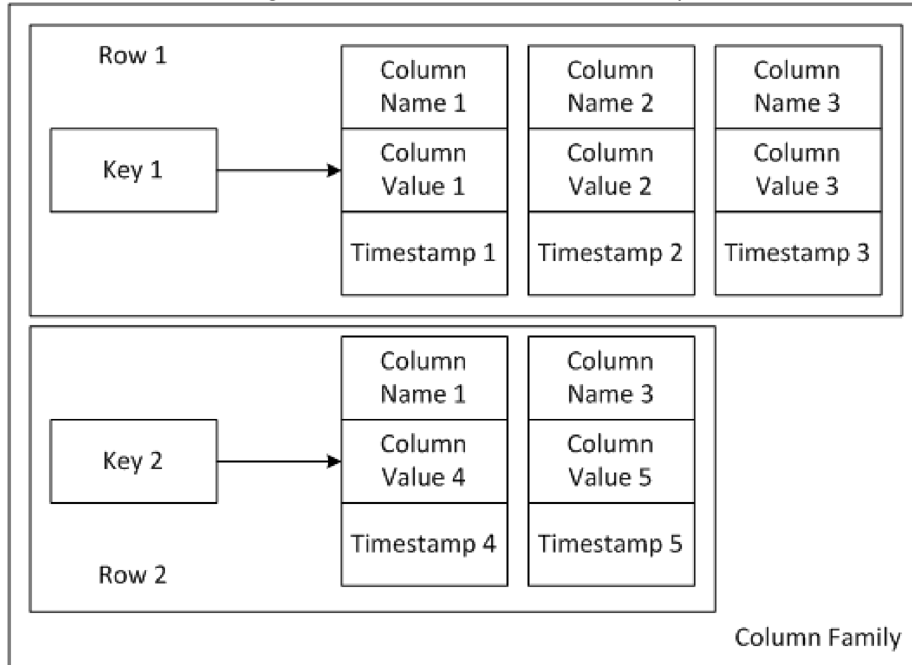
Di conseguenza nella tabella una row-key è presente tante volte quanti sono gli attributi colonna con un valore, da cui il termine multidimensionale. Le chiavi, memorizzate sono mantenute in un ordine lessicologico in modo da favorire le query per intervallo.

Figura 24: Cassandra: Data Model



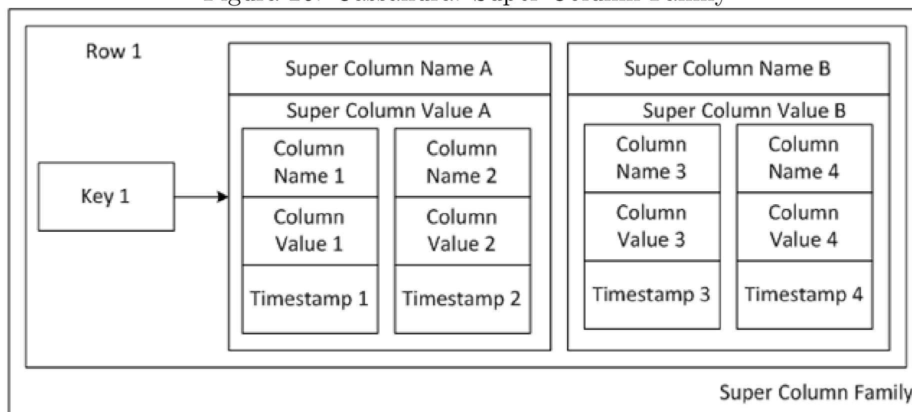
Analogalmente a BigTable, anche in Cassandra il meccanismo con cui sono definite le colonne rappresenta il vero punto di forza del database. Attraverso l'uso di un prefisso, **ColumnFamily**, si possono essere raggruppare righe sotto famiglie diverse in un concetto simile alle tabelle del modello relazionale. Esattamente come accade in BigTable, si possono aggiungere famiglie dinamicamente senza uno schema prefissato, ma è consigliabile limitarne il numero e non variarle troppo velocemente nel tempo.

Figura 25: Cassandra: Column Family



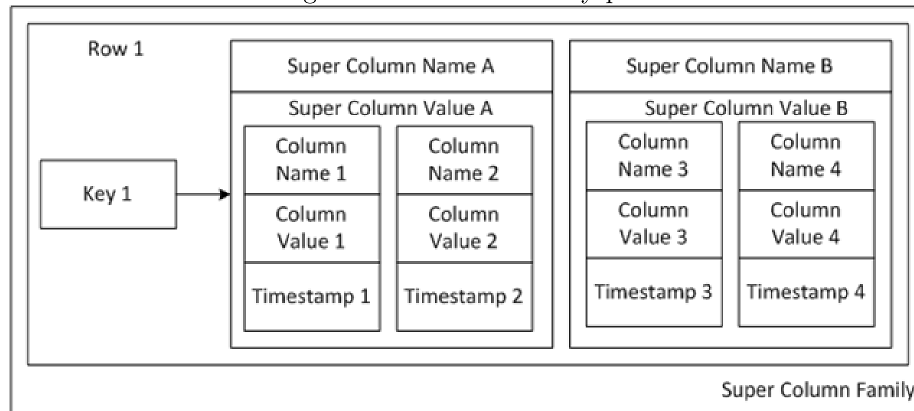
In cassandra il concetto di famiglia è stato ulteriormente evoluto con l'introduzione delle **SuperColumn** che permettono di raggruppare più attributi di una stessa riga sotto un unico nome. In altre parole modo le super colonne introducono una dimensione aggiuntiva all'interno di una riga stessa. Associate alle super colonne si hanno le **SuperColumnFamily** che permette di raggruppare le righe nella visione delle super colonne.

Figura 26: Cassandra: Super Column Family



Infine è aggiunto un livello più generale, il **keyspace**, per identificare dati appartenenti a contesti o applicazioni diverse.

Figura 27: Cassandra: Keyspace



È quindi possibile eseguire un sorta di conversione diretta tra i dati strutturati in un modello relazionale in quello di cassandra.

Modello relazionale	Modello Cassandra
database	keyspace
table	ColumnFamily
record	multiple tuple(row, column, timestamp, value)
primary key	row-key
attribute	column-key
value	value

Questo schema facilita di molto il porting di applicazioni preisistenti, ma rischia di far perdere reale potenza alle ColumnFamily. Per tale ragione, nel modellare una applicazione è consigliabile vedere il modello dei dati come una mappa ordinata di una seconda mappa innestate e anch'essa ordinata.

Mappa di 4 dimensioni:

- Keyspace ⇒ Column Family
- Column Family ⇒ Column Family Row
- Column Family Row ⇒ Columns
- Column ⇒ Data value

Mappa di 5 dimensioni:

- Keyspace ⇒ Super Column Family
- Super Column Family ⇒ Super Column Family Row
- Super Column Family Row ⇒ Super Columns
- Super Column ⇒ Columns
- Column ⇒ Data value

In generale è quindi necessario andare a cambiare l'ottica spessa con cui si modellano i dati per poter usare al meglio cassandra.

3.2 Query Model

Le API espose ai client di Cassandra riprendono la semplicità tipica di Dynamo. Consistono di giusto tre operazioni.

- get(table, key, columnName)
- insert(table, key, rowMutation)
- delete(table, key. columnName)

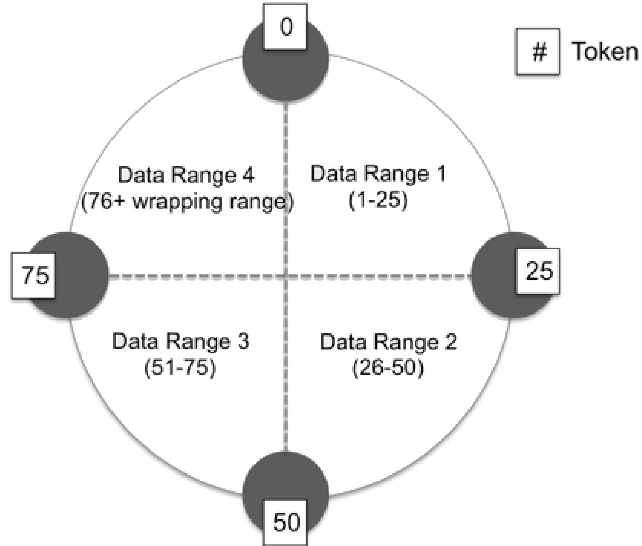
Nelle operazioni di get e insert indentificano sia una Column che una SuperColumn all'interno di una ColumnFamily oppure una ColumnFamily stessa.

Le query possono essere scritte in un linguaggio simile a SQL oppure utilizzate tramite il protocollo Thrift.

3.3 Sharding

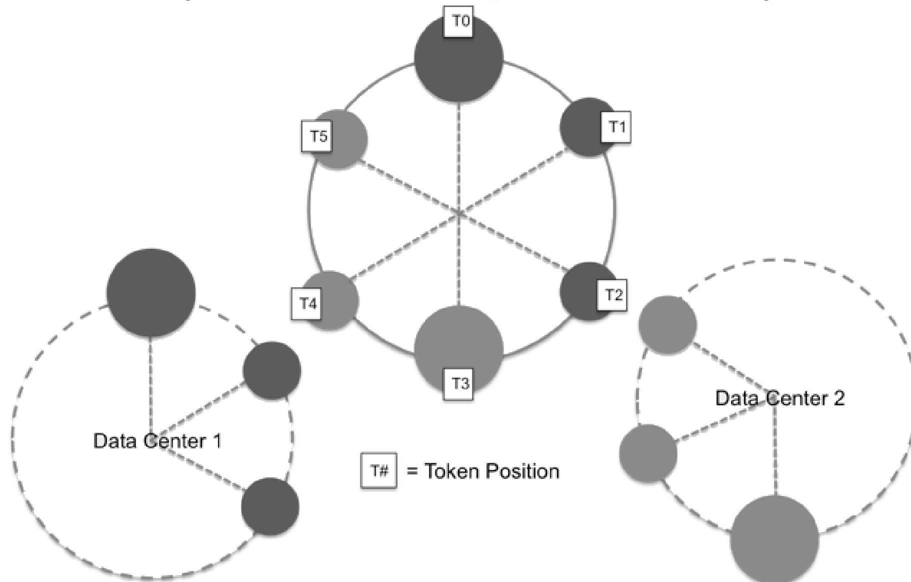
Cassandra utilizza la tecnica del **Consistent Hashing** in cui i nodi della rete formano un anello. Poiché Cassandra è stato ideato pensato in contesti di elevato dinamismo della rete, questa tecnica permette di ridurre al minimo la quantità di dati da dover rimappare al variare della quantità di nodi presenti. Nella sua versione di base, le chiavi vengono distribuite in maniera uniforme senza contare le reali risorse hardware delle macchine fisiche.

Figura 28: Cassandra: Sharding via Consistent Hash



Dynamo risolve questo problema facendo girare tanti più nodi virtuali tanti più alte sono le capacità computazioni della machina fisica. Cassadra usa una via diversa, suddivide il datacenter in diversi anelli e sposta i nodi al fine di migliorare l'utilizzo delle risorse disponibili.

Figura 29: Cassandra: Multiple Datacenter Sharding



Nella configurazione iniziale di Cassandra è possibile selezionare due strategie di partizionamento. Una casuale (**RandomPartitoner - RP**) che distribuisce le coppie

chiave-valore con il vantaggio di avere un buon bilanciamento dei dati a discapito di dover contattare diversi nodi per ricostruire una riga. Oppure è possibile designare un ordinamento che preserva l'ordine (**OrderPreservingPartitioner** - **OPP**) distribuendo le coppie in modo tale che le stesse chiavi non siano distanti tra loro.

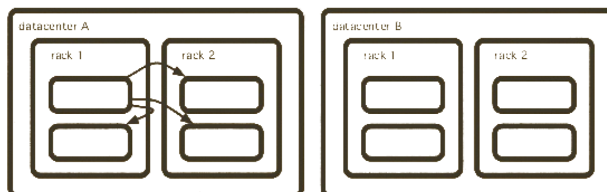
3.4 Replication

La replicazione è gestita da un nodo **coordinator** che per ogni chiave è il nodo designato dalla funzione di consistent hashing come responsabile del range di valori in cui la chiave stessa appartine.

Nella configurazione di Cassandra è possibile definire diverse strategie di replicazione in cui è sempre possibile definire il fattore di replicazione N, ossa il numero di nodi che devono gestire la stessa chiave.

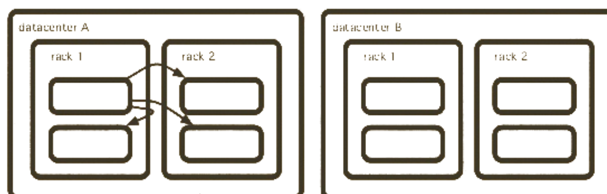
Rack Unaware replicazione di base gestita dentro un unico anello in cui si eseguono tante copie quante sono definite dal fattore di replicazione N nei nodi successivi a quello responsabile di una determinata chiave, muovendosi in senso orario.

Figura 30: Cassandra: Rack Unaware



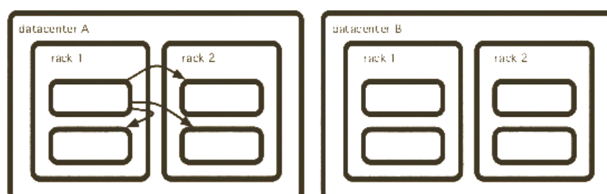
Rack Aware una strategia che sposta una delle N replica su un altro anello

Figura 31: Cassandra: Rack Aware



Datacenter Shard una strategia che replica un insieme M degli N nodi desiganti dal fattore di replicazione in un altro anello.

Figura 32: Cassandra: Datacenter Shard



E anche possibile definire una strategia di replicazione custom e integrarla in fase di configurazione.

3.5 Consistency

In Cassandra la consistenza si riferisce a come aggiornare e sincronizzare una riga di attributi tra tutte le repliche. Nella sua implementazione estende il concetto di **eventual consistency** offrendo una consistenza il cui livello di robustezza, **tunable consistency**, è variabile e può essere deciso da client stesso sia nelle operazioni di lettura che di scrittura.

Nelle write il livello di consistenza dipende fortemente dal numero W delle repliche N ($W <= N$) che devono completare con successo la scrittura sia sulle proprie tabelle e sia sui file di log. Se tutti i nodi sono irraggiungibili (W = 0) allora si può usare la tecnica opzionale dell'hinted handoff in cui si designa un qualunque altro nodo a mantenere temporanemanete quella chiave a consegnarla al primo nodo che torna operativo. In questo lasso di tempo il valore non sarà leggibile dalle successive read e per tale ragione questo livello può essere considerato il più basso.

Livello	W	Descrizione
ANY	W=0	Write su almeno un nodo con tecnica di hintend handoff nel caso in cui nessun nodo è reggiungibile.
ONE	W=1	Conferma di successo della write da almeno un nodo.
TWO	W=2	Conferma di successo della write di almeno due nodi.
THREE	W=3	Conferma di successo della write di almeno tre nodi.
QUORUM	W<N	Conferma di successo della write di un numero fissato di nodi.

LOCAL_QUORUM	W<N	Conferma di successo della write di un numero fissato di nodi nello stesso datacenter.
EACH_QUORUM	W<N	Conferma di successo della write di un numero fissato di nodi di tutti i datacenter.
ALL	W=N	Conferma di successo della write di tutte le repliche designate.

Nelle read il livello di consistenza specifica il numero di nodi R delle repliche N $(R <= N)$ che deve rispondere prima di consegnare il risultato al client. Cassandra controlla su queste risposte il dato più recente secondo il timestamp associato. Similmente alle write anche nelle read abbiamo livelli di consistenza che vano dal più basso ONE al più forte ALL.

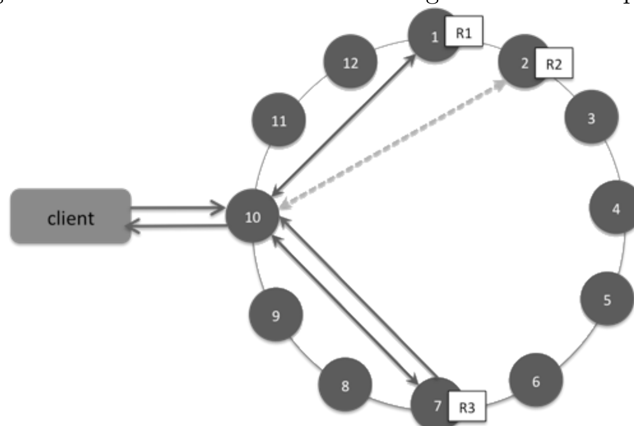
Livello	W	**Descrizione**
ONE	R=1	Ritorna una risposta dalla replica più vicina. Per default una read repair viene eseguita in background.
TWO	R=2	Ritorna una risposta delle due repliche più vicine.
THREE	R=3	Ritorna una risposta delle tre repliche più vicine.
QUORUM	R<N	Ritorna il valore con il timestamp è più recente dopo che un valore di soglia di repliche ha risposto.
LOCAL_QUORUM	R<N	Ritorna il valore con il timestamp è più recente dopo che un valore di soglia di repliche ha risposto. Evita la latenza di repliche tra datacenter diversi.
EACH_QUORUM	R<N	Ritorna il valore con il timestamp è più recente dopo che un valore di soglia di repliche ha risposto. Impone che la risposta avvenga da ogni datacenter
ALL	R=N	Ritorna il valore con il timestamp più recente dopo che tutte le repliche rispondono. La richiesta fallisce se anche una sola replica non risponde.

I livelli **ONE**, **TWO**, **THREE** e **QUORUM** e **ALL** rappresentano una sorta di scala da livello di consistenza più basso a quello più forte. Nel caso si voglia evitare la latenza di comunicazione si può imporre il livello **LOCAL_QUORUM** oppure se l'affidabilità di avere dati dislocati su diversi datacenter è necessaria si può imporre il livello **EACH_QUORUM**. Se la strategia di partizionamento configurata in Cassandra non prevede l'utilizzo di più datacenter allora la richiesta fallisce sempre sia in LOCAL che EACH.

Le richieste di write sono sempre inoltrate su tutte le repliche di tutti datacenter in quanto di consistenza indica solo quante di esse devono completare la scrittura sia sul commit log che sulle proprie tabelle e confermare il successo al coordinatore. Per le read, invece, esistono due tipologie di richieste che il coordinatore può effettuare. Una **direct read** il cui numero di richieste effettuate corrisponde al livello di

consistenza desiderato. La **background read** via inviata a tutte le altre repliche. Per garantire che i dati letti più frequentemente si mantengano consistenti il coordinatore controlla i valori letti e in invia richieste di write per aggiornare le repliche obsolete.

Figura 33: Cassandra: Direct and Background Read Requests



Questa tecnica viene detta **read repair** ed è abilitata per default. Anche l'integrità delle read può essere configurata come **weak consistency** o **strong consistency** se il controllo di integrità viene effettuata prima di ritornare i risultati.

Per i dati letti meno frequentemente o per gestire nodi che falliti temporaneamente un processo di **anti-entropy node repair** assicura che tutti i dati di una replica siano aggiornati. Questo processo può essere eseguito anche a per regolari fasi di mantenimento del cluster (non dovuti a fallimenti).

I client può decidere il livello di consistenza delle richieste in base alle esigenze dell'applicazione. Il livello ONE nelle read è tipicamente usato in contesti dove la bassa latenza è una priorità rispetto alla probabilità di leggere un dato obsoleto. Il livello ALL viene utilizzato nelle write dove c'è l'assoluta necessità di non far fallire mai una write anche a costo di perdita della disponibilità.

Più in generale è possibile impostare il giusto trade-off giocando con i valori R e W. Se la consistenza ha un'elevata priorità è necessario assicurare che la somma dei numero di write assicurate W e delle read ricevute R sia maggiore nel numero di repliche stesse.

$$R + W > N \tag{2}$$

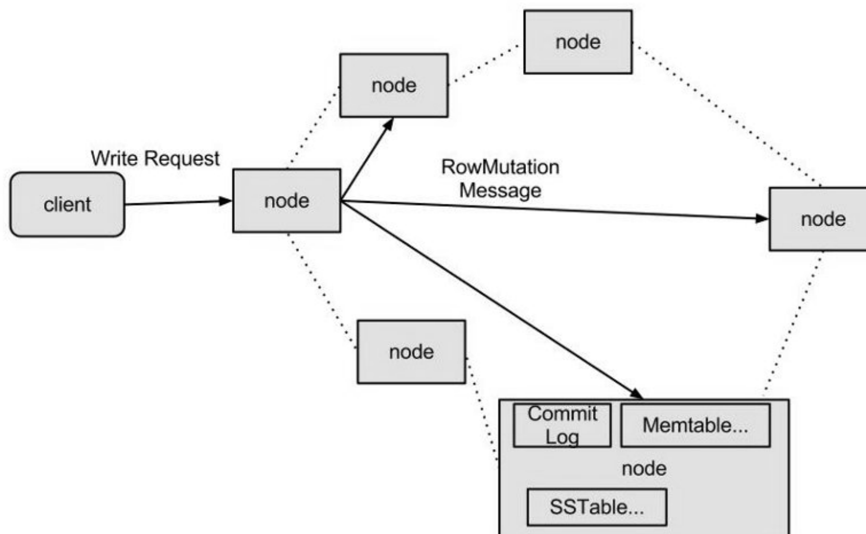
Il motivo è che si vuole garantire che l'insieme dei nodi che eseguono la più recente delle write interseca un qualunque insieme di nodi che risponde ad una successiva read. In altre parole, tra le repliche che rispondono a una richiesta di lettura di un dato, ci saranno (R+W)-N nodi che hanno il valore più recente.

3.6 Architecture

L'architettura di Cassandra è **shared nothing**, in cui i nodi formano un anello senza un master pincipale. Il client può contattare un qualunque nodo che diventerà il coordinatore della richiesta per individuare quali sono i nodi responsabili di una determinata chiave. Il bilanciamento del carico è fortemente dipendente dal tipo di partizionamento che è stato configurato per il cluster. Tutte le informazioni topologiche sono memorizzate nel servizio di snich che mappa gli ip ai rack e datacenter definendo come i nodi sono raggruppati. Ogni nodo di uno stesso claster deve utilizzare lo stesso snich che può essere configurabile.

Essendo shared nothing, Cassandra rende persistente i dati solo in locale senza utilizzare un servizio di file system distribuito. Nonostante ciò, le modalità con cui i dati vengono resi persistenti ricordano le idee sviluppate da BigTable. Similmente ai tablet server, anche i nodi cassandra hanno un modulo di scrittura delle write che salva i commit su un file di log e su una struttura in-memory che viene salvata su disco in maniera asicnrona una volta raggiunta una dimensione soglia. Le scrittura su disco avvengono in un formato indicizzato per la lettura basata su chiave in maniera analoga agli indici di blocco nei file SSTable con la differenza che Cassandra aggiunge anche indici sulle column-family e sulle column. All'aumentare dei file su disco, un processo in background esegue compressione e accorpamento alf ine di ottimizzare le risorse su disco. Le operazioni di read, esattamente come in BigTable sono il risultato della visione unificata tra le informazioni in-memory e quelle su disco.

Figura 34: Cassandra: Architecture



3.7 Membership e Failure

Il servizio di membership in Cassandra è gestito attraverso un particolare protocollo di tipo GOSSIP noto come Scuttlebutt che offre vantaggi in termini di efficienza nell'utilizzo delle CPU e della rete.

Quando un nodo si aggiunge al cluster, calcola un token casuale per trovare la sua posizione all'interno dell'anello e l'intervallo di chiavi di cui deve essere responsabile. Tali informazioni sono memorizzate sia in locale e sia su un servizio di sincronizzazione distribuito simile a ZooKeeper per BigTable. Una volta entrato nella rete il nodo inizia a cercare indirizzi degli altri sia dalle configurazioni locali e sia dal servizio ZooKeeper ed annuncia ad essi la sua presenza lanciando, così, il protocollo GOSSIP che porterà l'intera rete sulla nuova membership.

Io nodi della rete Cassandra provano a identificare se un nodo è attivo o meno utilizzando un meccanismo di **failure detector** in cui non si determina in maniera booleana il fallimento di un nodo, ma se ne determina un livello di sospetto. Questo meccanismo permette un'ottima accuratezza e buona velocità nel reagire alle condizioni della rete.

3.8 Implementation

Tutti i modoli che permettono a una macchina di entrare in una rete Cassandra sono scritti in Java e operano su un software orientato agli eventi in cui lo scambio di messaggi utilizza il trasporto UDP per le operazioni di systema, mentre TCP per rispondere alle richieste del client.

3.9 Conclusioni

Cassandra viene spesso visto come il punto di incontro tra l'architettura fortemente orientata alla distribuzione e alla scalabilità di Dynamo con il modello dei dati potente e flessibile di BigTable.

Con il database di Amazon Cassandra condivide l'architettura P2P in cui l'assenza di nodi speciali rende il sistema assente da colli di bottiglia e punti critici per il fallimento del sistema e la conseguente la gestione del cluster basata su protocollo GOSSIP. Anche la consistenza eventuale a livello configurabile da client può essere vista come una evoluzione di Dynamo.

Da BigTable cassandra riprende la visione di modello di dati a mappa sparsa e orientata alle colonne così come i meccanismi con cui vengono resi persistenti attraverso il connubio di una gestione in-memory e di una basta su file immutabili sullo stile di SSTable. Le ultime versioni di Cassandra si integrano con Hadoop l'insieme di software che riprendono le tecnologie di Google di cui BigTable ne era solo una parte.

Problema	Tecnica	Vantaggi
Partizionamento	• Consistent Hash • Multiple DataCenter	• Minimizzazione delle reallocazioni • Elevata configurabilità della strategia di partizionamento
Consistenza e Isolamento	• Tunable consistency • Read repair	• Adattabilità alle esigenze delle applicazioni • Integrità automatica
Gestione fallimenti temporanei	• Failure Detector • Hinted Handoff	• Affidabile • Veloce adattabililtà
Gestione fallimenti permanenti	• Consistent Hash repartition	• Minimizza la quantità di dati da reallocare
Membership	• GOSSIP protocol	• Assenza di un nodo speciale

Gran parte dei vantaggi risiedono nel buon compromesso trale idee di due database molto potenti e sviluppati. In generale Cassandra può essere vista come una delle migliori implementazioni di sistemi shared nothing da cui ne trae gran parte dei vantaggi e svantaggi. In termini del teorema CAP il database si colloca sulla classificazione di tipo AP, ossia di database orientato alla scalabilità e alla disponibilità a discapito di una consistenza non garantita da uno strato di file system distribuito. Inoltre Cassandra richiede una amministrazione continua del database in relazione all'andamento dei dati.

Vantaggi:

- Reso Open source.
- Modello dei dati di BigTable con supporto alle SuperColumn.
- Range e filtered query efficienti.
- Livello di consistenza dinamico.
- Controllo di integrità e riparazione automatica.
- Elevatissime prestazioni e scalabilità.
- Elevata disponibilità.

- Assenza di colli di bottiglia sul nodo master.
- Bilanciamento dei dati dinamico.
- Gestione membership e failure dinamici.
- Elevata configurabilità.

Svantaggi:

- Sistema di difficile amministrazione.
- Fortemente dipendete dalla strategia di partizionamento.
- Supporto a MapReduce non nativo.
- Nessuna gestione di relazioni tra righe.
- Consistenza non sempre garantita.

4 Mongo DB

MongoDB rappresenta una delle più famose implementazioni di database basato su documenti. Scritto in linguaggio C++ è orientato alla velocità e scalabilità pur mantenendo un buon livello di consistenza. Può essere usato anche per memorizzare e distribuire file binari di grandi dimensioni quali immagini o video. Questo database mira a colmare l'enorme distanza tra la velocità e la scalabilità tipica dei database chiave-valore con le ricche funzionalità dei database relazionali.

4.1 Data model

La visione dei dati di MongoDB è **document-based** che viene spesso considerata come un modello di dati chiave-valore di nuova generazione. In tutti i DBMS NoSQL classici il valore memorizzato viene trattato come un'informazione binaria e si demanda all'applicazione sovrastante la gestione della struttura dei dati. La filosofia di questi database è quella di rilassare le funzionalità offerte dal database in modo da ridurlo a poco più che un semplice servizio di storage che risulta facilmente esportabile sopra un cluster di macchine, come noto dal Teorema CAP. L'esempio più estremo è Dynamo di Amazon, un database che offre un servizio di salvataggio dei dati come semplici tuple chiave valore. L'approccio di BigTable ha aggiunto una minima strutturazione dei dati in famiglie e colonne che lo avvicina di più al modello a record dei database relazionali, ma di fatto il dato effettivo resta ancora una semplice sequenza di byte. Nei database a documenti le informazioni vengono salvate secondo uno formato, tipicamente **JSON** o **XML**, che permette di vedere i valori come serializzazione degli oggetti di una applicazione. Vengono spesso visti come l'evoluzione dei database ad oggetti in cui lo strato di persistenza non è più strettamente legato al linguaggio dell'applicazione, ma solo allo standard testuale con cui salvare i documenti.

Figura 35: MongoDB: Document Model

```
{
  name: "sue",                          ←— field: value
  age: 26,                              ←— field: value
  status: "A",                          ←— field: value
  groups: [ "news", "sports" ]          ←— field: value
}
```

In MongoDB i documenti sono salvati in **BSON** (**Binary JSON**) che è una estensione del formato JSON con il supporto a tipi aggiuntivi. Ad ognui campo è possibile inserire un valore di tipo **double, string, array, binary, object id, boolean, date, null, regular expression, javascript, symbol, int32, timetamp, int64, minkey, maxkey**. Esistono una serie di convenzioni che permettono a

MongoDB di riconoscere documenti in JSON e trattarli come BSON. La struttura interna dei documenti non è legata ad uno schema.

Figura 36: MongoDB: BSON Format

```
{"hello": "world"}                  → "\x16\x00\x00\x00\x02hello\x00
                                       \x06\x00\x00\x00world\x00\x00"
                                      "\x31\x00\x00\x00\x04BSON\x00\x26\x00
                                       \x00\x00\x020\x00\x08\x00\x00
{"BSON": ["awesome", 5.05, 1986]} →  \x00awesome\x00\x011\x00\x33\x33\x33\x33\x33\x33
                                       \x14\x40\x102\x00\xc2\x07\x00\x00
                                       \x00\x00"
```

L'applicazione può, quindi, aggiungere e rimuovere un campo senza restrizioni garantendo la massima flessibilità, con solo alcune restrizioni relative principalmente alla presenza di alcuni campi e caratteri riservati.

Oltre ai documenti MongoDB introduce il concetto di **Collection** che permette di raggruppare documenti simili tra loro.

Figura 37: MongoDB: Collection Model

Le collezioni possono essere viste come l'equivalente delle tabelle in un database relazionale che accorpano record sotto forma di righe. La differenza sostanziale risiede nel fatto che i documenti di una stessa collezione non forza la struttura interna come accade con gli attributi delle tabelle. Tipicamente i documenti di una collection hanno uno stesso scopo, ma possono avere campi diversi. In qualche modo è possibile vedere un analogia con le ColumnFamily dei database derivati da BigTable che possono avere colonne diverse. All'interno di una collection tutti i documenti hanno una chiave univoca.Sulle collection è possibile definire degli indici per ottimizzare la ricerca interna ai campi.

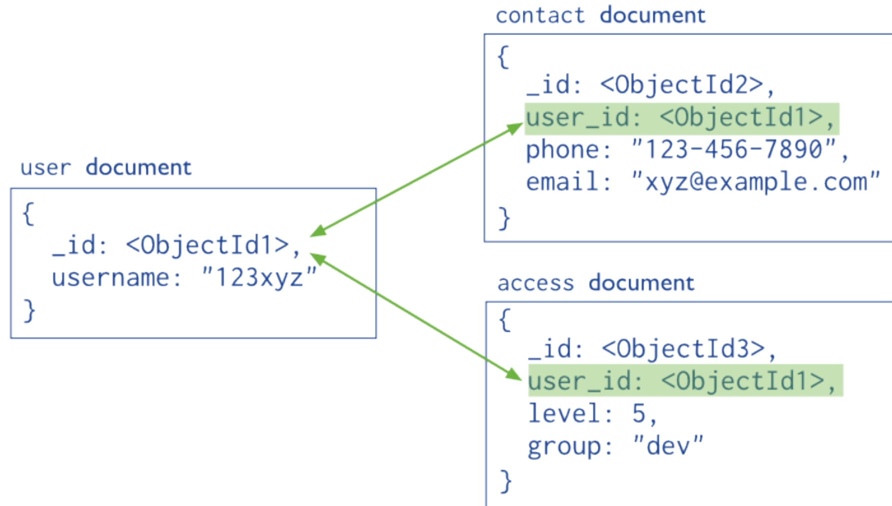
Similmente a come accade con tutti i database NoSQL, MongoDB non supporta il collegamento diretto tra due documenti attraverso operazioni simile alle JOIN. Questo ha un forte impatto nella modellazione dei dati che deve considerare la scelta tra ad un modello embedded (denormalizzato) o nomalizzato.

Embedded Data Model una modellazione in un o più documenti sono inseriti come sotto campo di un documento padre sfruttando al struttura ad albero/array di BSON. Questo approccio tipicamente porta ad avere i dati denormalizzati che possono essere letti con elevata velocità. Generamente si adatta bene per modellare relazioni di tipo "*contenimento*" stretto da due oggetti in analogia con le relazioni di aggregazione ("*has a*") dei diagrammi UML.

Figura 38: MongoDB: Embedded Data Model

Normalized Data Model una modellazione a referenze esterne attraverso l'uso delle chiavi univoche dei documenti in analogia con le chiavi esterne del modello relazionale. Un documento salva la referenze come valore di un proprio campo ed è possbile farlo in maniera manuale oppure rispettando una convezione nota come *DBRef*. qualunqueÈ possibile farlo in maniera manuale oppure utilizzando una convezione standard Mongo supporta due tipologie di relazioni: manuale o attraverso l'uso di dbref. Non supportando le JOIN nativamente è compito dell'applicazione client il dover risolvere il collegamento attraverso una seconda query. Questo modello si adattata bene per relazioni più complesse e per rappresentare gerarchie tra dataset di grandi dimensioni.

Figura 39: MongoDB: Normalized Data Model



Complessivamente esiste una analogia tra i concetti del database relazionale e quelli di MongoDB.

SQL	Mongo	Differenze
database	database	
table	collection	le collection non impongono restrizioni sugli attributi
row	document	valori come oggetti strutturati
column	field	maggior supporto a tipi di dato
index	index	
join	embedding/linking	gestione delle referenze lato applicazione
primary key	object id	

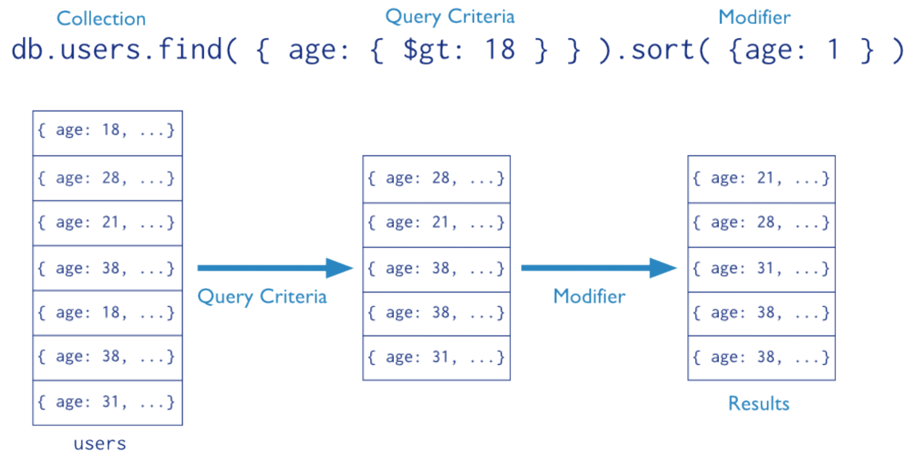
4.2 Query Model

Un'applicazione può comunicare con una istanza MongoDB attraverso una libreria client nota come driver che gestisce le interazioni con un linguaggio appropriato.

Similmente a come accade per i database relazionali, il driver supporta le operazioni **CRUD** (**Create**, **Read**, **Update** e **Delete**) che sono specificate attraverso un formato anch'esso a documento BSON.

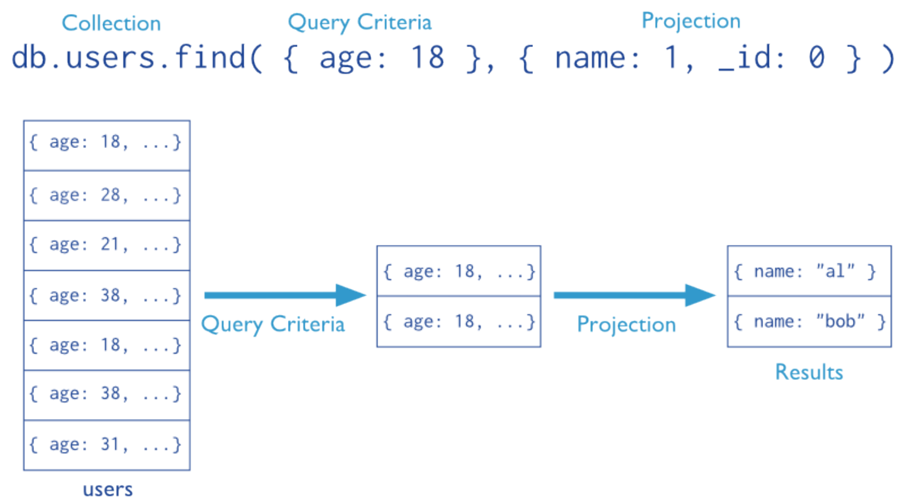
Le operazioni di read possono essere effettuate attraverso delle query su una sola collection per stabilire i criteri sui singoli campi con i quali restringe la collezione ad un sottoinsieme di documenti da ritornare al client. Sul risultato è possibile effettuare una proiezione su particolari campi e sono possibili operazioni di ordinamento o paginazione.

Figura 40: MongoDB: Query Criteria



Sulle query è possibile imporre delle proiezioni che limitano i risultati a un sottoinsieme dei campi dei documenti in modo da ridurre l'overhead di rete.

Figura 41: MongoDB: Projection



A livello di API le query si effettuano tramite una chiama alla funzione *db.collection.find()* che accetta sia i criteri che le proiezioni e ritorna un cursore sui documenti risultanti.
In seguito se ne mostra un esempio.

Figura 42: MongoDB: Find Method

```
db.users.find(                       ← collection
   { age: { $gt: 18 } },             ← query criteria
   { name: 1, address: 1 }           ← projection
).limit(5)                           ← cursor modifier
```

Che corrisponde, in terminologia SQL, alla seguente query.

Figura 43: MongoDB: Find to SQL

```
SELECT _id, name, address    ← projection
FROM   users                 ← table
WHERE  age > 18              ← select criteria
LIMIT  5                     ← cursor modifier
```

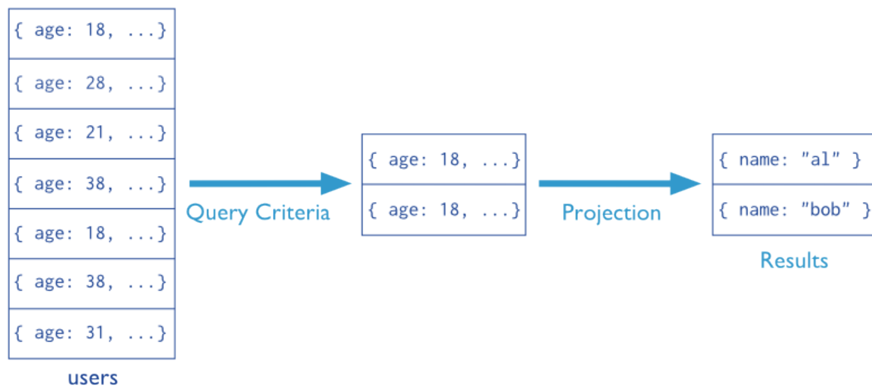
Un'altra via per processare i dati di una collection è l'uso delle aggregazioni.

Aggregation pipeline introdotta nelle versioni più recenti è una forma di aggregazione in cui i documenti entrano in una pipe-line a più stati per per essere ritornati sotto una forma aggregata. Una delle forme più comuni è quella di filtrare i documenti in maniera analoga alla query per poi trasformali in un formato diverso. In generale questa forma di aggregazione può essere usata come alternativa più performante a map-reduce per operazioni di complessità minore.

Figura 44: MongoDB: Aggregation pipeline

Collection Query Criteria Projection

```
db.users.find( { age: 18 }, { name: 1, _id: 0 } )
```



MapReduce supporto per aggregazioni nella forma di MapReduce introdotta da BigTable. Dalla funzione di map, eseguita su tutti i documenti di una collection, vengono emessi altri oggetti che verrano raggrupati attraverso la funzione reduce.

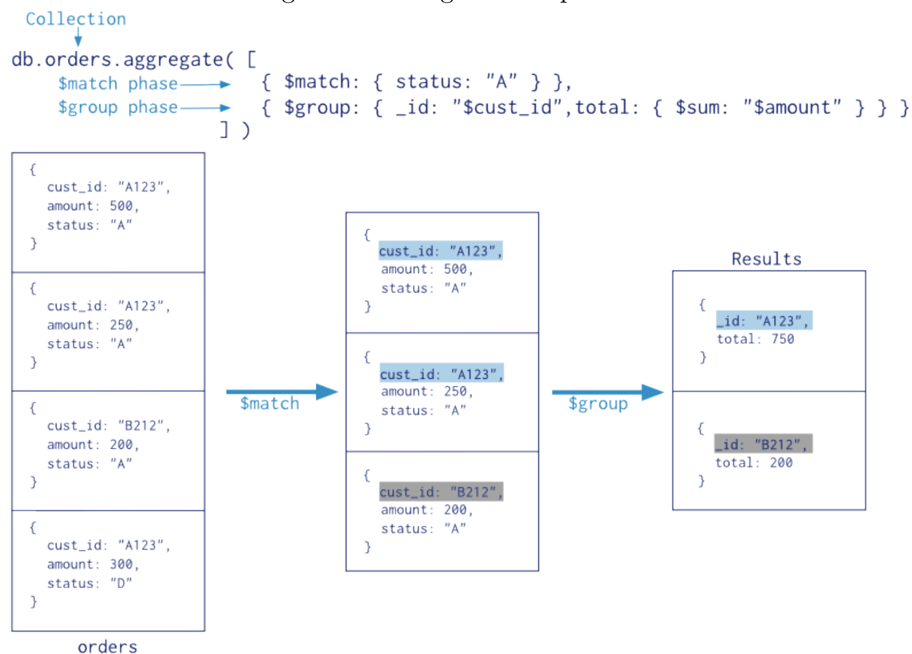


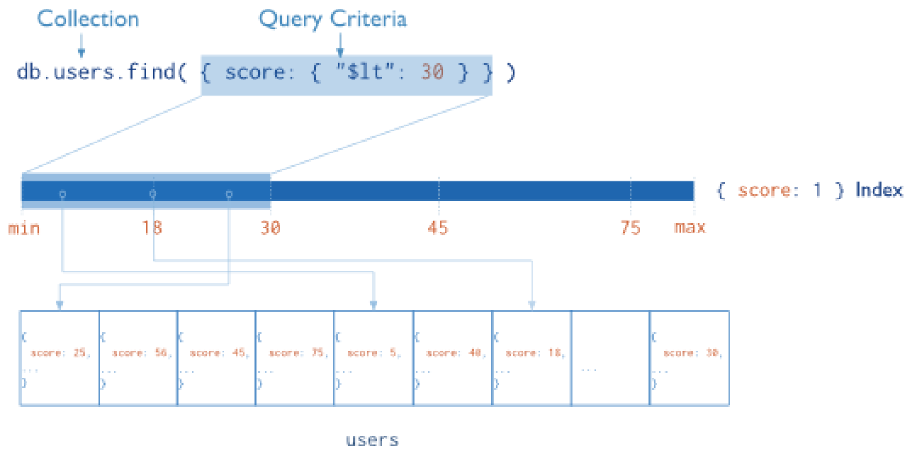
Figura 45: MongoDB: MapReduce

Single Purpose per le operazioni di aggregazione più comuni da applicare a un'intera collezione si possono utilizzare delle operazioni speciali. Queste comprendono scopi del tipo contare i documenti presenti in una collection o eliminarne i duplicati eventualmente è possibile raggruppare per campi secondo un'ottica simile a GROUP BY di SQL.

Per query su larga scala l'esecuzione di un processo di parsing del documento per prelevare il valore di un campo potrebbe incidere in maniera pesante sulle prestazioni. Per tale scopo MongoDB definisce degli indici secondari il cui scopo è analogo a ciò che fanno i database relazionali: ottimizzare l'accesso ai campi più frequenti attraverso l'uso di B-Tree. I campi indicizzati permettono di mantenere un ordinamento utile per le query su intervalli. Per default tutte le chiavi degli oggetti hanno un indice. Dalla versione 2.4, l'ultima attualmente disponibile, sono stati introdotti gli indici di tipo hash utile nel partizionamento su larga scala. Gli indici sono sempre definiti a livello di collection.

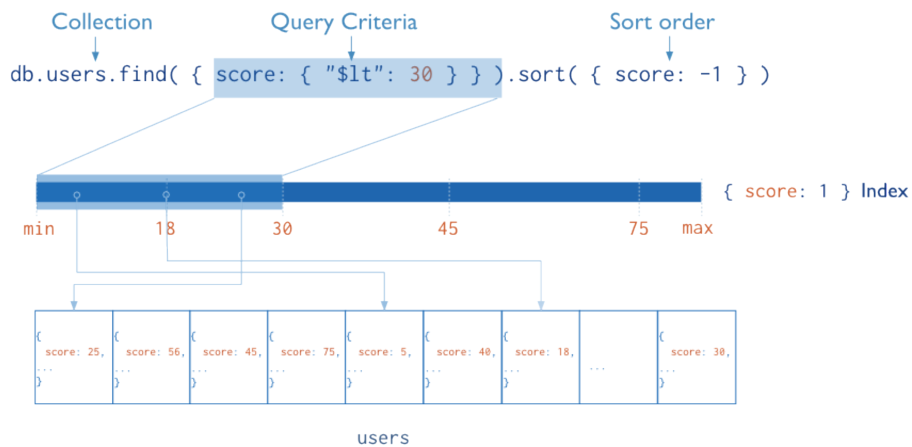
Gli indici sono delle strutture dati speciali che memorizzano una piccole porzione dell'intero dataset di una collezione in una maniera ottimizzata per il traversing di una query.

Figura 46: MongoDB: Indexing



Attraverso gli indici è possibile anche avere ordinamento su campi diversi dalla chiave.

Figura 47: MongoDB: Indexing

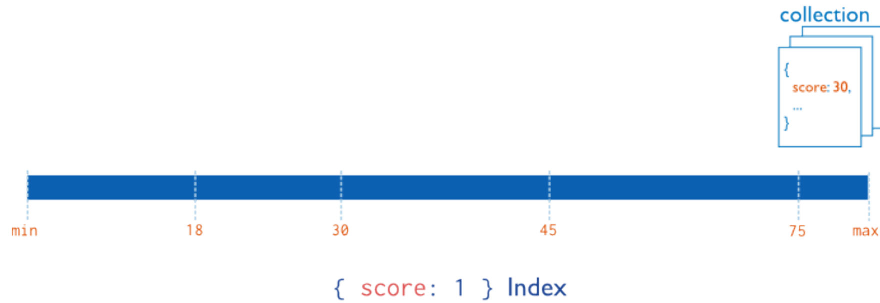


MongoDB supporta diversi tipi di indici.

Default ID tutte le collezioni di MongoDB hanno un indice implicito sull'id del documento stesso che viene memorizzato nel campo "*_id*". Se l'applicazione non specifica un valore esplicito della chiave allora sarà il motore stesso a crearne uno univoco.

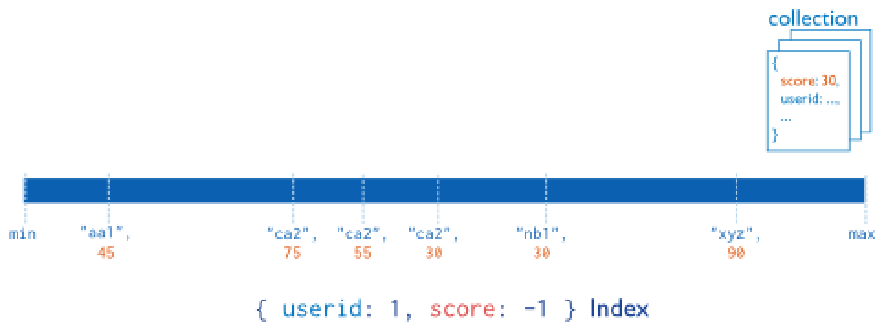
Single Field oltre a indicizzare il campo chiave Mongo supporta la possibilità di definire indici secondari su campi custom come illustrato in figura.

Figura 48: MongoDB: Single Field Index



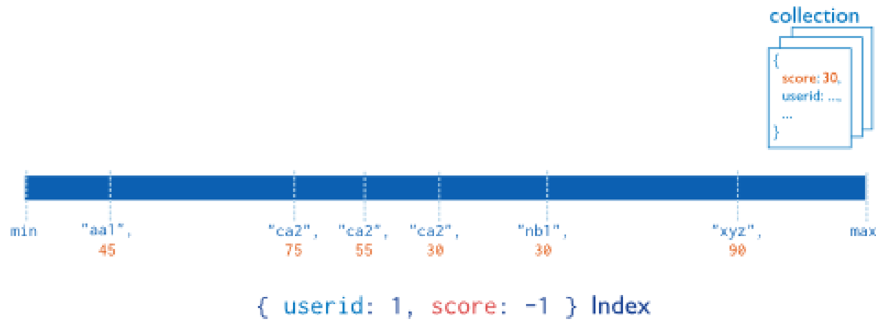
Compound Index MongoDB supporta anche degli indici su campi multipli. Il loro comportamento è simile a singoli campi e quindi possono essere usati nei criteri delle query. L'ordine di tale indice è importante: se ad esempio viene definito l'indice sui campi multipli *userid* e *score* , l'ordinamento avviene guardando prima id utente e poi il suo punteggio. La figura mostra tale esempio.

Figura 49: MongoDB: Compound Index



Multikey Index MongoDB usa gli indici su chiave multipla per indicizzare il contenuto di array. Se un indice di un campo mantiene un array di valori allora il motore crea tanti indici quanti sono le sue entry. Queste chiavi permettono alle query di poter imporre criteri su tutti o alcuni elementi di un array. La creazione di un indicie su chiave multipla o meno viene determinato da MongoDB stesso in base al calore corrente del campo evitando che sia lo sviluppatore a doverlo specificare esplicitamente.

Figura 50: MongoDB: Compound Index



Geospatial Index per supportare query ottimizzare a delle coordinate geospaziali dei dati MongoDB fornisce indici bidimensionali o sferici

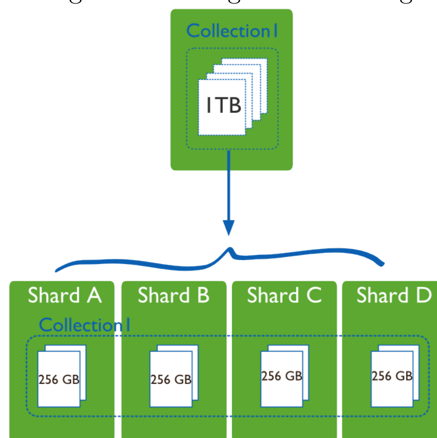
Text Indexes MonboDB fornisce anche il supporto a ricerche di tipo full-text del contenuto dei documenti riuscendo anche ad eliminare dalla ricerca parole poco significative ed eseguendo sempre lo stamming delle parole chiave.

Hashed Indexes Per supportare lo sharding basato su hash MongoDB fornisce un indice che non lavora sul valore diretto di in campo, ma su quello generato da una funzione di hash.

4.3 Sharding

Il partizionamento in MongoDB viene sempre eseguito a livello di collection attraverso la **shard key**, un indice definito su tutti i documenti di una collezione.

Figura 51: MongoDB: Sharding

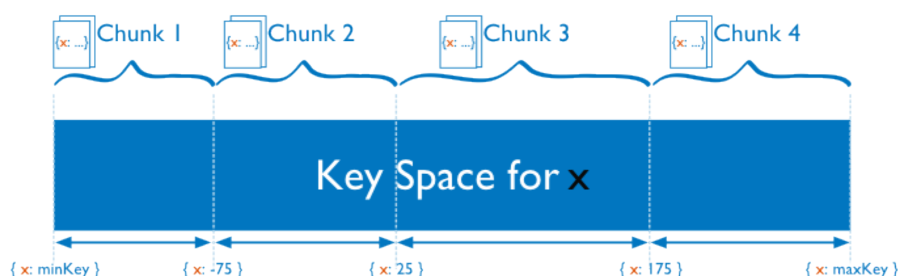


Tutti valori della shart key vengono divisi in sotto-insiemi detti **chunks** che saranno l'unità di base distribuite su un rete di nodi server, detti **shard**. La strategia

con cui vengono generati i vari chunks può essere basta su intervalli o su una funzione hash.

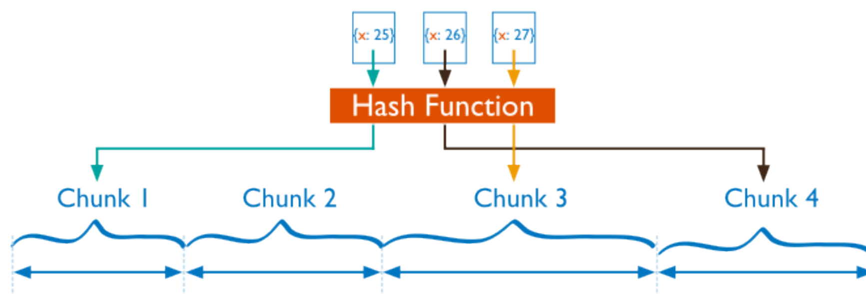
Range Based Sharding in questa strategia si considera l'intero range di possibili valori della shard key e li si suddivide in intervalli contigui che non si intersecano tra loro. Ogni valore della shar key viene associato al blocchi in cui cade generando così il chunk.

Figura 52: MongoDB: Range Based Sharding



Hash Based Sharding Si utilizza una funzione di hash sul valore della shard key e sarà tale valore ad essere utilizzato nel trovare il blocco contiguo in cui cade.

Figura 53: MongoDB: Hash Based Sharding

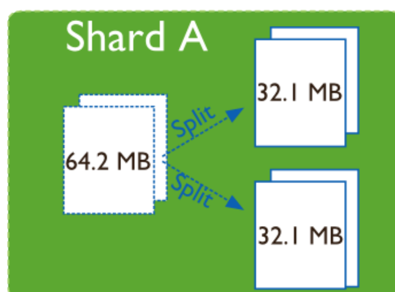


Il partizionamento basato su range ottimizza l'esecuzione di query su intervalli perché è necessario contattare solo i nodi il cui intervallo di valori interseca quello desiderato. Il risultato è una distribuzione non uniforme che potrebbe generare colli di bottiglia e più un generale un cattivo utilizzo delle risorse disponibili. Ad esempio se il valore del campo indicizzato è incrementale, un partizionamento basato su range porterebbe tutte le richieste verso i nodi designati a gestire i valori più bassi.

In questi casi l'utilizzo di una funzione hash potrebbe garantire una distribuzione più uniforme dei documenti su tutti i nodi.

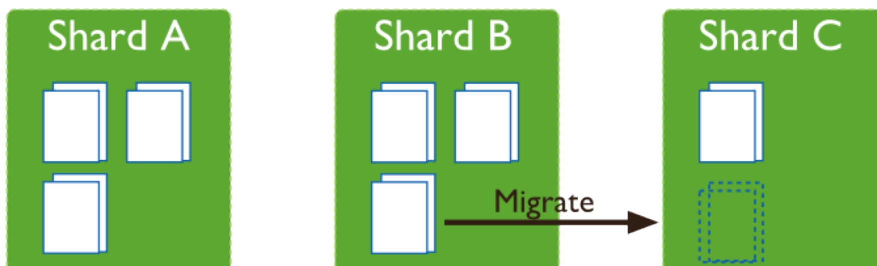
I chunks hanno una massima dimensione prefissata in fase di configurazione. Nell'utilizzo da parte di una applicazione può accadere che un blocco di documenti cresca e raggiunga tale dimensione. In tal caso Mongo esegue una operazione di **splitting** che divide il chunk in due di eguale dimensione e notificherà al cluster la nuova configurazione senza intervenire sui dati in maniera diretta che continueranno ad essere gestiti dal nodo stesso.

Figura 54: MongoDB: Splitting



Poiché non splitting non esegue migrazione un aumento del numero di chunk su uno shard potrebbe comunque portare il sistema ad essere asimmetrico. La fase di bilanciamento del carico cerca di mantenere eguale il numero di chunk gestiti da ogni shard eseguendo una fase di migrazione in background attraverso il processo **balancer**. Durante una migrazione tutti i documenti di un blocco vendo trasferiti dallo shard di origine a quello destinazione. Per garantire una continuità sulla disponiblità del sistema l'**origin shard** continuerà a rispondere alle richieste sui dati interessati e manderà le modifiche alla destinazione che le applicherà in modo da avere i dati perfettamente aggiornati non appena la migrazione termina e e sarà il **destination shard** a rispondere alle richieste.

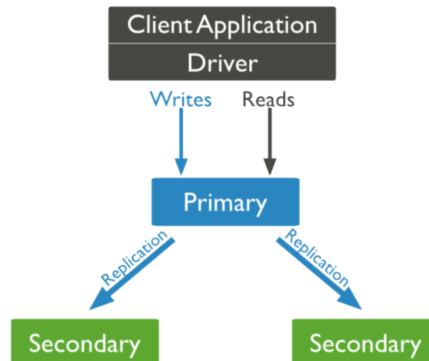
Figura 55: MongoDB:Shard Migration



4.4 Replication

I nodi shard atti a mantenere le porzioni del dataset secondo la strategia di partizionamento possono essere resi più affidabili attraverso la replicazione nota come **replica set**: uno schema di replicazione a backup-up primario

Figura 56: MongoDB: Replication



Le operazioni di write possono essere eseguite esclusivamente dal primario che li scriverià sia su commit log e sia sulle proprie tabelle. I nodi secondari ricevono i commit log e li utilizzano per aggiornare i proprio dati. Per le operazioni di read, quando il client ha necessità di avere sempre la versione più recente di un dato, contatta direttamente il nodo primario. Questo è anche il comportamento di default, ma in contesti dove la riduzione della latenza nelle risposte e più in generale il volume di richieste ha priorità maggiore rispetto alla consistenza delle letture, i client possono distribuire le richieste di read anche sugli shard secondari. I driver di MongoDB supportano una modalità di lettura a preferenze dove la lsita dei primari viene manenuta in locale. È possibile effetturare richieste ai secondari a livello di connessione, database, collection o per singola operazione in modo da offrire la massima flessibilità all'applicazione client.

Se il primario non è più raggiungere gli shard secondari iniziano una fase di elezione per il nuovo primario. Al fine di avere sempre un numero di votanti dispari è possibile definire dei processi arbiter il cui compito è di intervenire esclusivamente nella fase di elezione.

4.5 Consistency

Grazie alla usa natura a documenti, una qualunque modifica interna ad un oggetto, anche a livello di sotto documenti embedded, è sempre atomica. L'accesso concorrenziale è garantito dal fatto che le operazioni di write sono eseguite solo dal nodo primario di un cluster shard. Poiché MongoDB non gestisce alcun meccanismo di controllo di versione (MVVC), le modifiche che richiedono l'intervento su più documenti non possono essere isolate da un blocco transazionale.

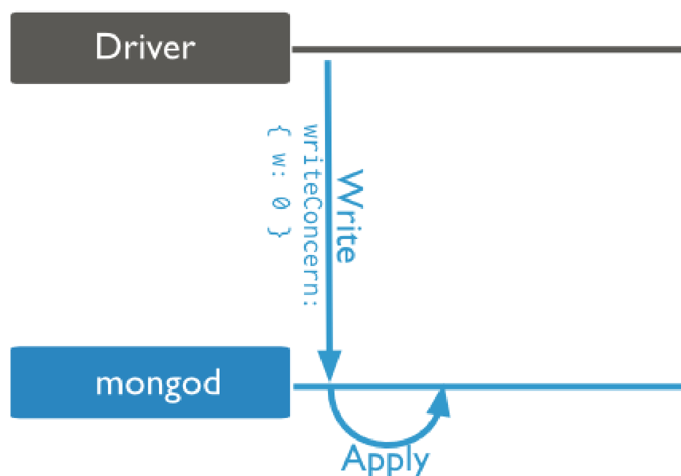
Nell'eseguire una scrittura il client può decidere il livello di robustezza noto come **write concern**. Il più robusto livello di concern assicura la persistenza del

risultato di una operazione, sia di insert, update o delete, quando se ne notifica il suo completamento al client. Per avere tempi di ripsosta migliori è possibile rinunciare a questa proprietà ed avere un livello più debole in cui, in alcuni scenari di fallimento alcune operazioni notificate con successo potrebbero, in realtà, andare perdute (assenza di durabilità delle operazioni).

Errors Ignored il livello più basso in cui nessuna notifica viene ricevuta dal client. Qualunque fallimento della rete o dello storage stesso viene ignorato.

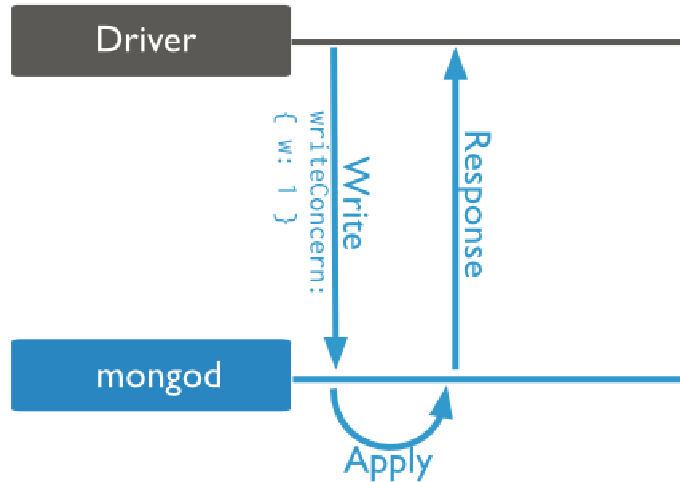
Unacknowledged assenza di riscontro da parte del client in maniera simile al livello ignored, con la differenza che il driver cattura errori della rete.

Figura 57: MongoDB: Unacknowledged



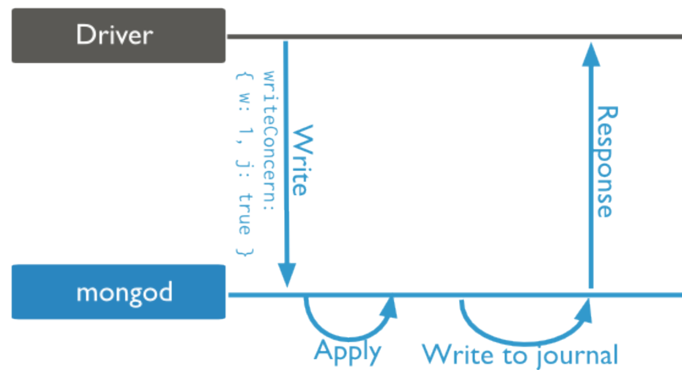
Acknowledged il client attende il riscrontro di avvenuta scrittura senza errori sia sulla rete che sullo storage (chiave duplicate o altri errori simili). È il livello di concern utilizzato per default.

Figura 58: MongoDB: Acknowledged



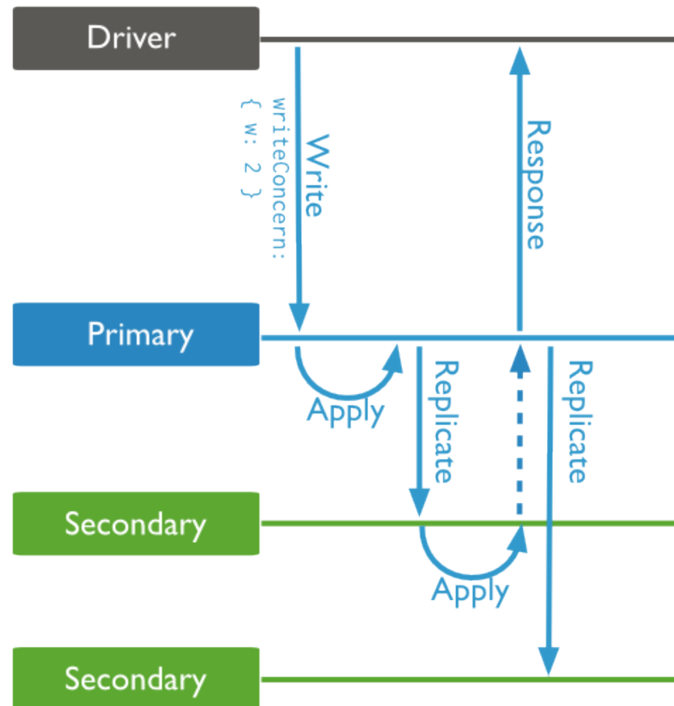
Journaled il server shard invia la notifica di successo solo dopo aver scritto l'operazione anche sul commit log journal. Questo livello permette il recupero delle informazioni anche ad a seguito di shutdown.

Figura 59: MongoDB: Journaled



Replica Acknowledged livello che garantisce che un'operazione di scrittura sia propagata anche sulle repliche.

Figura 60: MongoDB: Replica Acknowledged



4.6 Architecture

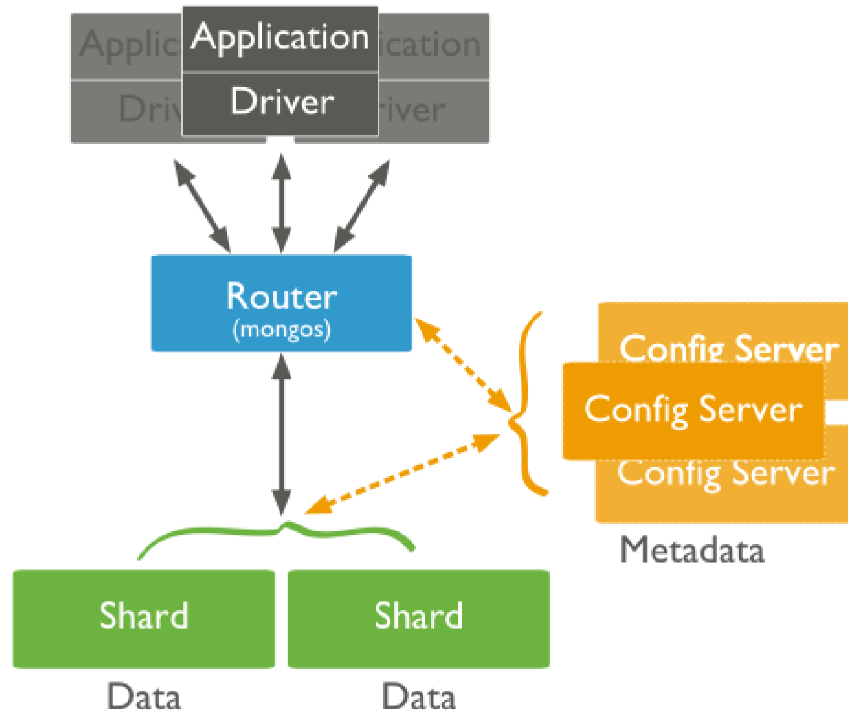
L'architettura di mongo prevede l'utilizzo di tra grandi componenti: shards, query router e config server.

Shards sono i nodi che memorizzano effettivamente i dati sotto forma di chunks. Per garantire l'affidabilità e la disponibilità tipicamente gli shards sono configurati sotto forma di cluster con replicazione interna.

Query Router sono istanze mongo che si interfacciano con le applicazioni client e direzionano le operazioni agli shard opportuni. Per garantire un bilanciamento nell'esecuzione delle arichieste un cluster potrebbe avere più query router per interfacciarsi ai client.

Config Server nodi atti a memorizzare le meta informazioni in particolare la tengono traccia di quale chunks è responsabile ogni shard presente nella rete. Per manetnere un'elevata consistenza i dati sono salvati attraverso il procollo di TwoPhase Commit.

Figura 61: MongoDB: Architecture



Poiché ognuno di questi componenti è in realtà un processo, è possibile eseguirli in maniera libera sulle macchine fisiche della rete. Ad esempio eseguendo più shard su una sola macchina è possibile gestire le differenti risorse hardware.

In questa architettura la replicazione è resta trasparente poiché ogni cluster di shard viene visto dalla rete come un unico blocco responsabile di alcuni chunk.

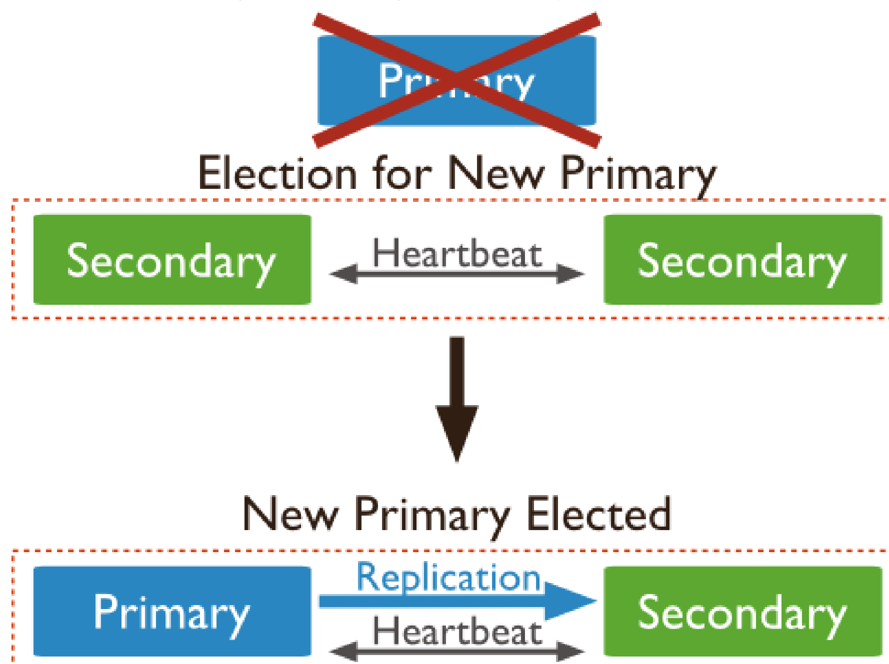
4.7 Membership e Failure

I vari componenti dell'architettura Mongo sono dei processi per cui la gestione della membership ne risulta agevole. I nodi aggiunti al replica set devono essere vuoti in quanto, in assenza di un controllo basato su timestamp, non sarebbe possibile individuare i singoli dati non aggiornati. L'aggiunta di un nuovo cluster shard nella rete viene automatizzata dal processo di load balancing automatico che prevede la migrazione di chunk in presenza di sbilanciamenti nella loro distribuzione.

Mongo ha diversi comportamenti in base al tipo di fallimento che è occorso al sistema. Il fallimento di un processo query server non è critico per il sistema in termini di disponilbiltà perché le richieste del client possono essere inoltrate su altri processi. Inoltre, essendo un semplice componente di routing senza responabilità sullo stato della rete, un riavvio del processo è sufficiente per risolvere il crash. Anche il fallimento di una replica di un cluster shard non incide sulla disponibilità perché le operazioni di write sono sempre affidate al primario. Se è il processo

primario a cadere, invece, rimane solo temporaneamente non raggiungibile finché un nuovo primario non viene eletto.

Figura 62: MongoDB: Primary Set Failure



Se il fallimento coinvolge l'intero cluster di shard tutti i chunk di cui erano responsabili non sono più utilizzabili nelle richieste del client, ma il sistema nel suo complesso resta raggiungibile perché è in grado di rispondere alle operazioni sugli altri chunk. Un fallimento anche su un solo nodo dei config server rende il sistema impossilbilitato a gestire split e migrazione dei chunk ma resta comunque attivo per la gestione delle operazioni.

4.8 Implementation

MongoDB è stato scritto completamente in C++ è consiste nell'implementazione di due tipi di servizi. **mongod** che implementa la gestione dei database e **mongos** per la gestione del routing e autoshading.

4.9 Conclusioni

Nonostante sia un database NoSQL relativamente recente, MongoDB si è velocemente imposto come uno dei più conosciuti ed utilizzati. Il suo modello di dati è più in generale la filosofia nel garantire elevate prestazioni e buona affidabilità senza gravare i compiti più difficili all'applicazione client.

Le elevate prestazioni sono garantire dal modello di dati che permette di accorpare molte informazioni sotto forma di documenti atomici sui quali applicare indici secondari. Il modello a documenti, sotto questo punto di vista, può essere visto come una forma di partizionamento implicito delle informazioni. Questo schema si adatta bene a molti contesti ma ha il difetto di rendere il sistema poco flessibile nel tempo e al cambiamento della struttura dei dati che ne può conseguire. La disponibilità è offerta dal modello a replicazione con gestione automatizzata dei fallimenti in cui è anche possibile decidere il livello di consistenza desiderato. Infine il partizionamento è resto efficiente dalla gestione automatizzata dei blocchi di documenti che vengono spittati e spostati nella rete con processi in background.

A livello di CAP mongo permette quindi di lasciar scegliere al client un comportamento consistente (CP) o più disponibile (AP).

Problema	Tecnica	Vantaggi
Partizionamento	• chunk unit block • based/hashed partitioning	• possibile ottimizzazione di range query solo se richiesto • Gestione automatizzata del bilanciamento del carico
Consistenza e Isolamento	• Write concern	• Livello di consistenza adattabile lato client
Gestione fallimenti temporanei	• Replica set • Primary election	• Pochi punti di fallimento di tutto il sistema
Gestione fallimenti permanenti	• Replica set • Primary election	• Pochi punti di fallimento di tutto il sistema
Membership	• Aggiunta di processi mongod	• Facile

Vantaggi:

- Doppia licenza.
- Supporto ufficiale per molti linguaggi.
- Modello a documenti facile da usare.

- Modello query ad oggetti.
- Indicizzazione campi e sotto-documenti.
- Operazioni su documenti atomiche.
- Livello di consistenza adattabile lato client.
- Partizionamento automatico.
- Ottimizzazione range query opzionale.
- Assenza colli di bottiglia.
- gestione fallimenti automatica.
- Membership facile via processi.

Svantaggi:

- Modello a documenti poco flessibile a grandi cambiamenti strutturali.
- Operazioni di write solo su un singolo nodo.
- Nessuna gestione delle transazioni.
- nessun supporto al MCCV.

5 BigMemory

Sviluppato da Terracotta[3] BigMemory è un software per gestire grandi quantità di dati in maniera veloce attraverso una gestione in memoria dei dati costruita su un cluster di macchine che garantisce affidabilità.

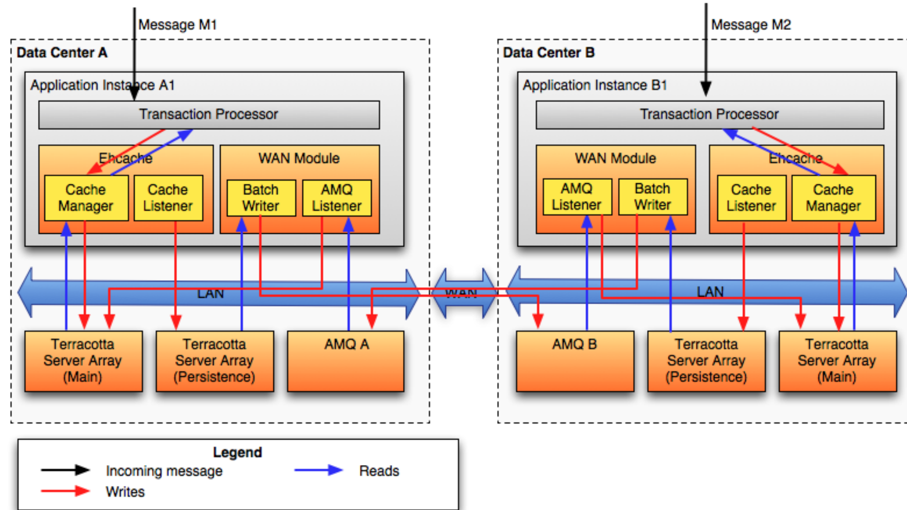
- una gestione di una qualità potenzialmente illimitata di dati tenuti sulle RAM di una batteria di server in cluster.
- una gestione della memoria off-heap che evita il degrado delle prestazioni dovuto al garbage collector della JVM.
- un'interfaccia d'utilizzo compatibile con il software open-source Ehcache.
- una garanzia della consistenza dei dati di livello configurabile.
- un supporto nativo per vari per piattaforme client Java, C# e C++.
- un supporto alla ricerca ottimizzata attraverso l'utilizzo di indici configurabili.
- un supporto all'utilizzo di per dispositivi di storage SSD da abbinare all'uso della RAM.
- un supporto alle interrogazioni dei dati attraverso un linguaggio SQL-Like.
- un supporto transazionale locale e distribuitito.
- un monitoraggio avanzato dei dati attraverso. un'interfaccia WEB.

Terracotta ha sviluppato quello che è da molti considerato uno dei prodotti opensource più utilizzati in ambito di cache per applicazioni Java: **Ehcache**. BigMemory si appoggia alle API di Ehcache per offrire accesso ai dati sul cluster, ma in generarel è un pacchetto software indipendente in grando di incrementare le prestazioni levando dal database il carico delle richieste più frequenti semplificando la scalabilità. È largamente usata nelle applicazioni Java perché è robusta, collaudata e completa nelle caratteristiche. Ehcache riesce a scalare dal singolo processo a situazioni miste per configurazioni di terabyte di dati in cache sia nel processo stesso che fuori da esso.

La figura in seguito mostra una possibile integrazione di ehcache all'interno dell'architettura cluster di Tterracotta.

[3] http://www.terracotta.org/

Figura 63: Terracotta: Ehcache



L'azienda Terracotta continua a sviluppare e mantenere Ehcache come progetto open source sotto licenza Apache 2.

Le classi chiavi di Ehcache sono tre.

CacheManager gestisce la creazione e la rimozione delle singole istanze cache. La creazione può essere effettua in maniera singleton o meno. Dalla versione 2.5 in su la creazione è vincolata dal nome univoco all'interno di una JVM.

Ehcache Interfaccia delle cache gestite dal manager. Ogni cache ragiona in termini di nome e attributi e contiene elementi salvati in memoria o, se configurato, su disco.

Element lentry atomica delle cache. Raggruppa la chiave e il valore da memorizzare più un record degli accessi. Sono gli elementi ad essere inseriti, rimossi o invalidati in base ai parametri della cache che li sta usando.

La API di Ehcache può essere utilizzata nelle seguenti topologie:

Standalone il data set è mantenuto nel nodo dell'applicazione in maniera indipendente dalle altre. L'assenza di comunicazione tra le varie istanze di Ehcache causa una consistenza debole dei dati. In ogni nodo è possibile aggiungere tutte le caratteristiche di BigMemory, compresa la gestione offheap.

Distributed i dati sono mantenuti in un cluster Terracotta Server Array con un sotto insieme dei dati usati di recente mantenuti nelle instanze di Ehcache sui singoli nodi delle applicazioni. Il supporto di BigMemory permette di aggiungere anche la consistenza forte.

Replicated i dati presenti nelle istanze di Ehcache presenti in ogni nodo sono copiati o invalidati senza l'utilizzo di lock. La replicazione può essere sincrona o meno bloccando le scritture. L'unica consistenza possibile èq uella debole.

Le applicazioni che girano in cluster possono soffrire di alcuni comportamenti indesiderati.

Cache Drift le modifiche su una cache non appaiono alle altre istanze. Ehcache in configurazione distribuita o replicata mantiene la sincronia.

Database Bottleneck In un ambiente distrivuita è necessario mantenere aggiornata la cache rispetto ai valori del database. L'overheader dovuto alle fase di refresh potrebbe portare a colli di bottiglia sul database. La configurazione distribuita limita questo fenomeno.

La cache distribuita usa il Terracotta Server Array, disponibile con BigMemory Max, per abilitare la condivisione dei dati su diverse JVM. Combinando le potenzialità del Terracotta Server Array con Ehcache è possibile avere:

Scale-out scalabilità lineare delle applicazioni.

Reliability affidabilità sui dati mantenuti consistenti sul cluster.

Throughput elevate prestazioni dovute alla gestione InMemory.

Powerful API accesso a delle API più potenti.

Per tale ragioni l'uso della cache distribuita è raccomandato per applicazioni che richiedono scalabiltà orizzontale.

L'affidabilità dei dati ottenuta attraverso meccanismi di replicazione basati su RMI, JGroups, JMS o Cache Server. Quest'ultimo è supportato dalla versione 1.5 e permette ad ogni JVM di scrivere e leggere dati comunicando con il Cache Server che ha a sua volta una JVM propria.

La consistenza della cache può avere forti impatti sulla prestazioni globali. Per tale ragione Ehcache può essere sincrona o meno in base alle esigenze dell'applicazione. Il meccanismo principale per garantire la consistenza avviene tramite una effettiva copia di un valore in una instanza della cache su tutte le altre generando una latenza nella risposta. Ehcache supporta anche la replicazione per invalidazione. A seguito di una modifica di un dato, viene inoltrata una notifica di invalidazione sulla chiave associata per forzare le altre istanze ad aggiornare i loro valori in maniera autonoma. Questo approccio garantisce tempi della singola risposta più brevi ma potrebbe ridurre le prestazioni globali perché l'operazione di refresh richiede l'accesso al database.

La strategia di memorizzazione InMemory è sempre abilitata e non è possibile manipolarla direttamente, ma è un componente di ogni cache. Questa è la strategia di memorizzazione più veloce in quanto elemento è sempre memorizzato nella

RAM della macchina con sincronizzazione rispetto all'accesso concorrente di diversi Thread Java.

Dalla configurazione è possibile specificare la dimensione della cache InMemory attraverso il numero di elementi massimo che può essere indicizzato. Quando viene raggiunta la dimensione massima altri gli elementi già presenti vengono cancellati se non è abilitata l'opzione di overflow. La strategia con cui si selezionano gli elementi da cancellare può essere una tra tre tipologie.

Least Recently Used (LRU) viene cancellato l'elemento meno recente in base al suo timestamp che viene aggiornato ad ogni accesso o inserimento di dati.

Least Frequently Used (LFU) viene cancellato l'elemento meno usato in base a un contantore di accessi.

First In First Out (FIFO) Elementi sono cancellati nello stesso ordine in cui vengono inseriti.

Per tutte le tipologie di ordinamento è sempre possibile, attraverso API, eseguire un accesso o inserimento senza aggiornamento del timestamp.

L'integrazione con BigMemory permette ad Ehcache di usare una tipologia di memoria addizionale fuori dallo spazio standard gestito dalla JVM. Questo spazio off-heap non essendo soggetto al garbage collector permette la creazione di cache di dimensione estreme senza degrado prestazionale. Poiché questo spazio di memoria è gestito a byte grezzi posson oessere memorizzati chiavi e valori di oggetti che implementano l'interfaccia standard Serializable. Di conseguenza ogni accesso alla cache prevede una fase di serializzazione e deserializzazione.

Oltre all'utilizzo in memoria, sia essa standard oppure off-heap, è possibile imposse l'utilizzo anche dello spazio sul disco per garantire persistenza della cache anche a riavvi della macchina. In maniera analoga allo spazio di memoria gestito da BigMemory anche la persistenza su disco richiede la serializzabilità delle chiavi e dei valori memorizzati. Inoltre gli accessi sul disco sono sincronizzati rispetto all'utilizzo concorrente.

5.0.1 BigMemory Go & Max

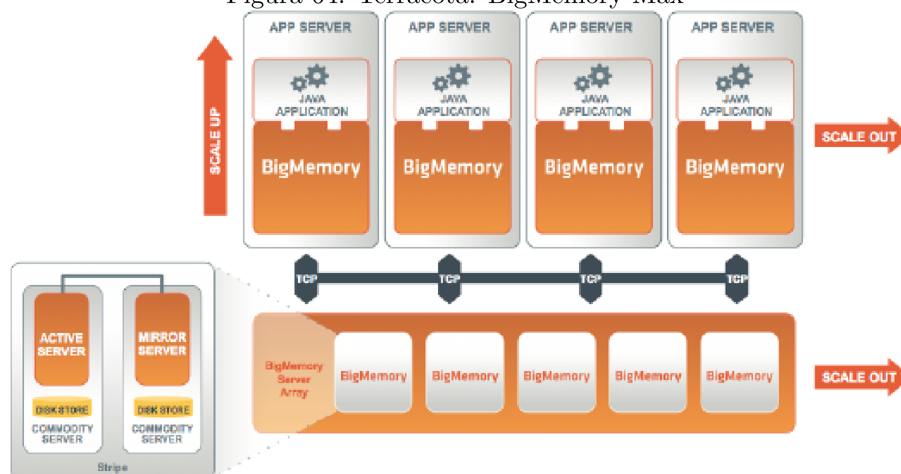
BigMemory permette alle applicazioni Java di poter utilizzare tutta la memoria che richiedono limitata solo dalla quantità fisicamente istallata sulla macchina. In altre parole le caratteristiche principali di **BigMemory Go** sono le seguenti.

- Gestione potenzialmente illimitata di dati InMemory senza il Java GC.
- Monitoraggio e ricerca avanzate sui dati.
- Locking sui dati e operazioni transazionali.

L'obiettivo principare di BigMemory è quello di garantire lo Scale-up, una scalabilità verticale delle prestazioni in base alle risorse fisiche su una singola macchina.

BigMemory Max è la versione più completa del pacchetto software sviluppato da Terracotta che aggiunge la gestione di un cluster di un numero indefinito di macchine sotto forma di cluster ad elevate prestazioni. Se BigMemory Go, grazie alla gestione off-heap della memoria. permetteva lo scale-up, la versione Max offre anche lo scale-out, ossia una scalabilità orizzontale che incrementa all'aumentare delle macchine installate.

Figura 64: Terracota: BigMemory Max



A livello di utilizzo BigMemory Go/Max può essere visto come una estensione di Ehcache per il supporto allo storage off-heap e per tale ragione ne condivide le stesse chiassi classi chiave al quale aggiunge l'esecuzione di complesse query attraverso degli indici secondari che offrono la possibilità di avere criteri di ricerca alternativi. Questi indici sono definiti **searchable** e possono essere estratti sia dalla chiave che dai valori di un elemento in cache. È anche possibile avere indici per valori aggregati. In ogni caso devono essere trattati come attributi tipati secondo le più comuni classi Java: Boolean, Byte, Character, Double, Float, Integer, Long, Short, String, Date (java.util), Date (java.jsq) ed Enum.

Le quiery sono formulate attraverso il linguaggio proprietario **BigMemory SQL** dialetto dell'SQL standard. Esiste anche una API che permette di costruire query in maniera programmatica direttamente dal codice Java.

5.1 Data Model

Essendo che BigMemory condivide la API di cache a cui si può abilitare la persistenza distribuita, il modello di dati che ne scaturisce è quello di semplice database a chiave valore. Nello standard di Ehcache ogni chiave associa un oggetto di tipo contenitore, detto elemento, che ingloba oltre al valore effettivo da memorizzare anche informazioni quali marcatore temporale e contatore di accessi.

5.2 Query Model

Il valore che elemento racchiude può essere una istanza di qualunque oggetto non volatile Java. Affinché si possa sfruttare al meglio tutte le caratteristiche di BigMemory l'oggetto memorizzato deve essere una istanza dell'interfaccia standard Serializable in modo da poter essere memorizzata in forma binaria nello spazio off-heap e su disco locale.

Le API per accedere ad Ehcache offrono un utilizzo basato su operazioni CRUD o tramite **BigMemory SQL** dialetto dell'SQL standard.

5.3 Sharding

BigMemory nella versione Max utilizza un componente software, Terracotta Server Array, per clusterizzare i dati di ehcache su un gruppo di macchine. Attraverso una interfaccia compatibile con Ehcache. BigMemory permette di smistare le richieste sui vari server in maniera autonoma e trasparente per avere un bilanciamento del carico e una distribuzione dei dati uniforme sul cluster.

5.4 Replication

Il componente software detto Terracotta Server offre un livello software per implementare il cluster sotto le API di BigMemory. L'architettura del cluster è configurabile ed in fase di produzione è possibile abilitare la modalità ad elevata disponibilità che permette al cluster di mantenere il suo funzionamento anche in presenza di fallimenti parziali di alcuni nodi. Per farlo TSA utilizza una replicazione del tipo passiva in cui un server primario, detto attivo, effettua le richieste ed altri server slave, detti mirror, hanno solo il compito di mantenersi sincronizzati al fine di poter sostiuire il principare in caso di crash.

È possibile aggiungere server mirror in fase di configurazione, ma l'uso eccessivo potrebbe generare un overhead non più trascurabile a causa della necessità del cluster di dover mantenere la sincronizzazione.

5.5 Consistency

La consistenza della cache è configurabile attravero la configurazione o via API. È possibile controllare il comportamento delle istanze rispetto al trade-off consistenza e performance attraverso tre modalità.

eventual questa modalità garantisce che i dati in cache abbiauno una consistenza eventuale. Le operazioni di lettura e scrittura risultano velocizzate dal potenziale rischio di avere dati incosistenti per brevi periodi. Questa modalità non può essere modificata programmaticamente via API.

strong questa modalità garantisca che i dati rimagnono consistenti su tutto il cluster in ogni momento. Una get sulla cache restituisce il valore aggiorno dopo che tutte le write di sincronizzazione sono completate. Ogni punt risulta essere in una transazione separata. Il costo dei lock distribuiti attraverso il two-phase commit forte potrebbe incidere sulle prestazioni globali.

bulk load questa modalità è ottimizzata per il caricamento in blocco di dati nella cache senza il preso dei lock. È simile alla modalità eventuale, ma orientata al batch quindi più elevate velocità di scrittura e una consistenza ancora più debole. L'attivazione e disattivazione di questa modalità avviene solo programmaticamente tramite API.

Oltre ai settaggi espliciti anche altre fattori possono incidere sulla consistenza e le prestazion della cache.

Explicit locking questa API fornisce metodi a livello di applicazione che incidono sul cluster a livello globale. Se abbinato con la modalità strong rende ogni operazione racchiusa in un unico blocco transazionale. Se abbinata alla modalità eventuale rende transazionali solo le operazioni all'interno di un blocco di lock.

UnlockedReadsView un opzione per abilitare le letture sporche. Può essere usata solo con istanze di cache in cui è stata abilitata la consistenza forte. Lidea èq uella di incrementare le prestazioni delle letture senza perdere la consistenza forte sulle scritture.

Atomic method per garantire le scritture consistenti ed evitare potenziali race condition sulle possono essere usati due metodi di put e replace che lanciano errore se la usati su una cache in cui è stata abilitata la consistenza eventuale.

Abilitando il lock espliciti è possibile decidere tra vari livelli di funzionamento.

off transazioni disabilitate.

local i cambiamenti sulle varie cache sono effettuati atomicamente solo localmente. Questa modalità è utile quando i dati calcolati dalla istanza corrente non sono relazionati con le altre.

xa i cambiamenti di una cache sono controllati rispetto alle altre attraverso l JTA (Java Transazion API), ma senza il prezzo di un two-phase commit completo. In questa modalità i dati in cache non sono sincronizzati con le altre risorse per cui sono utili quando si accetta la presenza di informazioni non aggiornate per brevi periodi.

xa_strict versione più robusta delle transazioni xa sopra JTA in cui un two-phase commit completo garantisce una sincronizzazione perfetta tra i dati nelle cachevietando che si possa accedere a informazioni non aggiornate.

5.6 Architecture

La gestione del cluster di macchine viene effettuata attraverso un modulo, T**erracotta Server Array,** che fornisce una piattaforma generica anche per altri prodotti di Terracotta queyli **Quartz** e **Web Sessions**.

Le principali caratteristiche sono.

Distributed In-memory Data Management gestisce da 10 a 100 volte i dati in memoria rispetto alle griglie.

Scalability Without Complexity facile configurazione per aggiungere macchine server per incontrare la crescita di richieste e più in generale per pianificare le capacità del sistema.

High Availability gestione dinamica dei fallimenti per garantire continuità del servizio.

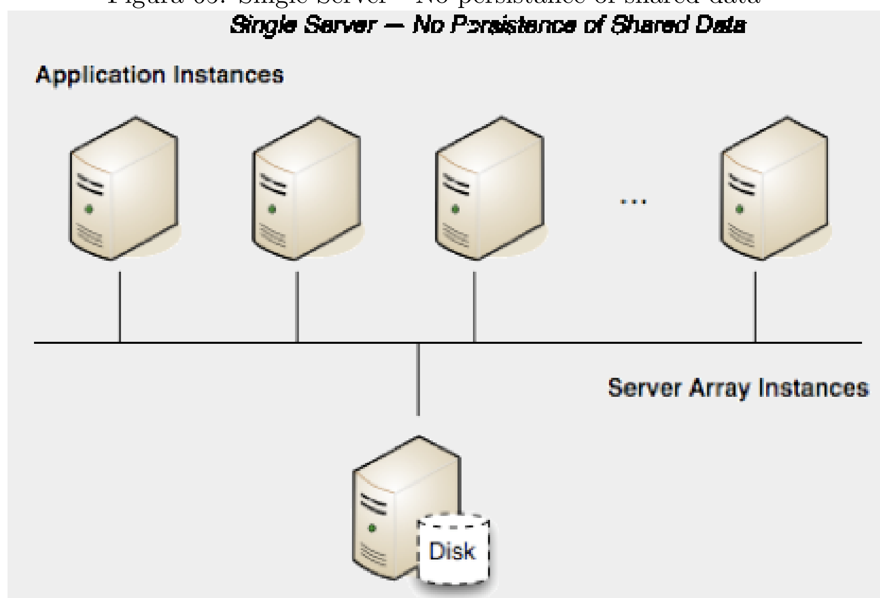
Configurable Health Monitoring meccanismo per il monitoraggio dei singoli nodi attraverso una console WEB based o in maniera programmatica attraverso una API REST.

Persistent Application State creazione di fotografie del sistema attraverso la persistenza automatica di tutti i dati condivisi da tutti i Thread.

Automatic Node Reconnection Gestione automatica di fallimenti e disconnessioni temporanee.

Un server array può essere creato a partire da un minimo di due nodi e può essere installato in diverse architetture adatte sia alla fase di sviluppo che quella di produzione. In un ambiente di sviluppo, la persistenza condivisa dei dati è spesso inutile e anche inconveniente. Eseguire un cluster senza persistenza è una buona soluzione per creare ambienti comodi per lo sviluppo di un software.

Figura 65: Single Server - No persistance of shared data

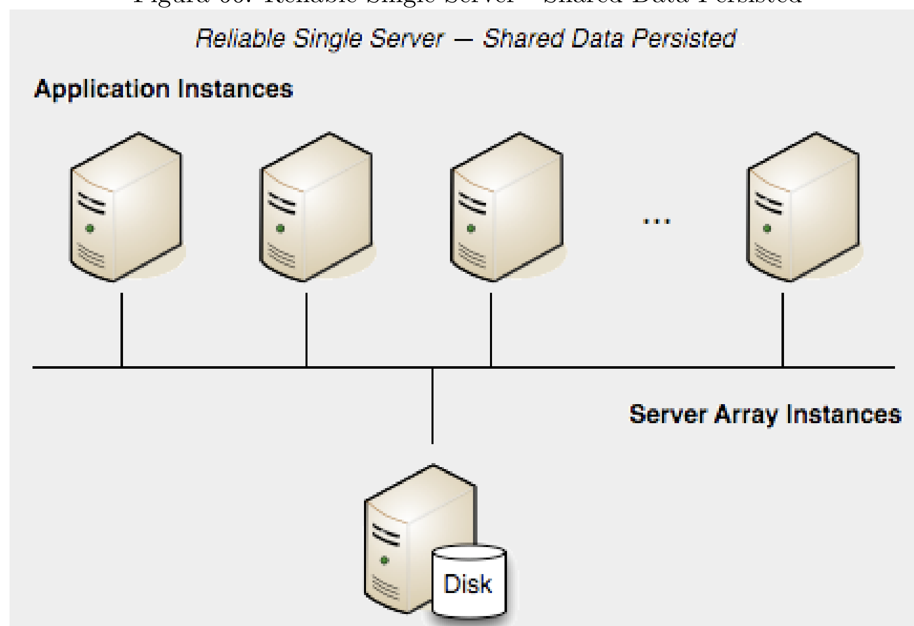


In questa configurazione non è gestita il riavvio delle macchine senza perdita dei dati. Se un server va in crash e fallisce tutti i dati condivisi verranno persi.

Inoltre anche le sessioni con i client vengono perdute e devono nuovamente eseguire l'accesso al cluster.

In una fase di produzione la persistenza risulta essere un requisito fondamentale e questo necessità di dover configurare i server per l'utilizzo dei disco.Terracotta offre persistenza con riavvio veloce (Fast Restartability).

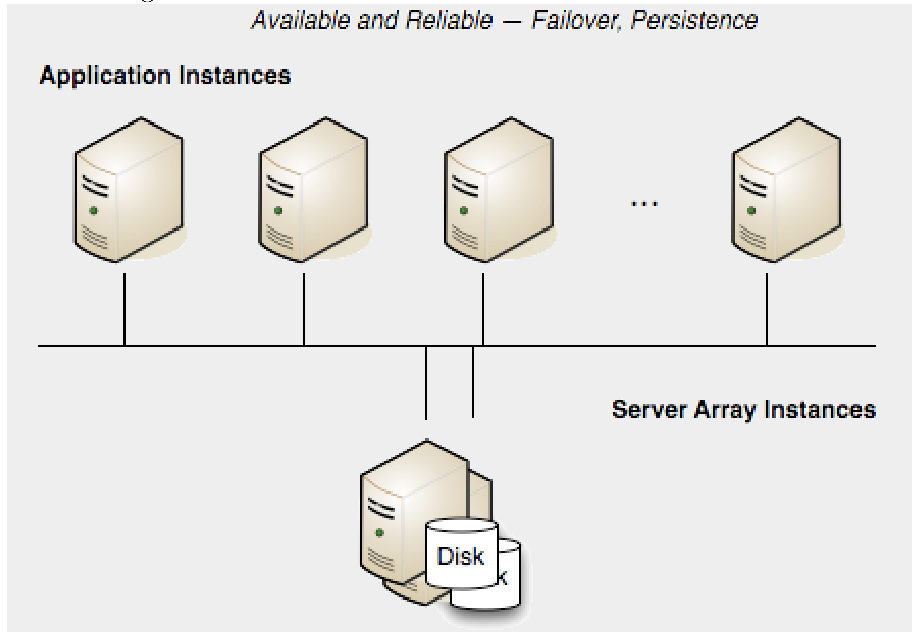
Figura 66: Reliable Single Server - Shared Data Persisted



Questa caratteristica fornisce una capacità di recupero da fallimenti temporanei mantenendo la consistenza completa dei dati in memoria. A seguito di qualunque tipo di stop del sistema, anche di un solo nodo sia esso pianificato o meno, l'intero cluster va in shutdown. Al successivo riavvio l'applicazione troverà tutti i dati di BigMemory Max ancora disponibili. Abilitando il Fast Restart si forza una persistenza in tempo reale di tutti i dato in memoria sul disco locale. Dopo un riavvio i dati (off-heap o meno) vengono ripristinati. In oltre anche le sessioni client precendemente attive vengono ripristinante senza dover rieffettuare l'accesso.

La configurazione con Fast Restart riesce ad essere estremamente affidabile rinunciando alla disponibilità continua del sistema.

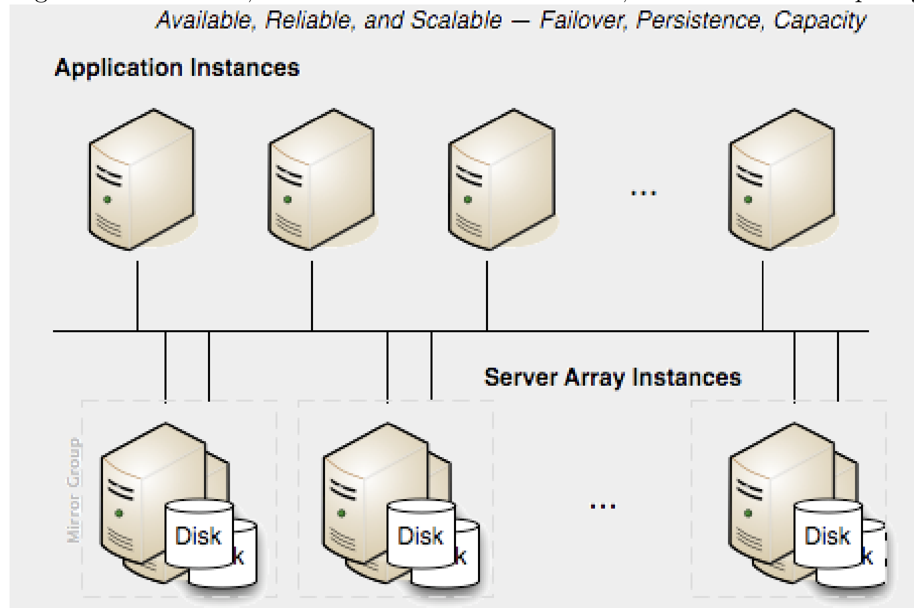
Figura 67: Available and Reliable - Failover and Persistence



Questo perché anche il fallimento di un solo server rende l'intero sistema inaccessibile per le applicazioni. Il limite è dovuto al fatto che manca la ridondanza per fornire la gestione automatica dei fallimenti. Per garantire una continuità è necessario avere dei server mirror che possono sostiuire in qualunque momento quello originale in caso questi risultasse inacessibile.

La configurazione a elevata disponibilità richeide la perfetta sincronizzzazione tra tutti i server del cluster e questo rende il sistema perfettamente robusto ai fallimenti. Finché anche un solo server è attivo lìintero cluster è sicuramente funzionamento, ma il prezzo da pagare il degrado delle prestazioni a causa dei messaggi di sincronizzazione. è Se le richieste di capacità del sistema diventano sempre più onerose è possibile migliorare la scalalabilità del sistema usando gruppi di mirror. Questa configurazione permette loi scale-out, la scalabilità orizzontale del sistema che riesce ad aumentare le prestazioni all'aggiunta di nodi server al cluster.

Figura 68: Available, Reliable and Scalable - Failover, Persistence and Capacity



All'interno di ogni gruppo i server si comportano nella logica master-slave della configurazione ad elevata disponibilità. Di conseguenza ogni server partecipa al protocollo di elezione di un solo nodo attivo che rimarrà tale finché non avviene un fallimento o viene manualmente rimosso dal cluster. Una volta che ogni gruppo ha terminato ha terminato la fase di elezione, tutti i server attivi iniziano a cooperare per gestire il cluster. Se un server attivo fallisce il cluster rimane inaccessibile per il solo periodo di elezione del nuovo master.

5.7 Membership e Failure

L'aggiunta o rimozione di nodi sul cluster di Terracotta Server Array avviene tramite file di configurazione in maniera agevole e facile da aggiornare.

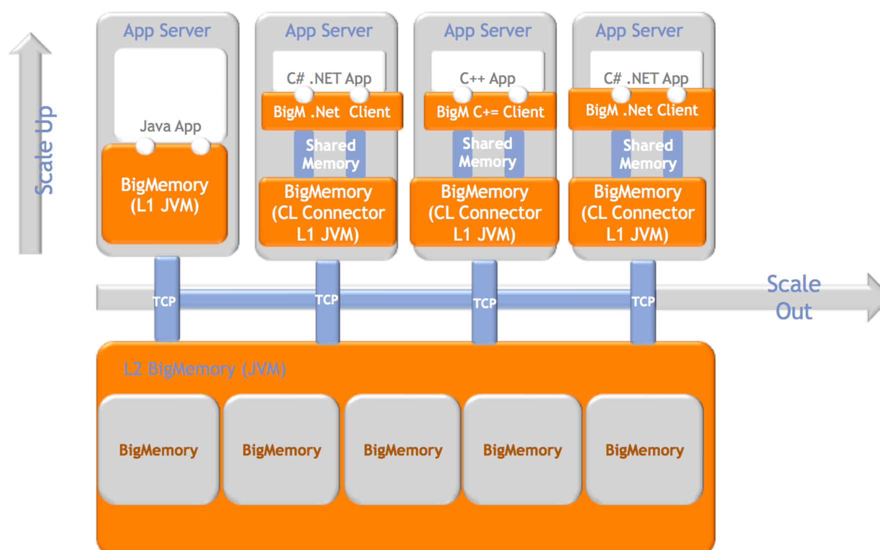
La gestione dei fallimenti avviene solo se è abilitata la replicazione tramite la tecnica a mirror.

La selezione del server master e dei mirror avviene in fase di avvio del cluster attraverso un protocollo di elezione tra le macchine coinvolte. Lo steso protocollo viene utilizzato anche a fronte di un fallimento del server attivo, ma in questo caso solo tra i server che erano perfettamente sincronizzati con il master prima che diventasse irragiungibile. La situazione in cui il server attivo fallisce prima in un momento in cui nessun mirror aveva completato la sincronizzazione è l'unico scenario in cui un singolo fallimento causa una inaccessiblità del'intero cluster. Terracotta permette alle istanze mirror di poter essere eseguite con persistenza o meno. Se un nodo attivo va in crash e viene eletta un'altra macchina lo stato del sistema salvato sul disco locale della macchina fallita risulterà obsoleto e quindi, prima che essa possa riagganciarsi al cluster i dati verrano cancellati e aggiornati. Se Tutti i

server con persistenza dovessero fallire e tornare attivi il server che in precedenza era attivo sarà automaticamente eletto per evitare perdita di dati. In una ipotetica situazione di in cui un problema di rete disconnette i server mirror da quelli attivi senza crash del sistema, si potrebbe giungere ad una situazione in cui due master sono presenti nel cluster che viene definita Split Brain. Quando i nodi torneranno a comunicare un meccanismo automatico di Terracotta permette di rportare ad uno stato ordinario il cluster senza perdita di dati condivisi. Anche se in questi casi è raccomandato confermare l'integrità.

5.8 Implementation

Il codice di BigMemory è scritto interamente in Java e compatibile sia con JVM versione 6 che 7. Poiché il software vuole essere facilmente integrabile in qualunque ambiente sono presenti dei moduli software ufficialmente supportati fa Terracotta. Oltre al client Java è presente uno per applicazione C #e C++. I client sono tra loro interoperabili in modo che una applicazione scritta in C# è in grado di comunicare con una scritta in C++ o JAVA senza alcun costo di integrazione da parte dello sviluppatore. Questo perché è connettore ad agganciarsi al clister di BigMemory.



Esiste anche una console con interfaccia WEB per la gestione dei vari server che compongono il cluster.

5.9 Conclusioni

Nonostante non sia esplicitamente un database NoSQL ad oggi BigMemory risulta una delle soluzioni più usate in ambienti enterprice. Il punto di forza è certamente la gestione dei dati InMemory su uno spazio riservato senza l'intervento del Garbage Collector che permette una scalabilità orizzontale potenzilamente infinita. Sul fronte della gestione distribuita il modulo TSA offre un livello software facilmente usabile con un buon livello di in grado di confiugrazione per adattarsi alle singole esigenze nei limiti imposti dal Teorema CAP. I client ufficiali offrono anche una bfacile integrazione tra ambienti Java, C++ e C#.

Lo svatanggio è l'assenza delle gestione tipiche dei datbase NoSQL puri. Il softare non offre una gestione eplicita dello sharding che viene completamente automatizzato in maniera trasparente.

Le versioni open source sono state deprecate.

Problema	Tecnica	Vantaggi
Partizionamento	Consistent Hash	Massima distribuzione e trasparenza
Consistenza e Isolamento	Configurabile	Livello di consistenza adattabile in fase di configurazione
Gestione fallimenti temporanei	Server Mirror	Elevata disponibiltà
Gestione fallimenti permanenti	Elezione nuovo server attivo	Forte robustezza
Membership	Configurazione	Facile

Vantaggi:

- Software ampiamente usato e testato in casi reali.
- Gestione dati in memoria off-heap.
- Supporto ufficiale per vari linguaggi.
- Ricerca per chiavi o dialetto SQL.
- Indicizzazione campi e sotto-documenti.
- Supporto lock espliciti e transazioni.
- Livello di consistenza configurabile.
- Partizionamento automatico.
- Disponibiltà configurabile tramite mirroring.
- Assenza colli di bottiglia.
- Gestione fallimenti automatica.
- Membership facile via configurazione.

Svantaggi:

- Versione open source deprecata.
- Assenza di un API NoSQL esplicita.
- Modello di dati basato su cache.
- Consistenza non tunable a runtime dal client.
- Partizionamento non descritto e non configurabile.
- Assenza del supporto a multicast per la membership.

6 Hazelcast

Hazelcast è un software per la creazione di cluster per l'esecuzione di codice e la memorizzazione di dati usando tecniche InMemory off-heap per garantire elevatissime performance grazie all'assenza del Garbage Collection della JVM.

L'obiettivo di Hazelcast è di offrire un framework in grado di garantire le seguenti caratteristiche.

- Scalabilità orizzontale potenzialmente infininta tramite la gestione off-heap
- scalabilità verticale attraverso la gestione di una griglia di server
- distribuzione di dati su una griglia di server
- partizionamento dei dati su una griglia di server.
- bilanciamento del carico automatico.
- facilità di utilizzo con configurazione tramite file o API.
- massima integrazione con le interfacce standard Java.
- framework leggero e senza dipendenze.
- supporto transazionale.
- gestione automatica della topologia.
- gestione automatica dei fallimenti.
- database NOSQL di tipo KV integrato.
- casching distribuita integrata.
- buona visione open source.
- versione enterprice relativamente economica.

6.1 Data Model

Hazelcast implementa una gestione InMemory e distribuita dei dati di tipo chiave valore, ma con approccio diverso rispetto ai classico database NoSQL. Ad una chiave viene associata una struttura dati in dipendenza dell quale il client può eseguire operazioni diverse. Tutte tipologie di strutture dati sono delle estensioni delle interfacce standard di Java, quali Map, Set o List, che permettono uno sviluppo familiare di applicazioni Java.

Il valore associato ad ogni chiave può essere un qualunque oggetto serializzabile, Di conseguenza l'API di base per ogni chiave permette operazioni di tipo CRUD sulle istanze delle varie strutture dati, mentre all'interno di esse è possibile eseguire operazioni più complesse che cambiano in dipendenza della struttura stessa.

Hazelcast ottimizza il cluster in base alla tipologia di struttura associata ad una chiave. La struttura comunemente utilizzata è quella delle mappe che vengono

implementate automaticamente in maniera distribuita (**Distributed HashTable - DHT**). Abilitando la persistenza sulle mappe distribuite si implementa un Database NoSQL di tipo chiave valore.

I dati memorizzati all'interno di una tabella vengono trattati come oggetti Java che implementano l'interfaccia Serializable. È tipico memorizzare oggetti nella convezione POJO in cui ogni proprietà ha i metodi get e set per l'accesso e modifica.

6.2 Query Model

Se all'interno del cluster è stata istanziata una tabella è possibile eseguire su di essa le operazioni tipiche dei metodi standard offerti da Java, ma le implemetazioni offerte da Hazelcast offrono funzioni aggiuntive. In particolare è possibile eseguire delle ricerche per mezzo di query scritte in un sotto insieme del linguaggio stadard SQL o programmaticamente per mezzo delle API proprietarie o quelle che implementano lo standard JPA.

Nell'esecuzione delle query Hazelcast deve ispezionare direttamente la proprietà per ogni oggetto impattando le performance. Al fine di ottimizzare i tempi di esecuzione in maniera analoga ai database è possibile definire degli indici sugli oggetti di una mappa distribuita. È possibile farlo sia da file di configurazione che via programmatica da API. Per ogni indice è anche possibile definire un ordinamento utile per le query range.

Dalla versione le 3.0 le DHT supportano strumenti avanzati.

Continous query è possibile mettersi in ascolto degli eventi di un sotto insieme di oggetti definito da un predicato.

Entry processor è possibile definire del codice custom da eseguire su ogni singola entry della mappa durante l'esecuzione di una query in modo da poter avere codice atomico senza utilizzo di lock espliciti.

Interceptors è possibile intercettare le operazioni standard get put e remove della tabella inserendo del codice custom.

Statistics nella implementazione standard di una mappa Java è necessario definire una istanza di entry che in Hazelcast aggiunge meta informazioni ad un oggetto memorizzato quali marcatori temporali o contatori di accessi.

6.3 Sharding

Hazlecast nasce per essere un cluster completamente peer-to-peer e per il partizionamento dei dati di una tabella distribuita utilizza la tecnica del consistent hashing. Questa tecnica permette di eseguire load balancing minimizzando il numero di chiavi e quindi oggetti, da trasferire tra i nodi al fine di mantenere uniforma la distribuzione. Ogni oggetto aggiunto attraverso la funzione di put viene assegnato univocamente ad un solo server del cluster che sarà responsabile di quella chiave. Questo assegnamento avviene ricavando un valore numerico dalla chaive (usando hashCode di

java ad esempio) e individuando il range numerico in cui esso ricade. Partizionando l'insieme di tutti possibili valori in intervalli e assegnando ognuno di essi ai nodi del cluster è possibile ricavare il server proprietario di una oggetto in maniera univoca partendo dalla chiave. Il numero di partizioni è deciso in fase di configurazione e, per default, risulta essere 271. Al fine di poter poter garantire un partizionamento che tenga conto anche delle diverse risorse hardware che ogni macchina fisica può avere è sufficiente lanciare diverse istanze di Hazelcast sulle macchine più potenti.

La tecnica delle hashing consistente è stata introdotta da Amazon è diventata uno standard di fatto per tutte le applicazioni distribuite orientate al modello chiave valore perché permette una elevata scalabilità senza che il numero di nodi pesi sull'architettura grazie alla natura totalmente decentralizzata.

6.4 Replication

la tecnica del consistent hashing permette la scalabilità orizzontale, ma non garantisce nulla in termini di affidabilità e continuità del servizio. A tale scopo anche Hazelcast introduce un concetto di replica il cui numero è configurabile. È importante che il cluster formato da Hazelcast non permette che un nodo possa gestire più di un partizionatomento sia esso proprietario che replica. Ma poiché è possibile lanciare diverse istanze JVM su una stessa macchina è necessario inserire un gruppo all'interno della configurazione che indica al cluster quali indirizzi appartengono ad uno stesso host. Il protocollo interno eviterà di assegnare uno stesso intervallo di partizionamento alla stessa macchina fisica. Per default le mappe distribuite hanno una replica di ogni entry.

Esistono dei casi in cui si evince la necessità di avere cluster multiple e sincronizzati. Questo scenario è tipicamente utilizzato in ambienti WAN perché si adatta bene a contesti in cui porzioni di cluster risiedono in due LAN fisicamente troppo distanti da loro e il costo della sincronizzaione sarebbe troppo alto. È possibile settare una replicazione attiva-passiva dove solo un nodo può effettuare gli update, oppure ad una totalmente attiva in cui ogni cluster può effettuare modifiche e repliche. In quest'ultimo caso eventuali conflitti possono essere risolti attravero una politica customizzabile.

6.5 Consistency

Nella gestione della replicazione ogni JVM gestisce una porzione di chiavi. Queste porzioni possono essere repliche e Hazelcast permette una configurazione sincrona o meno che ha impatti sulla consistenza. La modalità sincrona una operazione su un server attende sempre il riscontro delle repliche prima di notificare il completamento sul client. In modalità asincrona un server notifica solamente una operazione ricevuta da un client senza attendere riscontro immediato dell'effettivo completamento portando ad avere una consistenza eventuale.

Per default Hazelcast si configura in modalità sincrona in cui la lettura diretta delle repliche è disabilità. Questo garantisce la massima consistenza possibile, ma al fine di aumentare le performance delle letture è possibile abilitare la lettura diretta dei server backup diminuendo la robustezza della consistenza.

Per quanto concerne un blocco di operazioni il cluster offre il supporto alle transazioni. Attraverso l'uso di uninterfaccia TransactionContext è possibile gestire esplicitamente le tipiche operazione di open, commit e rollback. Il livello di isolamento può essere one.-phase, quindi transazione locale, oppure a two-phase che è anche il comportamento di default per garantire l'esecuzione di un blocco atomico di codice a livello di cluster. Il livello di isolamento è repeteable read in cui all'interno di una transazione sono visibili le write all'interno della stessa o committate dalle altre. Le operazioni fuori dalle transazioni vedono solo le modifiche committate.

6.6 Architecture

L'architettura di Hazelcast segue la logica di un sistema peer-2-peer che permette di rendere il sistema robusto a singoli fallimenti. L'accesso al cluster avviene per mezzo di uno strato software client che permette di effettuare tutte le operazioni supportate dal cluster senza esserne membro interno. È possibile connettersi ad un qualunque punto del sistema distribuito e delagare a tutto il cluster le operazioni oppure il client può connettersi a tutti i nodi e delegare le operazioni in maniera più intelligente.

La rete può essere suddivisa in sotto gruppi di cluster indipendenti.

6.7 Membership e Failure

Hazelcast supporta nativamente il protocollo multicast che permette ad una di unirsi ad un cluster in maniera dinamica e totalmente automatizzata. In aggiunta è possibile configurare un cluster sopra TCP/IP in cui una lista esplicita di IP può essere esplicitata. Questa lista non deve essere esaustiva, ma è necessario che almeno uno degli indirizzi indicati sia attivo affinché un nuovo membro con indirizzo sconosciuti possa aggiungersi al cluster.

La gestione dei fallimente viene affrontata attraverso l'uso di nodi server replica che può anche intervenire attivamente nella gestione delle read. Supportando la gestione di sottogruppi di cluster, Hazelcast gestisce lo scenario di split-brain in cui una caduta momentanea della rete fa credere a membri del cluster che altri siano andati in crash senza che lo siano davvero. QUesta situazione porta a gestire la fusione quando la comunicazione torna attiva. IL protocollo che gestisca questa situazione parte dal membro più vecchio del cluster che controlla se esistono altri cluster associati allo stesso nome e alla stessa password. Se ne trova cerca di capire se possono essere effettuata una fusione. Una volta avviata ogni membro coinvolto va in uno stato di pausa, disconnette tutte le connessioni, prende l'insieme di dati associati e si unisce al nuovo cluster inviando effettuando la richiesta rispetto alla mappa gestita in locale. stesso gruppo. La fusione avviene secondo la regola che il cluster più piccolo si unisce in quello più grande. Nella fase di fusione un controllo di versione (MVVC) potrebbe portare a casi di conflitto su una stessa chiave che in Hazelcast sono definiti attraverso l'implementazione di una interfaccia Java. Esistono alcune politiche built-in, ma è possibile crearne di customizzate a aggiungerle al cluster attraverso la configurazione.

6.8 Implementation

Hazelcast è implementato interamente in java e distribuito sotto forma di un unico pacchetto Jar facilmente integrabile in qualunque ambiente anche grazie a delle librerie client implementate in linguaggi Java e C#. Inoltre un'interfaccia REST e MEMCACHE permette l'utilizzo di Hazelcast totalmente indipendente dalla piattaforma applicativa.

6.9 Conclusioni

Meno famoso rispetto a BigMemory, Hazelcast si sta velocemente raccongliendo una fetta di mercato come possibile alternativa in grado di offrire un ambiente di lavoro più integrato e molto più attento agli standard Java e non. In maniera analoga a terracotta lo strato software esposto non è un database NoSQL nativo ma può essere implementato di sopra attraverso le mappe distribuite. La versione enterprice, oltre alla gestione della sicurezza, offre uno spazio di memoria off-heap non gestito dal Gabage Collector di Java per dare potenzialmente spazio illimitato alle applicazioni con prestazioni paragonabili a quelle ottenibili con linguaggi di più basso che permettono un utilizzo più ottimizzato della memoria.

La versione open source pur non godendo di alcune caratteristiche resta superiore a quella proposta da Terracotta in quanto non impone limiti sul numero di macchine e più in generale permette alle applicazioni di poter comunque effettuare tutte le operazioni tipiche del cluster.

Problema	Tecnica	Vantaggi
Partizionamento	Consistent Hash	Massima distribuzione e trasparenza
Consistenza e Isolamento	Configurabile	Livello di consistenza adattabile in fase di configurazione
Gestione fallimenti temporanei	Server Replica	Elevata disponibiltà
Gestione fallimenti permanenti	Redistribuzione	Forte robustezza
Membership	Mulsticast - TCP/IP	Totalmente automatizzabile

Vantaggi:

- Ottima implementazione degli standard Java e non.
- Gestione dati in memoria off-heap.
- Supporto ufficiale per vari linguaggi, memcache e REST.
- Ricerca per chiavi o dialetto SQL.

- Indicizzazione campi attraverso classi POJO.
- Serializzazione customizzabile.
- Supporto lock espliciti e transazioni.
- Livello di consistenza configurabile.
- Partizionamento automatico con supporto per cluster WAN.
- Disponibiltà configurabile tramite backup.
- Massima architettura peer-2-peer.
- Gestione fallimenti automatica.
- Membership automatica.
- Supporto ad eventi nel cluster.
- Supporto ad esecuzione di codice remota.

Svantaggi:

- Versione open source senza off-heap.
- Assenza di un API NoSQL esplicita.
- Partizionamento non descritto e poco configurabile.

Parte II
Analisi principali datastore JCR

7 Apache Jackrabbit

Apache Jackrabbit è l'implementazione ufficiale della specifica standard Java sui repository di contenuti: la specifica JCR, sviluppata sotto la Java Community Process come JSR-170 (nella version 1) e JSR-283 (nella versione 2). Il package contenente tutte le classi è javax.jcr.

7.1 Il clustering

Il clustering in Jackrabbit lavora in modo che i contenuti sono condivisi tra tutti i nodi. Questo intende che tutti i nodi devono accedere ad una stessa entità di persistenza qualunque sia la sua natura (persistence manager, datastore e repository file system). Di conseguenza sia il persistence manager che i datastore sono essere clusterizzabili.

Nonostante questo ogni nodo deve comunque avere una repository directory privata e ogni cambiamento fatto deve essere riportato in un journal che può essere basato su file o scritto su un database.

In una configurazione d'esempio[4] si desidera configurare un cluster usando Jackrabbit usando Apache Tomcat come tomcat, NFS per il filesystem distribuito e PostgresSQL come database.

In primo luogo è necessario configurare la referenza per la connesione JDBC verso PostgreSQL. Per farlo si deve editare il file server.xml nella directory CATALINA_HOME/conf di Tomcat configurando la sezione GlobalNamingResources.

[4] http://www.greymeister.net/blog/2011/11/28/jackrabbit-clustering-primer/

```
<!--
     Global JNDI resourcesDocumentation
     at /docs/jndi-resources-howto.html
-->
<GlobalNamingResources>
     <!--
          Editable user database that can also
          be used byUserDatabaseRealm to authenticate users
     -->
     <Resource
               name="UserDatabase"
               auth="Container"
               type="org.apache.catalina.UserDatabase"
               description="User database that can be updated and saved"
               factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
               pathname="conf/tomcat-users.xml"
     />
     <Resource
               name="jdbc/repository"
               auth="Container"
               type="javax.sql.DataSource"
               driverClassName="org.postgresql.Driver"
               url="jdbc:postgresql://192.168.0.8:5432/jr_repository"
               username="jackrabbit" password="jackrabbit"
               validationQuery="select version();"
               maxActive="20"
               maxIdle="10"
               maxWait="-1"
     />
</GlobalNamingResources>
```

In seguito è necessario aggiungere la risorsa JDBC come JNDI editando il file context.xml.

```
<!--
     The contents of this file will be loaded
     for each web application
-->
<Context>
     <!-- Default set of monitored resources -->
     <WatchedResource>WEB-INF/web.xml</WatchedResource>

     <!-- Uncomment this to disable session persistence across Tomcat restarts -
     <!--<Manager pathname="" /> -->

     <!--
          Uncomment this to enable Comet connection tacking
          (provides eventson session expiration as well as webapp lifecycle)
     -->
```

```
    <!-- <Valve className="org.apache.catalina.valves.CometConnectionManagerVal

    <ResourceLink
            global="jdbc/repository"
            name="jdbc/repository"
            type="javax.sql.DataSource"
    />

    <Resource
            name="jcr/repository"
            auth="Container"
            type="javax.jcr.Repository"
            factory="org.apache.jackrabbit.core.jndi.BindableRepositoryFactory"
            configFilePath="/srv/repository/repository.xml"
            repHomeDir="/srv/repository"
    />
</Context>
```

Dopo aver terminato la impostazioni di tomcat è necessario configurare il repository JCR tramite il file repository.xml.

Un primo aspetto fondamentale è la definizione del FileSystem per condividere i contenuti.

Nella sezione FileSystem si utilizza il DbFileSystem, presente nelle librerie standard di Jackrabbit. La configurazione permette l'accesso al database PostgreSQL tramite la connessione JNDI definita nei file di Tomcat.

```
<!--
        virtual file system where the repository stores global state
        (e.g. registered namespaces, custom node types, etc.)
-->
<FileSystem class="org.apache.jackrabbit.core.fs.db.DbFileSystem">
    <param
            name="driver"
            value="javax.naming.InitialContext"
    />

    <param
            name="url"
            value="java:comp/env/jdbc/repository"
    />

    <param
            name="schemaObjectPrefix"
            value="rep_"
    />

    <param
```

```
                name="schema"
                value="postgresql"
    />
</FileSystem>
```

Nella sezione DataStore si utilizza il FileDataStore anch'esso builtin in Jackrabbit.

```
<!--data store configuration -->
<DataStore class="org.apache.jackrabbit.core.data.FileDataStore">
    <param
                name="path"
                value="${rep.home}/datastore"
    />

    <param
                name="minRecordLength"
                value="100"
    />
</DataStore>
```

La sezione workspace d'esempio utilizza un PersistanceManager basato sulle raccomandazioni di Jackrabbit.

```
<Workspaces rootPath="${rep.home}/workspaces" defaultWorkspace="default"/>
<!--
        workspace configuration template used to create
        the initial workspace if there is no workspace yet
-->
<Workspace name="${wsp.name}">
    <!--
        virtual file system of the workspace:
            class: FQN of class implementing the FileSystem interface
    -->
    <FileSystem class="org.apache.jackrabbit.core.fs.local.LocalFileSystem">
        <param name="path" value="${wsp.home}"/>
    </FileSystem>

    <!--
        persistence manager of the workspace:
            class: FQN of class implementing the PersistenceManager interface
    -->
    <PersistenceManager class="org.apache.jackrabbit.core.persistence.pool.Post
        <param
                name="driver"
                value="javax.naming.InitialContext"
        />

        <param
```

```
                name="url"
                value="java:comp/env/jdbc/repository"
        />

        <param
                name="schemaObjectPrefix"
                value="ws_"
        />
        <param
                name="schema"
                value="postgresql"
        />
    </PersistenceManager>
</Workspace>
```

Il pezzo più importante è la sezione cluster che ancora una volta punta alla risorse PostgresSQL tramite JNDI. Tale risorse verrà usata per memorizzare il journal. L'attributo di deve essere univoco per ogni nodo.

```
<!-- Cluster configuration with system variables -->
<Cluster id="node1" syncDelay="2000">
    <Journal class="org.apache.jackrabbit.core.journal.DatabaseJournal">
        <param
                name="revision"
                value="${rep.home}/revision.log"
        />

        <param
                name="driver"
                value="javax.naming.InitialContext"
        />

        <param
                name="url"
                value="java:comp/env/jdbc/repository"
        />

        <param
                name="databaseType"
                value="postgresql"
        />

        <param
                name="schemaObjectPrefix"
                value="journal"
        />
    </Journal>
</Cluster>
```

Una volta terminata la configurazione su ogni macchina per avviare il cluster è necessario aggioranre il database del journal con l'id e la revisione di ogni nodo che si vuole aggiungere la cluster.

Terminato il tutto è sufficiente avviare le macchine e i tomcat installati per avere la gestione del repository JCR clusterizzata.

8 JBoss ModeShape

ModShape è un datastore distribuito, gerarchico, transazione e consistente che supporta query, ricerche full-text, eventi, versioning, referenze e uno schema dinamico e flessibile. È scritto interamente in Java e i client possono usare la specifica standard JSR-283 (nota come JCR) oppure un API REST proprietaria. Le query possono essere effettuare tramite JBDC e SQL.

ModeShape è perfetto per i dati che sono organizzati in una struttura ad albero dove informazioni correlati sono "vicini" tra loro e la loro navigazione è comune ed importante quanto il lookup di dati su un sistema key-value e quanto un'interrogazione basta su query.

L'organizzazione gerarchica è simile a un file sistem che rende ModeShape naturale per la memorizzazione di file con metadati. Inoltre può automaticamente estrarre informazioni strutturate all'interno dei file in modo che i client possono navigare o usare query per trovare i file che soddisfano determinati criteri.

ModeShape è adatto per memorizzare dati con schema complessi che possono evolvere nel tempo. Questo lo rende adatto per tutte le applicazioni che necessitano di un datastore distribuito.

ModeShape supporta al 100% le specifiche di base JRC2.0

- acquisizione di repository.
- autenticazione.
- lettura navigazione.
- query.
- export.
- discovery dei tipi di nodo.

Sono supportate molte delle specifiche opzione JCR2.0

- scrittura.
- import.
- observation.
- controllo d'accesso.
- versioning.
- locking.
- gestione dei tipi di nodo.

- sibling di nodi con lo stesso nome.
- nodi condivisi.
- mix:etag, mix:created and mix:lastModified mixins con proprietà autocreated.

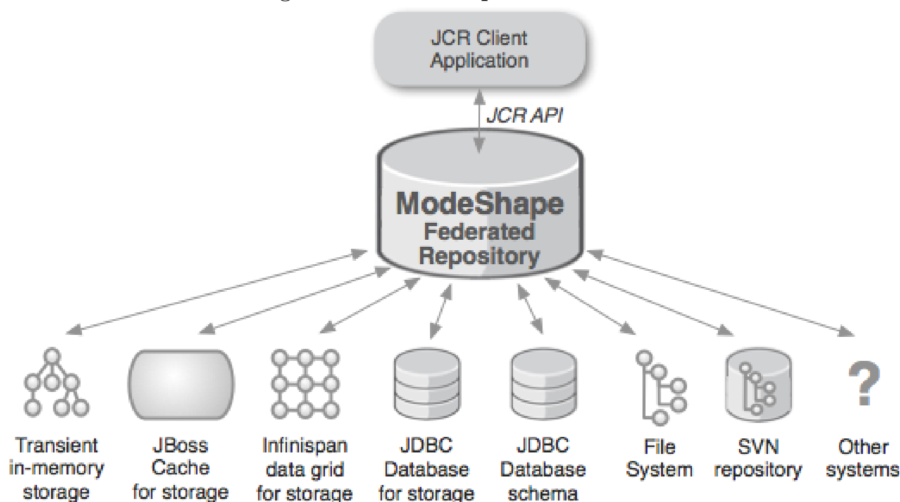
ModeShape supporta le due forme di interrogazione definite in JSR-283 (JCR-SQL2 e JCR-JQOM), alle quali aggiunge il supporto alle query JCR 1.0 (Xpath e JCR-SQL). Infine definisce anche un linguaggio per la ricerca full-text.

8.1 Caratteristiche chiave

L'idea chiave di ModeShape è di utilizzabile uno standard noto e stabile come il JCR per offrire un livello astratto con cui i client possono accedere a contenuti e informazioni preesistenti su altri servizi (database, filesystem, cloud ecc). L'idea è di usare JCR per dare una visione unificata ai client permettendo ai servizi di continuare a utilizzare normalmente la porzione di informazioni che possiedono. Grazie a questo strato d'alto livello è anche possibile memorizzare informazioni aggiuntive quali meta-informazioni e relazioni tra contenuti di alto livello.

L'immagine mostra uno schema delle entità in gioco in un sistema orientato a una gestione su JCR e ModeShape.

Figura 69: ModShape: Architecture



Sommariamente le caratteristiche chiave ModeShape sono riassumibili nei seguenti punti.

- Organizzazione gerarchica dei dati in forma d'albero di nodi con proprietà singole o multiple.

- Cache dei dati e memorizzazione su Infinispan che può persistente su filesytem, database, cloud.
- Distribuzione e replicazione dei dati su diverse macchine con gestione in-memory.
- Implementazione delle specifiche JSR-283 note come JCR 2.0.
- Definizione di schema con tipi di nodo e mixin che (opzionalmente) limite le proprietaria e figli alcuni tipi di nodo.
- Schema evolutivo nel tempo senza migrazione.
- Utilizzo di svariati linguaggi di query (JCR-SQL, JCR-SQL2, JCR-JQOM, Xpath, full-text engine).
- Utilizzo di sessioni per creare e validare grandi quantità di contenuto transiente da rendere persistente in blocco.
- Implementazione di gran parte delle elle specifiche opzionali di JRC-2.0.
  - JCR-2.0 transaction : supporto alle transazioni JTA tramite le JRC API.
  - JCR-2.0 events: supporto alle notifiche tramite eventi che catturano qualunque cambiamento nel cluster.
  - JCR-2.0 isolation:isolamento dei dati in workspace e repository multipli.
- Integrazione in qualunque Java SE, EE o applicazione WEB.
- Installazione su JBoss AS7 per configurazione, gestione e monitoraggio dei repository in maniera centralizzata.

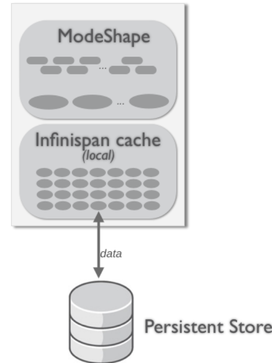
8.2 Clustering

É possibile creare un repository ModeShape sia in modalità standalone e sia tramite un cluster di repository multipli che lavorano per assicurare che tutti i contenuti sia accessibili. Quando si crea un cluster i client possono comunicare con un qualunque nodo avendo sempre la stessa visione. In altre parole la gestione del cluster è totalmente trasparente ai client che non possono sapere se stanno comunicando con un repository locale o distribuito.

Esistono vari modi per implementare un cluster, ma la decisione fondamentale è dove devono essere memorizzati i contenuti. Grazie alla flessiblità del data grid e cache in-memory Infinispan si possono usare varie topologie: locale, replicate, invalidate, distribuite e remote.

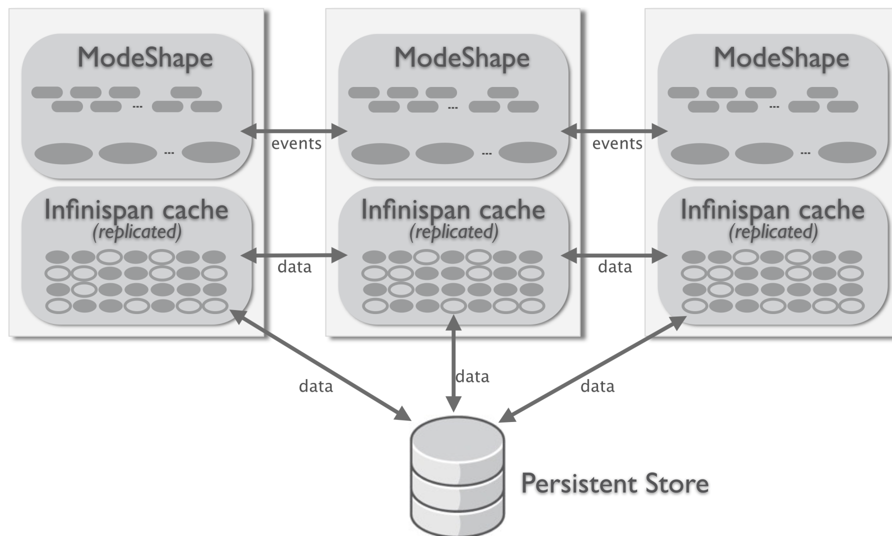
La modalità locale è quella di default che di fanno non effettua nessun cluster. Ogni processo lavora senza comunicazione o condivisione di contenuti e le modifiche non sono visibili tra processi.

Figura 70: ModShape: Local Configuration



La più semplice topologia di clustering è quella replicata dove diverse copie dei contenuti sono memorizzate in tutti i membri del cludter. Ogni membro ha il suo storage di contenuti privato in cui nulla è condiviso e sono i processi ModeShape (con Infinispan) a mantenere la sincronia.

Figura 71: ModShape: Clustering Configuration 1

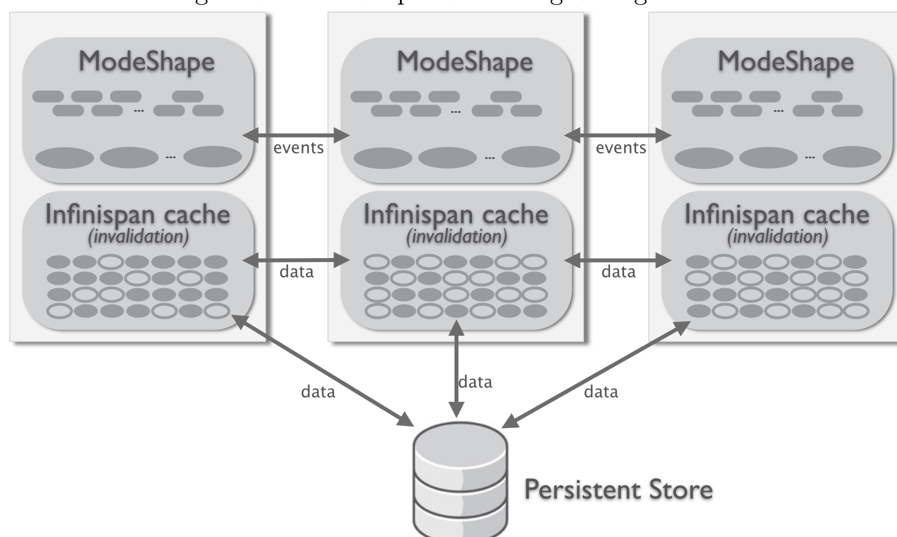


IL vantaggio di questa topologia è che ogni membro ha un insieme di contenuti completo in modo da poter soddisfare ogni operazione di read senza comunicazioni esterne. L'assenza di condivisione di contenuti facilita l'aggiunta e rimozione di server. Ogni contenuti è mantenuto in memoria in modo da velocizzare le operazioni di modifica e propagazione della stessa su tutto il cluster. In generale è una struttura adatta per repository di piccole e medie dimensioni.

La maggior parte delle volte ModeShape dovrebbe essere configurato in modo che tutti i membri dovrebbero condividere un livello di persistenza transazionale, e capace di coordinare accessi multipli allora i contenuti. In questo modo si riesce a recuperare tutti i contenuti del cluster anche ad uno stop di tutte le macchine. Poiché tutti i dati sono replicati è comunque possibile non avere il livello di persistenza purché si ha una ragionevole affidabilità di avere sempre almeno una macchina attiva e che ognuna di esse abbia memoria a sufficienza per contenere tutto il repository.

Una modalità simile alla replicazione è l'utilizzo dell'invalidazione. La differenza sta che quando un nodo cambia o si aggiunge un processo, un cluster non genera una notifica dell'aggiornamento, ma si forza a il livello di persistenza.

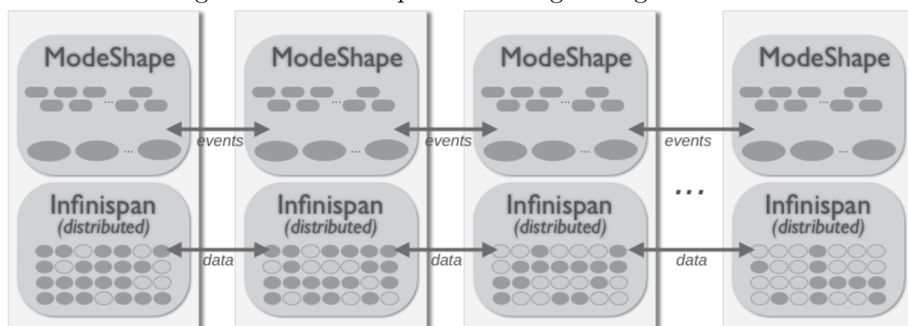
Figura 72: ModShape: Clustering Configuration 2



Quando la dimensione del cluster cresce, il peso di dover avere una copia completa per ogni nodo risulta eccessivo. Inoltre anche l'overhead di utilizzo della rete e dei lock non è più trascurabile. In scenari dove il numero di macchine è elevato diventa utile configurare una topologia distribuita. In questa modalità ogni contenuto è gestito da un sotto insieme delle macchine presenti nel cluster, tipicamente, per garantire una affidabilità, almeno due. Se il cluster risiede su data-center separati il sistema diventa robusto anche a catastrofi di tipo naturale (incendi o terremoti) e questo permette, se il numero di macchine è sufficientemente elevato, di poter evitare l'uso di un datastore persistente e di utilizzare solo il database in-memory. Fissando con n il numero minimo di repliche per ogni contenuto, si può avere un limite massimo di fallimenti contemporanei che il sistema è certamente in gredo di reggere senza perdita di dati. La scelta del fattore n è critica per il comportamento del cluster. Valori alti, pur aumentando l'affidabilità, avvicinano troppo la configurazione distribuita a quella replicata con il degrado che ne consegue. Valori

troppi bassi rischiano di portare il sistema a perdere dati. In ogni caso ad una perdita eccessiva di nodi Infinisopan automaticamente rimescola i dati per assicurare nuovamente il numero minimo di repliche per ogni contenuto.

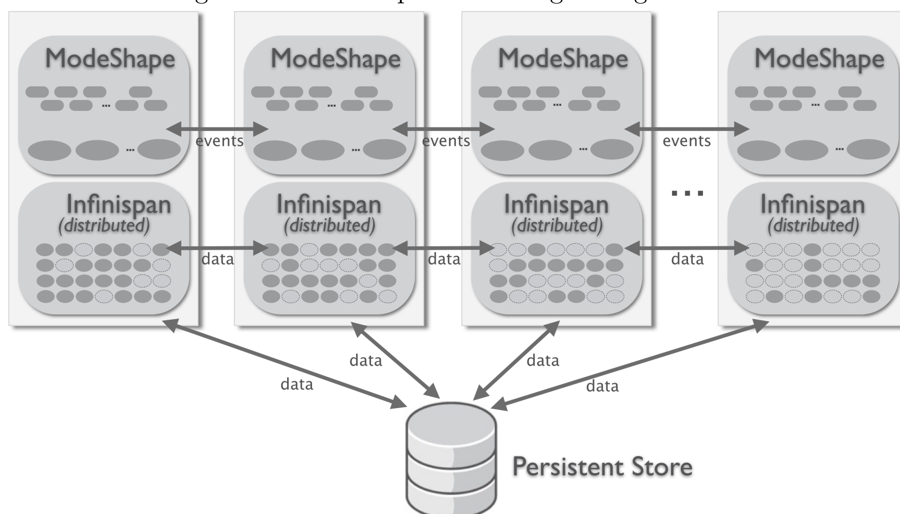
Figura 73: ModShape: Clustering Configuration 3



Per rendere meno critica la gestione dell'affidabilità è possibile rinunciare all'elevate prestazioni che una gestione solamente in-memory offre affiancando uno strato di persistenza condiviso.

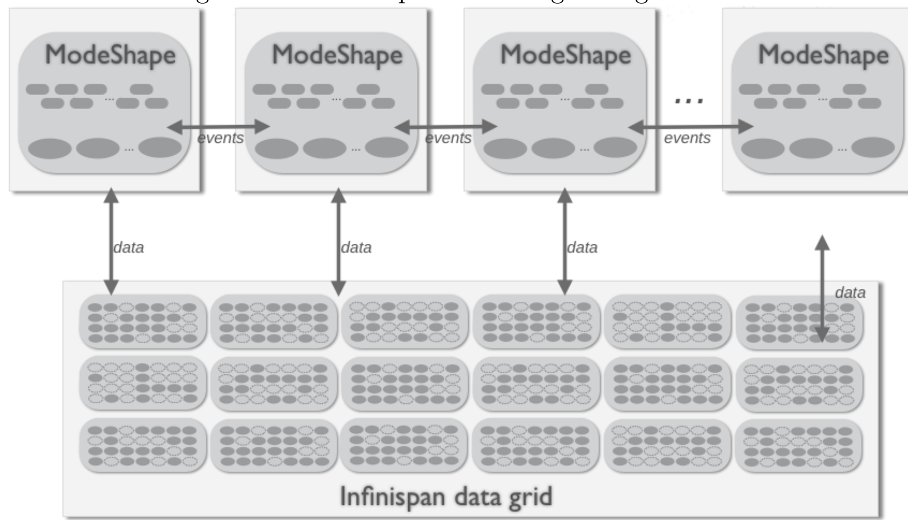
In altre parole il cluster si basa sulla strato di memorizzazione condiviso per mantenere la persistenza dei contenuti senza rinunciare alla natura distribuita di Infinispan che continua a mantenere in-memory le stesse informazioni persistenti. Se un nodo deve eseguire una read su un contenuto che non ha, tipicamente preferisce, per ragioni di performance, inoltrare la richiesta sul processo che gestisce quel contenuto piuttosto che eseguire una leggura sullo strato persistente.

Figura 74: ModShape: Clustering Configuration 4



In tutte queste topologie ModShape e Ininispan risiedevano su ogni nodo del cluster. In realtà potrebbe essere auspicabile separare la gestione della griglia di dati dal repository JCR. Per farlo si può configurare ModShape in maniera remota dove l'intera griglia di dati gestita da Infinispan risiede in un sistema esterno raggiungibile attraverso chiamate remote.

Figura 75: ModShape: Clustering Configuration 5



Parte III
Comparazione database NoSQL

Le soluzioni studiate rappresentano lo stato dell'arte per la persistenza distribuita che continua a essere in evoluzione e al centro dell'interesse delle attività di ricerca. La vastità e varietà di soluzioni NoSQL portano ad avere il mercato delle soluzioni di persistenza al sua massima flessibilità. Piuttosto che cercare la migliore soluzione a priori è preferibile analizzare il contesto reale che si vuole affrontare, quali sono i caratteristiche prioritarie e trovare il software che più si orienta verso quelle caratteristiche.

Il seguente rapporto[5] esegue una breva comparasione di tutte le soluzioni NoSQL attualmente in voga.

8.3 MongoDB (2.2)

Orientato a query dinamiche dove si preferisce l'utilizzo degli indici in luogo di funzioni map/reduce. Alternativa a CouchDB per dati che cambiano frequentemente.

- Scritto in C++
- mantiene alcune proprietà di SQL (ricerca a indicizzazione)
- Licenza: AGPL
- Protocollo: custom e binario (BSON)
- Replicazione Master/slave con auto failover
- Sharding integrato
- Ricerca tramite espressioni JavaScript
- Esecuzione arbitraria di codice JavaScript server-Side server-side
- Migliore update-in-place di CouchDB
- Usa file di che mappano la memoria su disco
- Prestazioni rispetto caratteristiche
- Opzione Journaling
- Su sistemi a 32bit è limitato a 2.5Gb
- POco adatto su sistemi legacy (database vuoti consuma 200MB di RAM)
- Utilizzo di GridFS per memorizzara grandi dati e metadati
- supporto a indicizzazione geospaziale
- Gestione dei data center multipli

Utilizzo tipico: migrazione da ambienti MySQL e PostgreSQL.

[5] http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis

8.4 Redis (2.8)

Prestazioni eccezionali nel cambiamento dei dati con dimensione del database predicibile.

- Scritto in C
- Velocità sfolgorante
- Licenza BSD
- Protocollo binario simile a telenet
- Database in-memory con utilizzo backend del disco
- Gestione memoria potenzialmente illimitata
- Replicazione master-slave con gestione automatica del failover
- Modello chiave-valore
- Utilizzo di dati semplici o strutturati
- Operazioni complicate come ZREVRANGEBYSCORE
- INCR & co (ottimi per rate limiting e statistiche)
- Operazioni basate su bit (come filtro bloom)
- Supporto alla set (anche union/diff/inter)
- Supporto alle liste (anche code e blocking pop)
- Supporto alle tabelle hash (oggetti di campi multipli)
- Supporto all'ordinamento (tabelle punteggio e range query)
- Supporto a script server-side tramite Lua
- Supporto alle transazioni
- Supporto alle cache tramite expire sui dati
- Supporto al meccanismo pub/sub per la messagistica

Utilizzo tipico: mercato azionario, gestione analitica delle informazioni, collezione di dati in tempo reale e più in generale dovunque sia utile avere un meccanismo che eredita le idee di memcache aggiungendo numerose caratteristiche.

8.5 Riak (1.2)

Alternativa open-source a Amazon Dynamo, ma meno complessa. Utile per aggiungere scalabilità e affidabilità senza il costo di una replicazione multi-site.

- Scritto in Erlang & C e JavaScript
- Orientato alla tolleranza ai guasti
- Licenza: Apache
- Protocollo HTTP/REST o protocollo custom binario
- memorizzazione di oggetti BLOB
- trade-off distribuzione e replicazione tunable
- esecuzione di codice JavaScript o Erlang prima e dopo un commit
- Esecuzione di map/reduce in JavaScript o Erlang
- Navigazione tramite concetto di database a grafo
- Indicizzazione secondaria una per volta
- Supporto a oggetti di grandi dimensioni (Luwak)
- Versione open-source" ed enterprise"
- Ricerca full-text tramite motore proprietario Riak Search
- Replicazione senza master e monitorazzio SNMP nella edizione commerciale

Utilizzo tipico: punti di vendita di piccole e medie dimensioni o ambienti dove si vuole evitare il disservizio.

8.6 CouchDB (1.2)

Orientato ad accumulare dati che cambiano raramente nel tempo e sulle quali si vuole effettuare ricerche predefinite. Ottima gestione del versioning.

- Scritto in Erlang
- Orientato alla consistenza
- Licenza Apache
- Protocollo HTTP/REST
- replicazione multi-master bidirezionale, continua o ad-hoc
- Supporto al controllo di versione (MVCC)
- Operazioni di write non bloccano le read
- Revisione dei documenti
- Necessità di una fase di compattamento dei documenti

- Views: Ricerca map/reduce integrata
- Formattazione Views: list e shot
- validazione documenti server-side
- Supporto all'autenticazione
- Update in Real-time updates
- Gestione attachment
- getsione di applicazione JavaScript standalone via ChurchApp

Utilizzo tipico: CRM, CMS, ambienti adatti alla replicazione multi master-master e alla gestione multi-site.

8.7 HBase (0.92.0)

Uno dei migliori utilizzi di Apache Hadoop per l'esecuzione di ricerche map/reduce su grandi moli di dati.

- Scritto in Java
- Orientato alla gestione di milioni di colonne
- Licenza Apache
- Protocollo HTTP/REST o Thrift
- Modellato su Google BigTable
- Basato su Apache file system distribuito Aoache Hadoop
- Predicati di ricerca server-side via scan e get
- Ottimizzazione per query in real time
- Elevate prestazioni tramite gateway Thrift
- Supporto XML, Protobuf e binario
- Console di comandi con ambiente JRuby (JIRB)
- Rolling restart per cambianti di configurazione e upgrade minori
- Accesso casuale ad elevate prestazioni stile MySQL
- Un cluster che consisten di differenti tipi di nodi

Utilizzo tipico: Motori di ricerca, analisi di log e qualunque scenario dove è necessario eseguire scan su grandi dataset orientati a tabelle bidimensionali senza operazioni di join.

8.8 Cassandra (1,2)

Ottime prestazioni sulle write con consistenza variabile lato client. Adatto in ambienti Java.

- Scritto in Java
- Basato su Google BigTable e Amazon Dynamo
- Licenza Apache
- Protocollo Thrif e linguaggio CQL3
- Replicazione e distribuzione tunable lato client
- Configurazione per ricerca tramite colonna, range di chiavi
- Modello dati ereditato da Google BigTable con l'estensione del supporto a Super ColumnFamily
- Può essere usato come un tabella distribuita (DHT) con linguaggio SQL-Like (senza JOIN)
- Gestione dati cache tramite expiration (Opzione nell'INSERT)
- Operazioni di write più veloci delle read
- Integrazione con Apache Hadoop per map/reduce
- Totalmente peer-to-peer (tutti i nodi sono dello stesso livello)
- gestione delle catastrofi naturali tramite multi data center

Utilizzo tipico: ambienti bancari, industria finaziaria (non intesa come transazioni, ma come trattamento delle grandi quantità di dati di cui dispongono). Ambienti dove le scritture devono essere più veloci delle letture.

8.9 Hypertable (0.9.6.5)

Una promettente alternative ad HBase e Google BigTable.

- Scritto in C++
- Più leggero e piccolo di HBase
- Licenza GPL 2.0
- Protocolli Thrift, client C++ o console HQL
- Esecuzione su Apache Hadoop HDFS
- Linguaggio HQL come dialetto SQL
- Ricerca per chiavi, celle o per valori in ColumnFamily
- ricerca su range di chiavi e colonne
- Sponsorizzato da Baidu

- Accesso alle ultime revisioni dei dati
- namespace sulle tabelle
- Esecuzione map/reduce su Hadoop

Utilizzo tipico: stessi scenari adatti ad HBase.

8.10 Accumulo (1.4)

Una promettente alternative ad HBase e Google BigTable.

- Scritto in Java e C++
- Estensione del modello di BigTable con supporto della sicurezza a livello di cella
- Licenza Apache
- Protocollo Thrift
- Clone di BigTable basato su Apache Hadoop
- Salvataggio su disco in caso di overflow della RAM
- Utilizzo della memoria off-heap tramite C++ STL
- Caratteristiche ereditate dall'ambiente Hadoop (ZooKepper etc ...)

Utilizzo tipico: stessi scenari adatti ad HBase.

8.11 Neo4j (1.5M02)

Adatto per scenari con dati ricchi, complessi e interconnessi in un modello a grafo.

- Scritto in Java
- Database a grafo
- Licenza open-source GPL e AGPL/commerciale
- Protocollo HTTP/REST o tramite client Java
- Utilizzo standalone o integrato in applicazioni Java
- Totale supporto alle proprietà ACID
- Presenza di meta-dati associati ai nodi e alle relazioni
- Ricerca pattern-matching basata su linguaggio Cypher
- Navigazione del grafo tramite linguaggio Gremlin
- Indicizzazione dei nodi e delle relazioni
- Amministrazione tramite interfaccia WEB integrata
- Ricerca di cammini sul grafo tramite algoritmi avazanti

- Ottimizzato per le read
- Supporto transazionale tramite API Java
- Supporto allo script server-side tramite Groovy
- Backup online, monitoraggio avanzato ed elevata disponibilità per la edizione commerciale

Utilizzo tipico: ricerca di relazioni in social network, documenti ipertestuali, mappe stradali o qualunque scenario con una topologia.

8.12 ElasticSearch (0.20.1)

Orientato a scenari dove gli oggetti hanno campi flessibili e necessitano di criteri di ricerche avanzate.

- Scritto in Java
- Ottimizzato per le ricerche avanzate
- Licenza Apache
- Protocollo JSON su HTTP (plug-in per Thrift e memcache)
- Modello di dati a documenti JSON
- Supporto al versioning
- Gerarchia padre figli sui documenti
- I documenti possono avere un tempo di expire
- Meccanismo di ricerca estremamente versatile, sofisticato.
- Supporto allo script server-side
- Consistenza sulle write tunable (one, quorum o all)
- Ordinamento tramite punteggio
- Ordinamento su logica geospaziale
- Ricerche Fuzzy (dati approssimati etc...)
- Replicazione asincrona
- Atomicità, update scriptati (ottimo per contatori etc...)
- Un solo sviluppatore (kimchy).

Utilizzo tipico: un servizio di appuntamenti dove gestire differenze di età, di residenza, gusti etc....

8.13 Couchbase (ex-Membase) (2.0)

Orientato a scenari dove si richiede accesso a bassissima latenza, elevata concorrenza e disponibilità.

- Scritto in Erlang e C
- Compatibilità con memcache con supporto a persistenza e clustering
- Licenza Apache
- Velocità di accesso ai dati per chiave
- Persistenza su disco
- Totalmente peer-to-perr (tutti i nodi sono identici)
- Operazioni di write de-duplication per ridurre l'IO
- Gestione del cluster tramite interfaccia WEB
- Connessione tramite proxy su pool e multiplexing
- Supporto a map/reduce incrementale
- Replicazione su data center multipli

Utilizzo tipico: casi di bassa latenza come gaming online.

8.14 VoltDB (2.8.4.1)

Pensato scenari in cui si deve agire velocemente su grandi quantità di dati in arrivo.

- Scritto in Java
- Focalizzato su transazioni veloci e rapidi cambiamenti di dati
- Licenza GPL3
- Protocollo proprietario
- Database relazionale in-memory
- Supporto all'export su Apache Hadoop
- Supporto ad ANSI SQL
- Supporto alle stored procedure in Java
- Supporto alla replicazione su diversi data center

Utilizzo tipico: analisi di punti di vendita.

8.15 Kyoto Tycoon (0.9.56)

Utile quando si deve lavorare con precisi algoritmi di storage senza perdere in velocità.

- Scritto in C++
- Una database leggero per rete di database (DBM)
- Licenza GPL
- Protocollo HTTP (TSV-RPC o REST)
- Successo di Tokyo Cabinet
- Utilizzo backend di diversi sistemi di storage (hash, tree, dir etc...)
- Oltre il milione di operazioni al secondo
- Leggermente meno efficiente rispetto a Cabinet a causa di un overhead
- Binding in svariati linguaggi (C, Java, Python, Ruby, Lua etc...)
- Utilizzo del pattern visitor
- Backup e replicazione asincrona
- Snapshop in background dei database in-memory
- Supporto al meccanismo di cache tramite expiration

Utilizzo tipico: server di cache, prezzi azionari, analisi dei dati, gestione di dati in tempo reale e dovunque sia adatto un meccanismo stile memcache.

8.16 Scalaris (0.5)

Ottimo utilizzo di Erlang per usare Mnesia, DETS o ETS in maniera più accessibile e scalabile.

- Scritto in Java
- Database distribuito peer-to-peer
- Licenza Apache
- Protocollo JSON-RPC e proprietario
- Gestione in-memory con utilizzo backend del disco se integrato in Tokyo Cabinet
- Integrazione con YAMS come server WEB
- Supporto alle transazioni (commit Paxos)
- Consistente, distribuito e orientato alle write

Utilizzo tipico: un sistema basato Erlang-based con binding in Python, Ruby e Java.