ALGORITMI DISTRIBUITI IN BAZE DE DATE NOSQL

Rețele de calculatoare în internet

Student: Alexandru Nicolae ION

Contents

Algoritmi distribuiți în baze de date NoSql

Scalabilitatea este unul dintre motoare principalele ale mișcării NoSQL. Ca atare, aceasta cuprinde un sistem de coordonare distribuit, toleranța la defecte, managementul resurselor și multe alte capacități. Sună ca o umbrelă mare și chiar este. Deși poate fi spus cu greu că mișcarea NoSQL a adus noi tehnici fundamental diferite de prelucrare a datelor distribuite, aceasta a declanșat o avalanșă de studii practice și studii reale cu diferite combinații de protocoale și algoritmi. Aceste dezvoltări au evidențiat treptat un sistem de blocuri de construcție de baze de date relevante cu eficiență dovedită practic. Acest proiect încercă să furnizez o descriere mai mult sau mai puțin sistematică de tehnici legate de operațiuni în baze de date NoSQL distribuite.[1]

Vom studia o serie de activități, cum ar fi replicarea, detectarea eșecului, care ar putea apărea într-o bază de date. Aceste activități, evidențiate mai jos cu caractere ingroșate, sunt grupate în trei secțiuni principale:

Consistența datelor. Din punct de vedere istoric, NoSQL și-a îndreptat atenția către compromisuri între consistență, toleranță la erori și performanță, pentru a servi sisteme distribuite geografic, latență mică sau aplicații extrem de disponibile. Fundamental, aceste compromisuri se învârt în jurul consistenței datelor, astfel încât această secțiune este dedicată replicarii datelor și reparațiilor de date.

Plasarea datelor. O bază de date ar trebui să se acomodeze la distribuții de date diferite, topologii de grup și configurații hardware. În această secțiune vom discuta despre cum să distribuim sau reechilibra datele în așa fel încât eșecurile sunt tratate rapid, garanțiile de persistenta sunt menținute, interogările sunt eficiente, și resursele de sistem cum ar fi RAM sau spațiu pe disc sunt folosite în mod egal de-a lungul clusterului.

Coordonarea sistemului. Tehnicile de coordonare, cum ar fi alegerea liderului sunt folosite în multe baze de date pentru pune în aplicare toleranța la erori și o puternică coerență a datelor. Cu toate acestea, chiar bazele de date descentralizate își urmăresc de obicei starea lor globală, detectează defecțiuni și modificări de topologie. Această secțiune descrie mai multe tehnici importante care sunt folosite pentru a menține sistemul într-o stare coerentă.[1]

Consistența datelor

Este bine cunoscut și destul de evident că, în sistemele distribuite geografic sau alte medii cu partiții de rețea sau întârzieri, în general nu este posibil să se mențină o disponibilitate ridicată, fără a sacrifica consistența, din cauza părților izolate ale bazei de date care trebuie să funcționeze independent în cazul de partițiilor de rețea. Acest fapt este adesea menționat ca teorema CAP. Cu toate acestea, consistența este un lucru foarte scump în sistemele distribuite, încât poate fi negociată nu numai cu disponibilitatea. Acesta este adesea implicată în mai multe compromisuri. Pentru a studia aceste compromisuri, observăm în primul rând că problemele de coerență în sistemele distribuite sunt induse de replicare și de separarea spațială a datelor cuplate, așa că trebuie să începem cu obiectivele și proprietățiile dorite ale replicării.[1]

Replicarea

Unele din principalele cerinte ale sistemelor distribuite sunt fiabilitatea si disponibilitatea. In conformitate cu acestea defectarea unuia dintre nodurile sistemului nu va paraliza functionarea sistemului si nici nu va afecta disponibilitatea datelor care au fost înmagazinate la nodul respectiv. De cele mai multe ori sunt păstrate mai multe copii ale acelorași date în mai multe locații cu scopul realizarii autonomiei locale cerute și a creșterii disponibilității. Pentru a se asigura consistența bazei de date este obligatorie replicarea fragmentelor între locații în scopul reflectării modificărilor asupra acestora. Aducerea la zi a datelor păstrate în mai multe copii ca fragmente distribuite poartă denumirea de replicare.[2]

Replicarea este un proces care constă în realizarea și distribuirea unor copii a datelor, numite și replici în scopul asigurării posibilității de procesare locală a acestora, oferind astfel un nivel cât mai ridicat de autonomie pentru bazele de date locale. Un nivel ridicat de autonomie și implicit de disponibilitate, implică o serie de concesii privind actualitatea informatiei utilizate.[2]

Replicarea este o tehnologie care permite ca informații ce provin de la una sau mai multe surse să poată fi distribuite catre una sau mai multe ținte, cu propagarea consistentă a modificărilor intervenite la surse catre țintele corespunzatoare.[2]

Este logic să considerăm că procesul de replicare nu se referă la replicarea întreagii baze de date, care ar încărca foarte mult sistemul de comunicație, ci doar un set de date, element care complică suplimentar procesul de replicare. Un alt termen des întâlnit în tehnologia replicării este cel de sincronizare, ca fiind procesul prin care se asigură capturarea, propagarea și reproducerea la ținte a actualizarilor de la surse.[2]

Obiectivele replicării

- Disponibilitatea. Părți izolate ale bazei de date pot servi cereri de citire/scriere în caz de partiționare de rețea.
- Latența la scrire/citire. Cererile de scrire/citire sunt procesate cu latență minimă.
- Scalabilitatea la citire/scrire. Cererile de citire/scriere pot fi împărțite pe mai multe noduri.
- Toleranța la defecte. Abilitatea de a deservi cereri de citire/scriere nu depinde de disponibilitatea unui nod particular.
- Persistenta datelor. Defectul nodurilor în masura unor limite nu duce la pierderea datelor.
- Consistența. Consistența este o proprietate mult mai complicată decât cele precedente.[1]

Consistența la *citire-scrire*. Din perspectiva citire/scriere, scopul cel mai de bază al unei baze de date este de a minimiza timpul de convergență al unei replici, adică timpul necesar pentru a propaga o actualizare la toate replicile și eventual să garanteze și consistența. Pe lângă aceste garanții slabe, putem fi interesați de garanții de consistență mai puternice:

- *Consistența citire după scriere*. Efectul unei operații de scriere pe un item X, va fi văzut întotdeauna ca o operație de citire succesivă pe X.
- *Consistența citire dupa citire*. Daca unii clienți citesc valoarea unui item X, orice citire succesivă pe X va returna întotdeauna aceeași valoare sau o valoare mai recentă.

Consistența la *scriere-scriere.* Conflictele de tip *scriere-scriere* apar în cazul partiționării bazei de date, deci o baza de date ar trebui ori să gestioneze aceste conflicte într-un mod anume, ori să garanteze ca scrierile concurente nu vor fi procesate de diferite partiții. Din această perspectivă, o bază de date poate oferi 2 modele de consistență:

Operații de *scriere atomice.* În cazul în care o bază de date oferă un API în care o cerere de scriere nu poate fi decât o asignare atomică și independentă de o valoare, un mod posibil mod de a evita conflictele scrie-scriere este de a alege "cea mai recentă" versiune a fiecărei entități. Acest lucru garantează că toate nodurile vor avea aceeași versiune a datelor, indiferent de ordinea de actualizări care pot fi afectate de erori și întârzieri de rețea. Versiunea datelelor pot fi specificată prin intermediul amprentelor de timp, numite și timestamp, sau folosind metrici specifice aplicației. Această abordare este folosită, de exemplu, în Cassandra.[1]

Operații de *citire-modificare-scriere atomice.* Aplicațiile realizează în mod obișnuit secvențe de *citire-modificare-scriere* în loc de scrieri atomice independente. În cazul în care doi clienți citesc aceeași versiune a datelor, o modifică și o scriu înapoi în același timp, cea mai recentă actualizare va suprascrie prima actualizare în modelul scrierilor atomice. Acest comportament poate fi semantic nepotrivit,de exemplu, în cazul în care ambii clienți adăugă o valoare într-o listă. O bază de date poate oferi cel puțin două soluții:[1]

- Prevenirea conflictelor. Operația de *citire-modificare-scriere* poate fi considerată ca un caz particular de tranzacție, astfel încât protocoalele de blocare distribuite sau protocoalele de consens reprezintă o soluție. Aceasta este o tehnică generică care poate suporta atât semantici atomice de *citire-modificare-scriere* și tranzacții izolate arbitrare. O abordare alternativă este prevenirea scrierilor distribuite concurente în întregime și rutarea tuturor scrierilor de un anumit articol de date pe un singur nod (principal sau ciob maestru la nivel mondial). Pentru a preveni conflictele, o bază de date trebuie să sacrifice disponibilitatea în caz de partiționare de rețea și să oprească toate partițiile exceptând o partiție. Această abordare este utilizată în multe sisteme cu garanții solide de consistență (de exemplu, cele mai multe RDBMS, HBase, MongoDB).
- Detecția conflictelor. O bază de date urmărește și ține evidența actualizărilor concurente ce duc la conflicte și ori fac rollback pentru una din actualizări sau păstrează ambele versiuni pentru rezolvarea pe partea de client. Actualizările concurente sunt de obicei urmărite prin utilizarea ceasurilor vector sau prin păstrarea întreagii istorii a versiunii. Această abordare este utilizată în sisteme precum Riak, Voldemort, CouchDB.

În continuare sunt prezentate tehnicile de replicare utilizate în mod obișnuit și o clasificare a acestora, în conformitate cu proprietățile descrise. Prima figura de mai jos Fig.1.1 prezintă relațiile logice dintre diferite tehnici și coordonatele lor în sistemul de compromisuri consistență-scalabilitate-disponibilitate-latență. Cea de a doua figură Fig1.2. ilustrează fiecare tehnică în detaliu.[1]

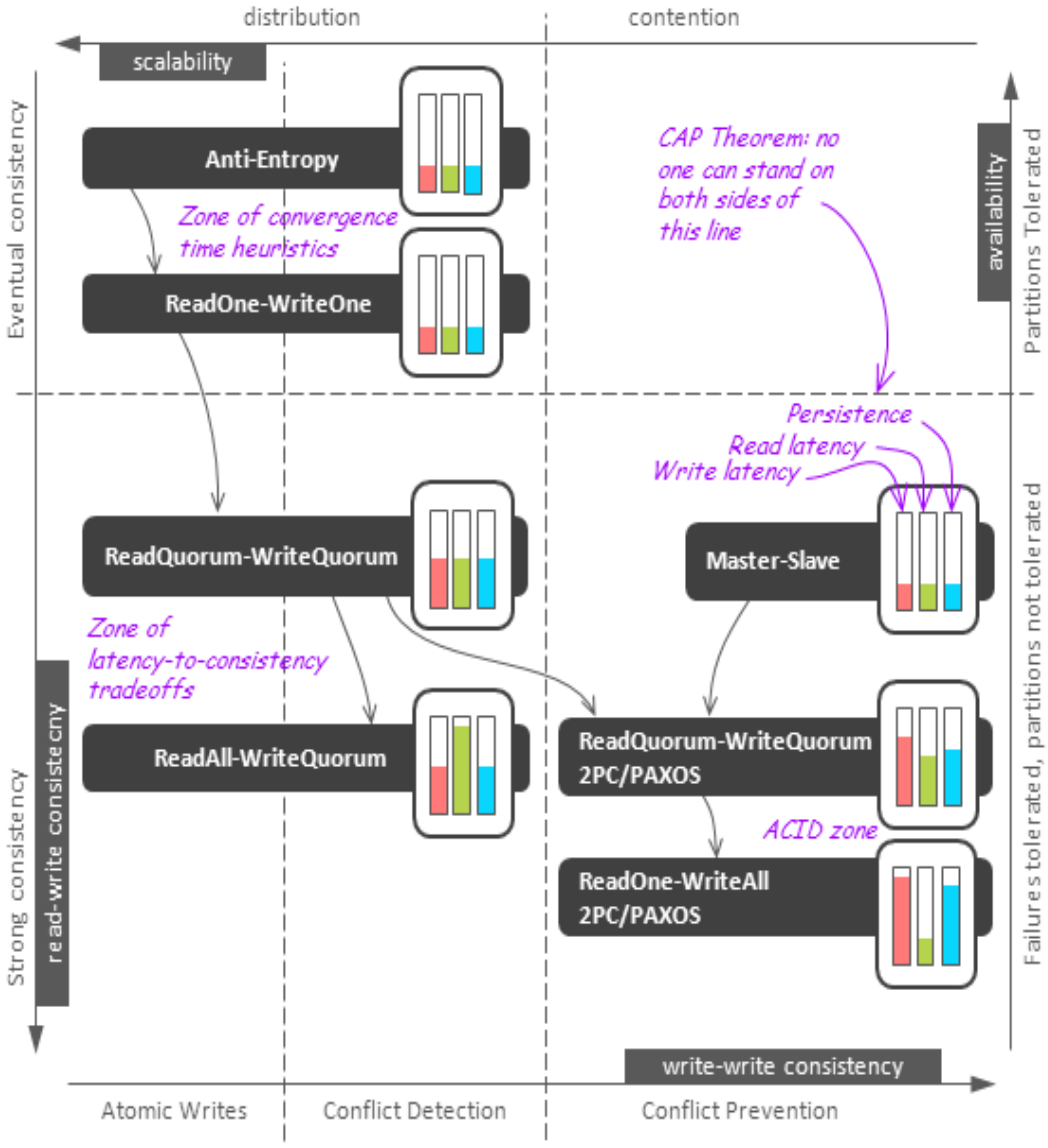

Fig.1.1 Relațiile logice dintre diferite tehnici de coordonare[1]

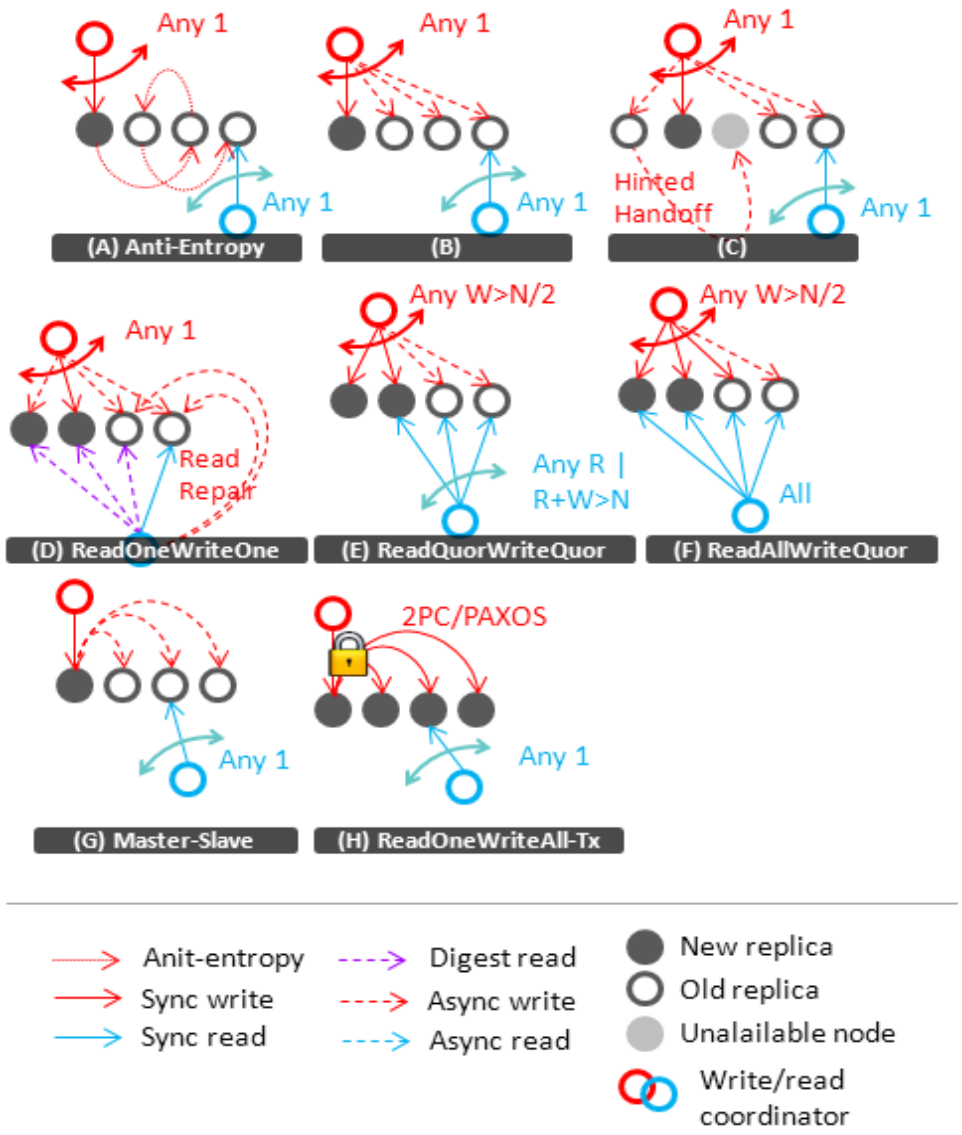

Fig.1.2 Factor de replicare 4. Se presupune că cordonatorul de citire/scriere poate fi fie un client extern ori un nod proxy în interiorul unei baze de date [1]

Parcurgând toate aceste tehnici deplasându-ne de la garanții de consistenta slabe la puternice:

A-Ant-entropy. Cele mai slabe garanțiile de consistență sunt furnizate de către această strategie. Cel ce efectuează scrierea actualizează orice copie arbitrar selectată. Cititorul citește orice replică și vede datele vechi până la o nouă versiune care este propagată prin intermediul protocolului de fond anti-entropie. Principalele proprietăți ale acestei abordări sunt:

- Propagarea la latență ridicată care îl face destul de impractic pentru sincronizarea datelor, de aceea este de obicei folosit doar ca un proces de fundal auxiliar care detectează și repară incoerențe neplanificate. Cu toate acestea, baze de date, cum ar fi Cassandra folosesc anti-entropia ca o modalitate primară pentru a propaga informații despre topologia bazei de date și alte metadate.
- Garanțiile de coerență sunt scazute: conflicte de scrie-scrie și discrepanțe de citire-scriere sunt foarte probabile chiar în absența eșecurilor.
- Disponibilitate superioară și robustețe împotriva partițiilor de rețea. Această schemă oferă o performanță bună pentru că actualizările individuale sunt înlocuite cu prelucrare lot asincron.
- Garanțiile de persistență sunt slabe, deoarece noile date sunt stocate inițial pe o singură replică.[1]

B. O îmbunătățire evidentă a schemei precedente este de a trimite o actualizare tuturor replicilor disponibile asincron de îndată ce o cerere de actualizare ajunge orice la o replică. Acesta poate fi considerat ca un fel de anti-entropie orientată.

- În comparație cu anti-entropie, aceasta îmbunătățește coerența cu o penalizare de performanță relativ mică. Cu toate acestea, garanțiile consistenței fomale și persistența rămân aceleași.
- Dacă o replică nu este disponibilă temporar ca urmare a eșecurilor de rețea sau eșecurilor/înlocuirii nodurilor, actualizările ar trebui să fie în cele din urmă livrate de către procesul de anti-entropie.

C. În schema anterioară, eșecurile pot fi manipulate mai bine folosind tehnica *hinted handoff*. Actualizările care sunt destinate nodurilor indisponibile sunt înregistrate pe coordonator sau oricare alt nod cu un indiciu că acestea ar trebui să fie livrate unui anumit nod de îndată ce va deveni disponibil. Acest lucru îmbunătățește garanțiile de persistență și timpii de convergență ai replici.[1]

D-Citeste unu Scrie unu. Având în vedere că transportatorul de *hinted handoffs* poate eșua înainte ca actualizărea amânată să se propage, este logic să se enforțeze consistența prin așa-numitele reparații de citit. Fiecare citire sau citiri aleatore declanșează un proces asincron care solicită un buletin, un fel de semnătură / hash al datelor solicitate de la toate replicile și reconciliază incoerențe dacă sunt detectate. Folosim termenul *ReadOne-WriteOne* pentru combinații de tehnici A, B, C și D - toate acestea nu oferă garanții stricte de consistență, dar sunt destul de eficiente pentru a fi utilizate în practică ca o abordare de sine stătătoare.[1]

E, Read Quorum Write Quorum. Strategiile de mai sus sunt îmbunătățiri euristice care scad timpi de convergență ai replicilor. Pentru a oferi garanții dincolo de eventuala coerență, trebuie să-și sacrifice disponibilitatea și să garanteze o suprapunere între seturile de scrie și citire. O

generalizare comună este de a scrie replici sincron W în loc de una și de a atinge R replici în timpul citirii.

- În primul rând, aceasta permite administrarea de garanții de persistență cu setari de W> 1.
- În al doilea rând, aceasta îmbunătățește coerența pentru R + W> N deoarece un set sincron de scris se va suprapune cu setul care este contactat în timpul citirii (în Fig. 1.2 W = 2, R = 3, N = 4), astfel încât cititorul va atinge cel puțin o replica proaspătă și o va selecta ca un rezultat. Aceasta garantează consistența dacă cererile de citire și scriere sunt emise secvențial dar nu garantează consistența globală de citire-după-citit. Să considerăm exemplul din figura de mai jos Fig1.3 pentru a vedea de ce citirile poate fi inconsistente. În acest exemplu R = 2, W = 2, N = 3. Cu toate acestea, scrierea a două replici nu este tranzacțională, astfel încât clienții pot primi ambele valori, vechi și noi în timp ce operația de scriere nu este finalizată:

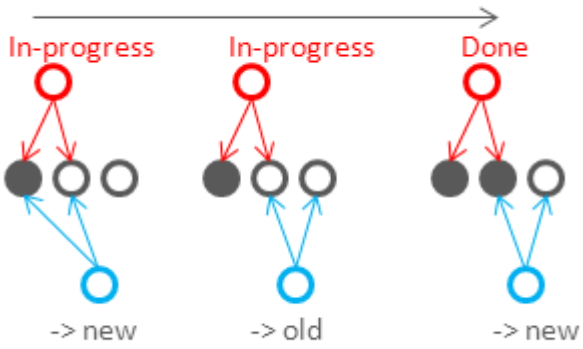

Fig.1.3 Citiri inconsistente[1]

- Diferite valori ale R și W permit schimbul din latență la scrie și persistenta la latența de citire și invers.
- Scrieri concurente pot scrie cvorumul disjuncte dacă W <= N / 2. Setarea W> N / 2 garantează detectare imediată a conflictului în operatiunea atomică citește-modifică-scrie cu modelul rollback.
- Strict vorbind, această schemă nu este tolerantă la partiții de rețea, deși tolerează eșecuri de noduri separate. În practică, euristici cum ar fi cvorum neglijent poate fi utilizat pentru a sacrifica consistența furnizată de o schemă standard de cvorum în favoarea disponibilitații în anumite situații.

F, Read All Write Quorum. Problema cu consistența citire-după-cititire pot fi atenuată prin contactarea tuturor replicilo în timpul citirii, cititorul poate prelua date sau poate verifica buletinele. Acest lucru asigură că o nouă versiune a datelor devine vizibilă pentru cititorii de îndată ce apare pe cel puțin un nod. Partițiile de rețea, desigur, poate duce la încălcarea acestei garanții.

G, Master-Slave. Tehnicile de mai sus sunt adesea folosite pentru a furniza fie scrieri atomice sau citire-modificare-scriere cu niveluri de consistență și detecție de conflicte. Pentru a atinge un nivel de prevenire a conflictelor, trebuie să folosească un fel de centralizare sau de blocare. O strategie simplă este de a utiliza replicarea asincron master-slave. Toate scrierile pentru un anumit articol de date sunt dirijate spre un nod central care execută operațiunile scrie secvențial. Acest lucru face în master un bottleneck, așa că devine crucială împărțirea datele în cioburi independente și să fie scalabile.

H, Transactional Read Quorum Write Quorum and Read One Write All. Abordarea cvorumului poate fi, de asemenea, consolidată prin tehnici de tranzacționare pentru a preveni conflictele pentru scriere-scriere. O abordare binecunoscută este de a utiliza comiterea cu protocol în două faze. Cu toate acestea, comitere în două faze nu este perfect sigură deoarece eșecul coordonatorului poate provoca blocarea resurselor. Protocolul de comitere Paxos este o alterative mai fiabilă, dar cu o penalizare pe partea de performanță.[1]

Plasarea datelor

Aceasta sectiune este dedicată algoritmilor care controlează plasarea de date în interiorul unei bazei de date distribuite. Acești algoritmi sunt responsabili pentru cartografierea între datele efective și nodurile fizice, migrarea datelor de la un nod la altul și alocarea globală a resurselor ca RAM de-a lungul bazei de date.[1]

Rebalansarea

Începem cu un protocol simplu care are ca scop să furnizeze migrarea datelor între nodurile clusterului fără întreruperi. Această sarcină apare în situații de expansiune de grup cum ar fi adaugarea de noi noduri, failover, sau reechilibrare (date devin distribuite inegal peste noduri). Să considerăm o situație care este reprezentată în secțiunea A din figura de mai jos - există trei noduri și fiecare nod conține o porțiune de date. Presupunem un model de date cheie-valoare, fără pierdere de generalitate, care este distribuit pe nodurile conform a unei politici de plasament al datelor arbitrar:[1]

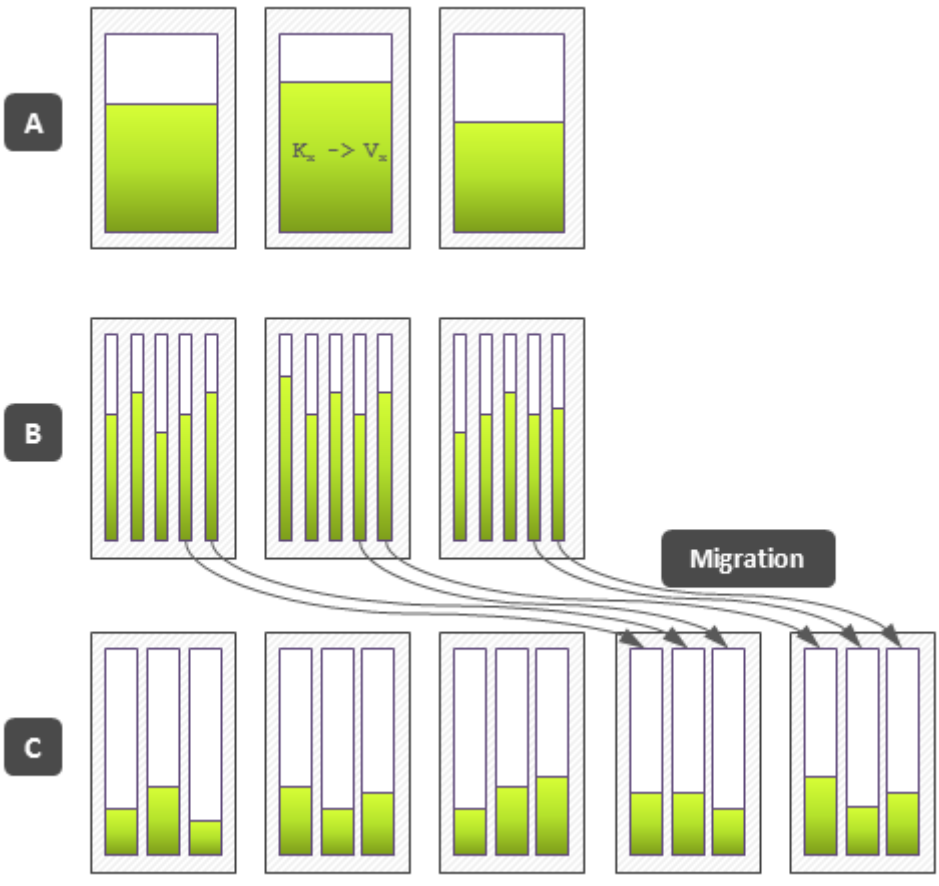

Fig.2.1 Migrarea datelor

Dacă nu avem o bază de date care realizează reechilibrarea datelor intern, probabil vom implementa mai multe instanțe ale bazei de date în fiecare nod așa cum se arată în secțiunea (B) din figura de mai sus. Acest lucru permite efectuarea unei expansiuni manuale a clusterului prin schimbarea unei instanță separate în starea off, copiind-o la un nod nou, și trecând-o în starea on, așa cum se arată în secțiunea (C). Deși o bază de date automată este capabilă de a urmări fiecare înregistrare în parte, multe sisteme, inclusiv MongoDB, Oracle Coherence și Redis Cluster folosesc tehnica descrisă pe plan intern, adică grupează înregistrările în shard-uri sau cioburi care sunt unități minime ale migrației folosite de pentru eficiență.[1]

Este destul de evident faptul că un număr de shard-uri ar trebui să fie destul de mare în comparație cu numărul de noduri pentru a asigura distribuirea uniformă a sarcinii. O migrare fără întrerupere a shard-urilor se poate face conform protocolului care redirecționează clientul de la nodul exportator la nodul care realizează importul în timpul unei migrări unui shard. Următoarea figură prezintă o mașină de stare pentru logica de get (key), așa cum va fi pusă în aplicare în Redis Cluster:[1]

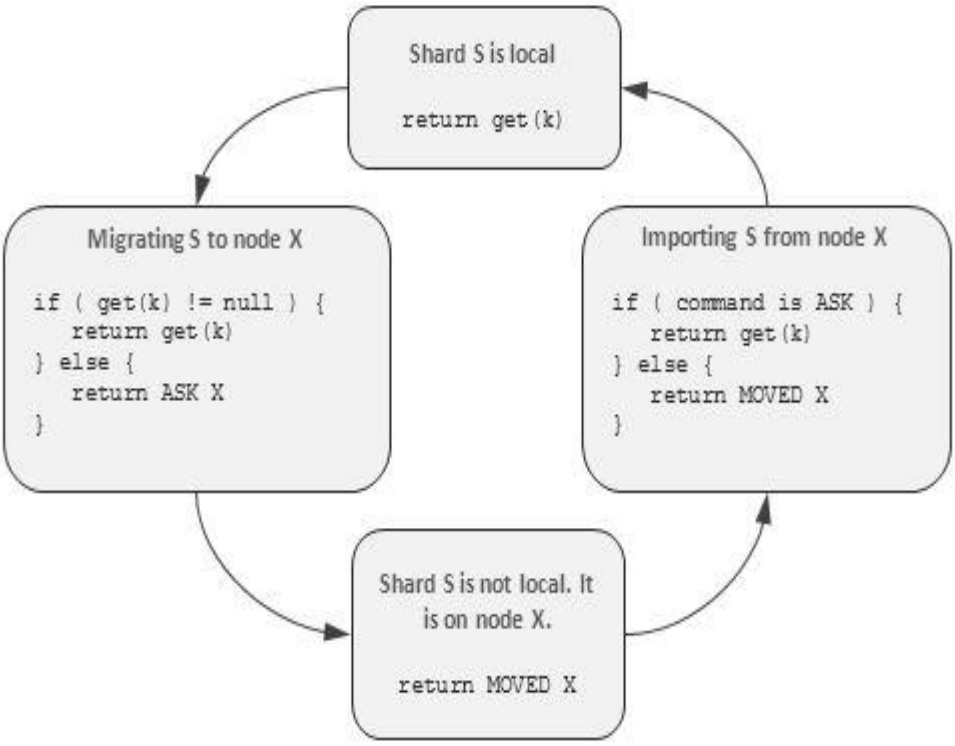

Fig.2.2 Redis Cluster

Se presupune că fiecare nod cunoaște o topologie de cluster și este capabil să mapeze orice key la un shard și un shard la un nod al cluster. Dacă nodul stabilește că cheia solicitată aparține unui shard local, atunci o caută la nivel local pătratul cel mai de sus în imaginea anterioară. Dacă nodul stabilește că cheia solicitată aparține unui alt nod X, atunci trimite o comandă de redirecționare permanentă clientului pătratul cel mai de jos în figura anterioară. Redirecționarea permanentă înseamnă că clientul este capabil să cache-uieze maparea între shard și nod. Dacă migrația shard-ului este în curs de desfășurare, nodurile care realizează exportul și nodurile care fac importul marchează acest shard în consecință și începe să mute datele sale blocare fiecare înregistrare în parte. Nodul exportator caută cheia local și în cazul în care nu a fost găsită redirecționează clientul la nodul care face importul presupunând că cheia este deja migrată. Aceasta redirecționare este realizată o singură dată și nu ar trebui să fie în cache. Nodul care relizează importul procesează redirecționarilela nivel local, dar interogările regulate sunt redirecționate permanent până cand migrația nu este finalizată.[1]

Sharding si replicarea în medii dinamice

Următoarea problemă abordată este modul în care se mapează înregistrările la nodurile fizice. O abordare simplă este de a avea o tabelă cu întervale de chei unde fiecare interval este atribuit unui nod sau de a utiliza proceduri, cum ar fi NodeID = hash (key)% TotalNodes. Cu toate acestea, hash-ing-ul pe bază de module nu adresează în mod explicit reconfigurarea clusterului pentru că adăugarea sau eliminarea nodurilor cauzează regruparea completă în întreagul cluster. Ca urmare, este dificil de manevrat replicarea și failover-ul.[1]

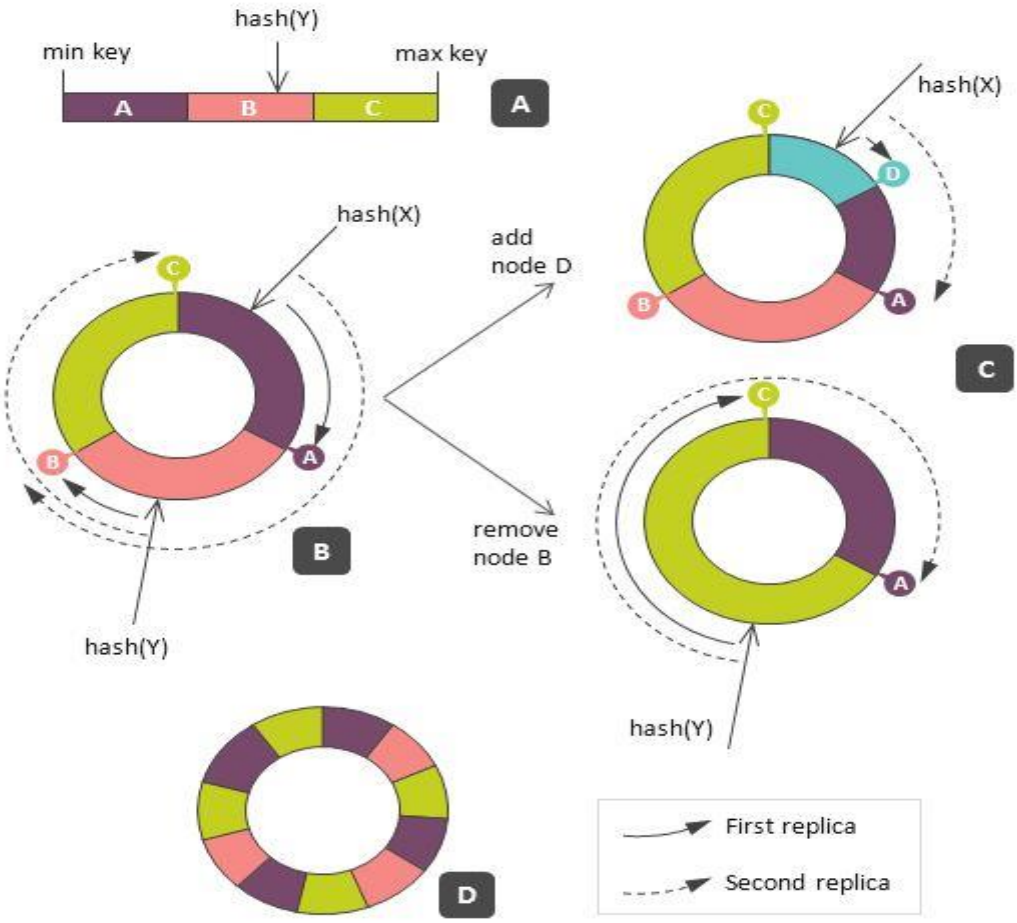

Fig.2.3 Hashing consistent

Hashing-ul consistent este de fapt o schemă de mapare pentru depozitul de key-value, adică mapează chei la nodurile fizice. Un spațiu de hash keys este un spațiu ordonat de stringuri binare de lungime fixă, așa că este destul de evident faptul că fiecare interval de chei este atribuit unui nod așa cum este descris în figura (A) pentru 3 noduri, și anume, A, B , și C. Pentru a face față replicarii, este convenabil să fie închis un spațiu sau interval de chei într-un inel și să fie traversat în sensul acelor de ceasornic până când toate replicile sunt mapate, așa cum se arată în figură (B). Cu alte cuvinte, punctul Y ar trebui să fie pus pe nodul B, deoarece cheia corespunde intervalului nodului B, prima replica ar trebui să fie pusă pe C, a doua replica pe A și așa mai departe.[1]

Avantajul acestei scheme este adaugarea și eliminarea eficientă a unui nod, deoarece provoacă reechilibrarea datelor numai în sectoarelele vecine. Așa cum se arată în figura (C), adăugarea nodului D afectează doar elementul X, dar nu și elemntul Y. În mod similar,

eliminarea sau eșecul nodului B afectează Y și replica lui X, dar nu X în sine. Cu toate acestea, așa cum a fost arătat în [4], partea întunecată a acestui beneficiu este vulnerabilitatea la suprasarcini - toată povara reechilibrarii este manipulată numai de vecini și ii face să replicheze volume mari de date. Această problemă poate fi atenuată prin maparea fiecarui nod nu doar la o singură gamă, ci la un set de intervale, cum se arată în figură (D). Acesta este un compromis - evită oblic în sarcini în timpul reechilibrării, dar păstrează tot efortul de reechilibrare rezonabil de scăzut în comparație cu cartografierea bazată pe module.[1]

Întreținerea unei viziuni complete și coerente a unui inel de hashing poate fi problematică în implementari foarte mari. Deși nu este o problemă tipică pentru bazele de date, din cauza clusturilor relativ mici, este interesant de studiat cum modul de plasare a datelor a fost combinat cu rutare de rețea în rețele peer-to-peer. [1]

Sharding pentru mai multe atribute

Deși hashing-ul consistent oferă o strategie eficientă de plasare a datelor atunci când datele sunt accesate prin intermediul cheiei primare, lucrurile devin mult mai complexe atunci când este necesară interogarea după mai multe atribute. O abordare simplă folosită în MongoDB este de a distribui date după o cheie primară indiferent de alte atribute. Ca rezultat, interogările care restricționează cheia primară pot fi dirijate spre un număr limitat de noduri, dar alte interogări trebuie sa fie prelucrate de toate nodurile din cluster. Acest lucru duce la următoarea problemă:

*Există un set de date și fiecare element are un set de atribute, împreună cu valorile lor. Există o strategie de plasare de date care limitează o serie de noduri care trebuie contactate pentru a procesa o interogare care limitează*o *un subset arbitrar al atributelor?*

O soluție posibilă a fost implementată în baza de date HyperDex. Ideea de bază este de a trata fiecare atribut ca o axă într-un spațiu multidimensional și a mapa blocuri în spațiu de noduri fizice. O interogare corespunde unui hiperplan care intersectează un subset de blocuri în spațiu, astfel încât numai acest subset de blocuri trebuie atins în timpul prelucrării interogării. Luăm în considerare următorul exemplu din [3]:

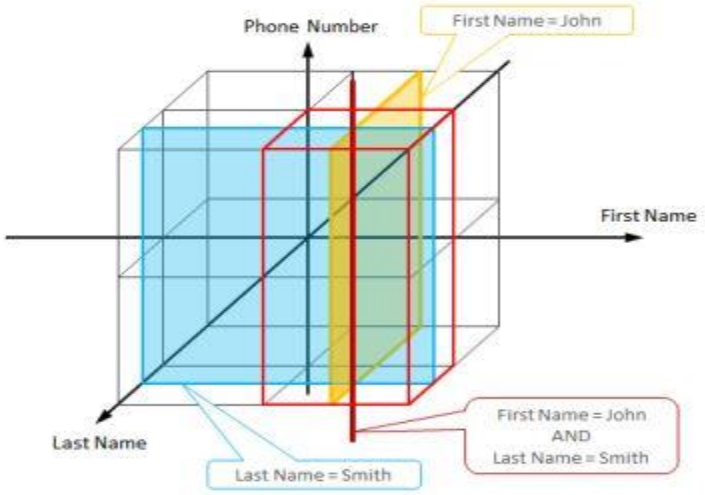

Fig.2.4 Fiecare atribut este tratat ca o axă într-un spațiu[1]

Fiecare element de date este un cont de utilizator care este atribuit prin nume, prenume, și număr de telefon. Aceste atribute sunt tratate ca un spațiu tridimensional și o posibilă strategie de plasare a date este de a mapa fiecare octant la un nod fizic dedicat. Interogări ca "prenume = Ioan" corespunde unui plan care intersectează 4 octant-uri, deci doar 4 noduri ar trebui implicate în prelucrare. Interogările care restricționează două atribute corespund la o linie care intersectează două octant-uri cum se arată în figura de mai sus, prin urmare, numai 2 noduri ar trebui implicate în prelucrare.[1]

Coordonarea sistemului

În această secțiune vom discuta despre o serie de tehnici care se referă la coordonarea sistemului. Coordonarea distribuită este un domeniu extrem de mare, care a fost un subiect de studiu intensiv pe parcursul mai multor decenii. În acest articol luăm în considerare doar o pereche de tehnici aplicate.[1]

Detectarea defectelor

Detectarea defecțiunilor este o componentă fundamentală a oricărui sistem distribuit tolerant la defecțiuni. Practic, toate protocoalele de detectare de eșecuri se bazează pe mesaje numite *heartbeat* care reprezintă un concept destul de simplu – componentele monitorizate trimit periodic un mesaj de *heartbeat* la procesul de monitorizare iar absența de mesaje *heartbeat* pentru o perioadă de timp îndelungată este interpretată ca un eșec sau defecțiune. Cu toate acestea, sisteme distribuite reale impun o serie de cerințe suplimentare, care trebuie să fie abordate:

1. **Adaptare automată**. Detectarea defecțiunilor trebuie să fie robustă pentru eșecurile și întârzierile temporare de rețea, schimbările dinamice în topologia de cluster, volumul de muncă sau lățimea de bandă. Aceasta este o problemă fundamental dificilă, deoarece nu există nici o modalitate de a distinge procesul eșuat față de unul lent[6]. Ca urmare, detectarea eșecului este întotdeauna un compromis între timpul de detectare a defecțiunilor (cât timp este nevoie pentru a detecta un eșec real) și probabilitatea de alarmă falsă. Parametrii acestui compromis trebuie ajustați în mod dinamic și automat.
2. **Flexibilitate**. La prima vedere, detectorul de eșecuri ar trebui să producă o ieșire de tip boolean, un proces monitorizat fiind considerat a fi fie viu sau moart. Cu toate acestea, se poate argumenta că ieșirea de tip boolean este insuficientă în practică. Să considerăm un exemplu din [5], care seamănă cu Hadoop MapReduce. Există o aplicație distribuită care constă dintr-un master și mai mulți slave sau lucrătorilor. Master-ul are o listă de job-uri și le prezintă slave-urilor. Master-ul poate distinge diferite "grade de eșec". În cazul în care masterul începe să suspecteze că unii lucrători au probleme, nu mai trimite noi job-uri către acei muncitori. Apoi, pe măsură ce timpul trece și nu există mesaje *heartbeat*, masterul înaintează joburile care se rulează pe acest lucrător la alti lucrători. În cele din urmă, master-ul devine complet încrezător că lucrătorul nu mai poate funcționa și eliberează toate resursele corespunzătoare.
3. **Scalabilitate și robustețe**. Detectarea defecțiunilor ca un proces de sistem ar trebui să scaleze odată cu sistemul. De asemenea, ar trebui să fie robust și consistent, adică toate nodurile din sistem ar trebui să aibă o viziune uniformă a proceselor din sistem care funcționează sau nu, chiar și în cazul unor probleme de comunicare.

O modalitate posibilă de a aborda primele două cerințe este așa-*numitul Phi Accrual Failure Detector* [5], care este utilizat cu unele modificări în Cassandra [8].

Cerința de scalabilitate poate fi abordată în mod semnificativ de zone de monitorizare organizate ierarhic care împiedică inundarea rețelei cu mesaje de *heartbeat* [7] și de sincronizare din diferite zone prin intermediul protocolului gossip (bârfă), sau printr-un depozit central tolerant la defecte. Această abordare este ilustrată mai jos (există două zone și toate cele șase detectoare de eșsec vorbesc între ele prin intermediul protocolului gossip sau prin depozit robust ca Zookeeper):[1]

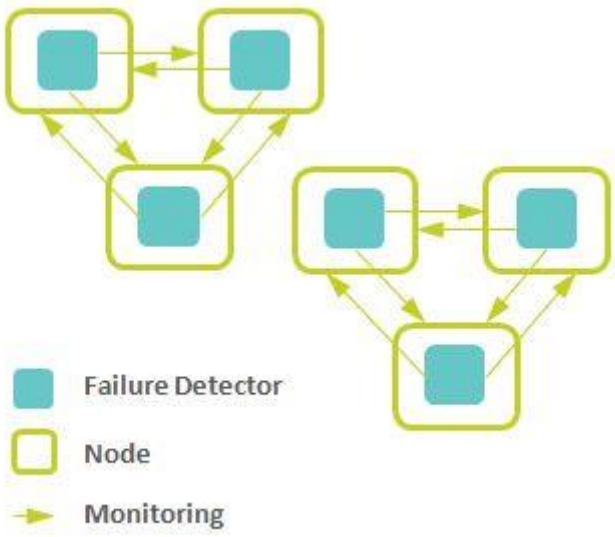

Fig.3.1 Monitorizarea clusterului[1]

Alegerea coordonatorului

Alegerea coordonatorului este o tehnică importantă pentru bazele de date cu garanții stricte de consistență. În primul rând, permite organizarea failover unui nod principal în sisteme master-slave. În al doilea rând, aceasta permite prevenirea conflictelelor la operații de scriere-scriere în cazul de partiții de rețea prin încheiere partițiilo care nu includ o majoritate de noduri.

Algoritmul *bully* (bătăuș) este o abordare relativ simplă la alegerea coordonatorului. MongoDB foloseste o versiune a acestui algoritm pentru a alege lideri în seturi replica. Ideea principală a algoritmului bătăuș este ca fiecare membru al clusterului se poate declara ca coordonator și să anunțe aceast lucru la alte noduri. Alte noduri poate accepta sau respinge această revendicare prin introducerea unui competitor pentru a fi un coordonator. Nod care nu se confruntă cu nici o dispută în continuare devine coordonator. Nodurile folosesc unele atribute

pentru a decide cine câștigă și cine pierde. Acest atribut poate fi un ID static sau o metrică recentă, cum ar fi ID-ul de la ultima tranzacție.[1]

Un exemplu de execuție algoritmului *bully* este prezentat în figura de mai jos. ID-ul static este folosit ca o metrică de comparație, un nod cu un ID mai mare câștigă.

1. Inițial sunt cinci noduri sunt în cluster și nodul 5 este un coordonator la nivel global acceptat.
2. Să presupunem că nodul 5 se defecteză și nodurile 3 respectiv 2 detectează această situație simultan. Ambele noduri începe procedura de alegere și trimit mesaje electorale la nodurile cu ID-uri mai mari.
3. Nodul 4 scoate nodurile 2 și 3 din concurs prin trimiterea OK. Nodul 3 elimină nodul 2.
4. Presupunem că nodul 1 detectează eșecul nodului 5 acum si trimite un mesaj electoral la toate nodurile cu ID-uri mai mari.
5. Nodurile 2, 3, și 4 elimină nodul 1.
6. Nodul 4 trimite un mesaj electoral la nodul 5.
7. Nod 5 nu răspunde, așa că nodul 4 se declară singur coordonator și anunță acest lucru la toate celelalte conexiuni.[1]

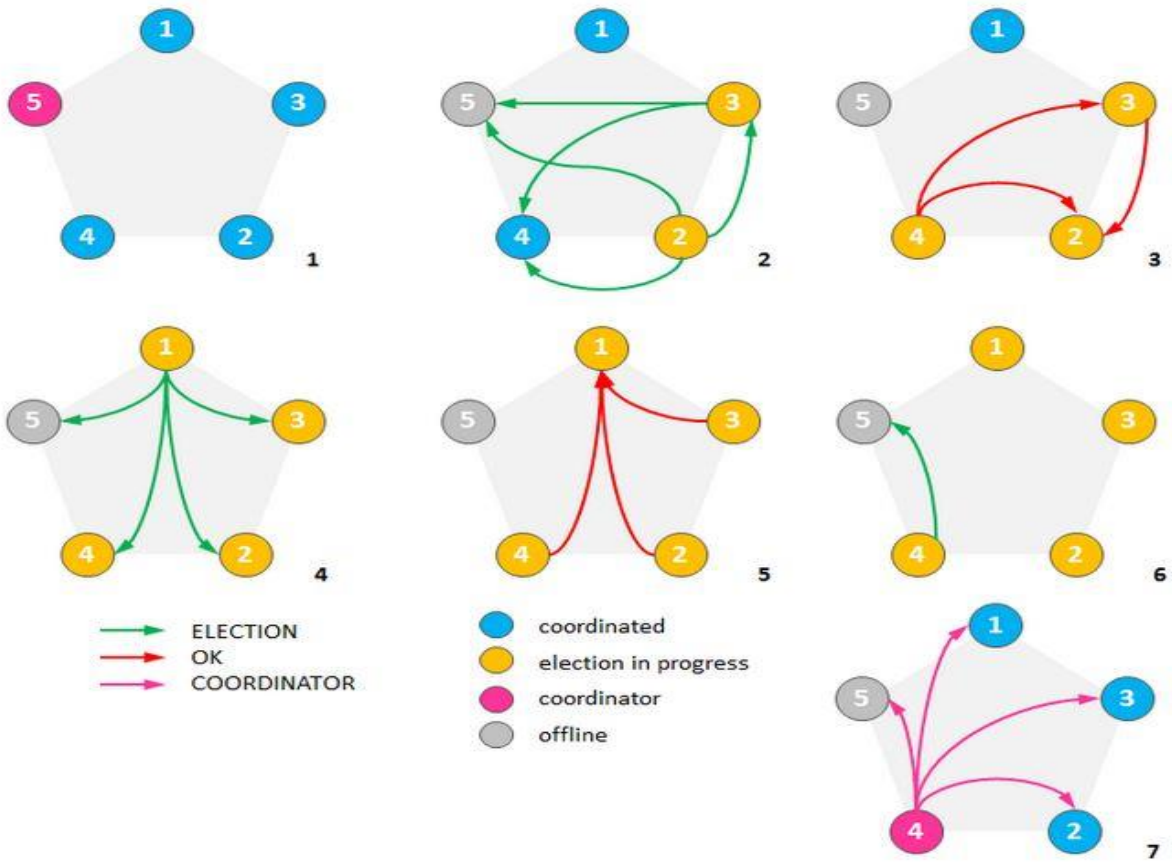

Fig.3.2 Algoritmul *bully* [1]

Concluzii

Mișcarea NoSQL a adus noi tehnici fundamental diferite de prelucrare a datelor distribuite, aceasta a declanșat o avalanșă de studii practice și studii reale cu diferite combinații de protocoale și algoritmi. Aceste dezvoltări au evidențiat treptat un sistem de blocuri de construcție de baze de date relevante cu eficiență dovedită practic.

Bibliografie

[1] https://highlyscalable.wordpress.com/2012/09/18/distributed-algorithms-in-nosql-databases/

[2] Sisteme de baze de date distribuite, Dorin Cârstoiu

[3] R. Escriva, B. Wong, E.G. Sirer. HyperDex: A Distributed, Searchable Key-Value Store

[4] G. DeCandia, et al. Dynamo: Amazon's Highly Available Key-value Store

[5] N. Hayashibara, X. Defago, R. Yared, T. Katayama. The Phi Accrual Failure Detector

[6] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process

[7] N. Hayashibara, A. Cherif, T. Katayama. Failure Detectors for Large-Scale Distributed Systems

[8] A. Lakshman, P.Malik. Cassandra – A Decentralized Structured Storage System