DBFarm: Un Cluster Scalabil pentru Baze de Date Multiple

(În măsură mare, traducere a articolului ”DBFarm: A Scalable Cluster for Multiple Databases”, Christian Plattner.a.o., ETH Zurich, 2006)

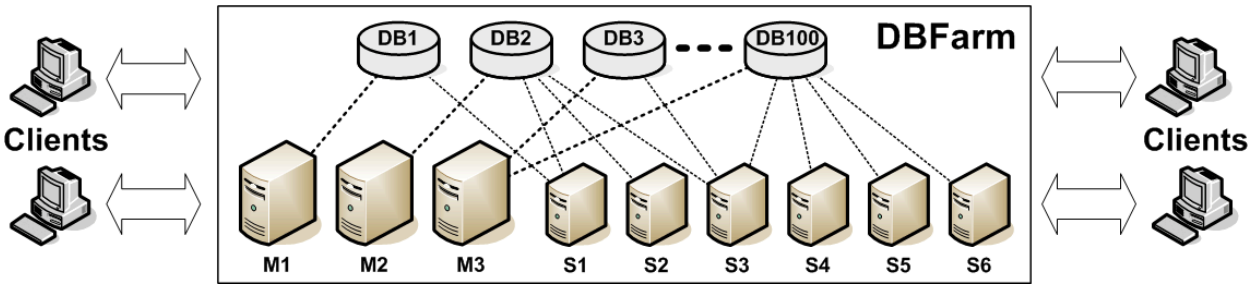
Cuprins

Introducere

In multe proiecte care au ca scop integrarea de aplicatii, systemul middleware a avut un rol major in asigurarea flexibilitatii si eficientei, din punct de vedere costuri, a sistemelor clusterelor. Exista (in piata) o multitudine de solutii middleware care pot asigura implementari distribuite si pot paraleliza/integra o serie intreaga de tipuri de aplicatii. Mai mult, o implementare de acest tip devine foarte eficienta in cazurile in care aceeasi platforma middleware poate fi folosita pentru integrarea mai multor aplicatii care ruleaza simultan/concurrent. Astfel, platforme ca J2EE sau .NET au fost special arhitecturate pentru a fi utilizate concurrent cu multiple alte aplicatii. Cu toate acestea, singura componenta a arhitecturii unei aplicatii care ramane, in cele mai multe cazuri, ca o solutie centralizata este Baza de Date. Chiar daca pana acum s-au inregistrat progrese semnificative in materie de clustere si baze de date paralele, accentul se pune intotdeauna pe imbunatatirea accesului la o baza de date unica. Aceasta abordare, a existentei unei singure instante optimizate, intra in contradictie cu principiul utilizarii mai multor instante ale bazei de date. Spre exemplu, un lucru primordial in exploatarea clusterelor il constituie abilitatea de a muta foarte usor resursele de la o masina la alta precum si alocarea rapida a masinilor in functie de incarcarea aplicatiilor care ruleaza. In cazul utilizarii unei singure baze de date, aplicatia-client se conecteaza pur si simplu la respective baza de date, pe cand intr-o arhitectura/solutie de tip cluster, aplicatiile-client se vor conecta la o singura baza de date chiar daca resursele sunt sau pot fi mutate dynamic in functie de incarcarea sistemelor. Astfel, utilizarea eficienta a resurselor unui cluster implica/presupune ca instanta DB-ului sa permita distributia catre mai multe noduri de calcul. In cazul in care exista probleme in utilizarea unei instante, celelalte nu vor fi impactate negative de acest incident. Nu in ultimul rand, aplicatiile-client trebuie sa foloseasca intotdeauna o baza de date consistenta, iar aceasta trebuie asigurata fara limitari ale scalabilitatii clusterului si fara incarcarea inutila a fluxurilor de transfer de date (overhead).

Toate aceste limitari descrise mai sus, sublinieaza necesitatea solutiilor middleware, din moment ce optimizarile la nivelul bazelor de date s-au concentrat doar asupra unei singure instante, de a implementa/facilita accesul la mai multe instante independente ale bazei de date in cadrul aceleasi arhitecturi. Astfel, marea provocare o constituie nelimitarea din punct de vedere al scalabilitatii solutiei.

Exemplu: Operarea a 100 de baze de date, fiecare ruland 100 de tranzactii/secunda, presupune procesarea rapida a informatiilor, obiectivul fiind ca Solutia sa fie chiar mai scalabila de atat, sistemul trebuind sa garanteze consistenta si o singura interfata pentru aplicatiile-client.



In acest material/document ne propunem sa descriem arhitectura si modul de implementare al unui DB Farm, respectiv a unui server de baze de date de tip cluster utilizat ca mediu support pentru cateva sute de instante ale bazei de date.

Un DB Farm este bazat pe 2 tipuri de servere de baze de date:

1. un server de tip master (conform figurii 1)
2. mai multe servere satelit.

Serverele master pot fi extrem de fiabile si cu performante deosebite prin utilizarea de echipamente hardware specializate, sisteme RAID, folosirea de metode de redundanta sophisticate (hot standby) si a strategiilor de backup.

O astfel de abordare asigura fiabilitatea necesara arhitecturii software unde resursele sunt utilizate de catre toate sistemele de baze de date. Scalabilitatea devine posibila cu ajutorul unui clusterului care furnizeaza sistemelor satelit copii ale bazei de date.

Un aspect important al unui DBFarm il constituie faptul ca utilizatorii sistemelor/aplicatiilor care ruleaza pe o astfel de arhitectura nu vor sti daca la un moment dat interactioneaza cu o “copie” a bazei de date sau chiar cu baza de date a serverului master, sistemul oferind accces intotdeauna la un DB consistent/solid.

O alta functionalitate cheie a unui DBFarm o constituie distribuirea puterii de procesare, dintre baza de date master si copiile acesteia de la nivelul serverelor satelit, bazata pe caracteristicile de citire/scriere (read/write) ale operatiunilor executate de catre utilizatorii aplicatiilor. Asfel, operatiunile de scriere sunt executate la nivelul bazei de date master iar cele de citire la nivelul “copiilor” bazelor de date de la nivelul serverelor satelit. Aceasta distribuire a puterii de procesare permite reducerea semnificativa a “incarcarii” sistemului master (existand posibilitatea ca sistemul sa permita un numar mult mai mare de baze de date pe aceeasi masina) in timp ce se asigura un nivel foarte ridicat de paralelizare a serverelor satelit (constituie bazele scalabilitatii).

Aceasta alocare dinamica a resurselor reflecta caracteristica incarcarilor bazei de date, cand operatiunile de update a datelor existente reprezinta operatii minore cu localizare precisa, in timp ce tranzactiile de tip read-only impacteaza mult mai mult operatiile de tip input/output (pentru extragerea de date si indexare) cauzand un volum mare de date tranzactionate (pentru operatii de tip join, order, average, sau group by).

Avantajele acestei abordari sunt confirmate prin efectuarea de numeroase teste si experimente. Am demonstrat cum un DB Farm ofera o performanta mult mai stabila si mai scalabila (atat in ceea ce priveste timpii de raspuns cat si a transferului de date) pentru o configuratie de pana la 300 TPC-W sisteme de baze de date prin comparatie cu instalarea unui singur server de baze de date. De asemenea, s-a demonstrat performanta unui DBFarm folosind sistemul de referinta RUBBoS (baze de date foarte extinse cu tranzactionari/operatii foarte complexe) si s-a aratat modul in care se pot stabili prioritati individuale ale bazelor de date, astfel incat acestea sa obtina o mai buna performanta decat altele.

Din punctul de vedere al aplicatiilor, un DBFarm poate fi utilizat intr-o varietate foarte mare de configuratii/setari. Astfel, se poate folosi pentru a transforma un cluster de masini intr-un serviciu

al DB-ului care poate rula intr-o retea LAN (companie, universitate, etc), asigurand astfel mentenanta si administrarea bazelor de date de la nivelul masinilor pe care acestea ruleaza.

De asemenea, se poate folosi la implementarea de servicii ale bazei de date ca parte din Application Service Provider, companiile mici si medii putand utiliza astfel bazele de date proprii ca parte a unui DB Farm pus la dispozitie de un furnizor specializat. Folosirea in comun a resurselor in cadrul unui DB Farm, reprezinta o solutie foarte eficienta care asigura, prin scalabilitatea pe care o ofera, accesul unui numar foarte mare de utilizatori.

Solutia oferita propune o abordare noua in ceea ce priveste replicarea si incarcarea datelor, care se pliaza foarte bine pe arhitecturile moderen, cum ar fi serverele web.

Spre deosebire de multe solutii existente, aceasta ofera consistenta fara a intampina probleme de performanta, DBFarm nepresupunand solutii software si hardware complexe (exemplu: infrastructure IT complexe/specializate, harddisck-uri share-uite) sau care necesita modificarea aplicatiilor existente. Solutia include, de asemenea, algoritmi inovatori de rutare a tranzactiilor pe un cluster de baze de date, evitand astfel limitarile cu care se confrunta solutiile de replicarea a bazelor de date.

Arhitectura

Distribuirea incarcarii datelor

Pentru a aputea asigura scalabilitatea bazelor de date, incarcarea trebuie distribuita catre mai multe masini/servere, aceasta constituind principal problema in cazul bazelor de date paralele, replicate si distribuite. Intr-un system DBFarm, provocarea consta in faptul ca distribuirea incarcarii sistemului trebuie sa se intample la nivelul bazelor de date asigurandu-se in acelasi timp si consecventa acesteia. Exista multe modalitati de a pune in aplicare distribuirea incarcarii la nivelul unei singure instante:

a. folosind controlul versionarilor si a concurentei la nivelul sistemului middleware pentru a ruta tranzactiile catre masini determinate/corespunzatoare.
b. Se bazeaza pe serverele-client pentru a produce o incarcare bine echilibrata la nivelul cluserului, diferitele sale noduri fiind sincronizate folosind group communication primitives.
c. Presupune ca schema bazei de date a fost pre-partajata in ceea ce se numeste conflict clases, ce vor fi folosite ulterior la distribuirea incarcarii
d. Se solicita serverelor-client sa puna la dispozitie volumul de date noi pentru a le putea redirectiona catre diferitele noduri ale cluster-ului.

Din pacate, niciuna dintre aceste metode nu este fezabila in cazul unui cluster multi-instanta. Duplicarea controlului concurentei la nivelul middleware-ului va rezulta intr-o utilizare limitata per fiecare tranzactie (in afara de problema complexa a mentenantei, deoarece middleware-ul ar trebui sa intretina controlul versiunilor si schemele de informatii pentru sute de instante). Distributia

incarcarii si asigurarea consistentei prin metoda Group communication iese din discutie din cauza tranzactiilor effectuate la nivelul fiecarui server-client si a sincronizarii implicite a nodurilor. Similar, utilizarea conflict classes presupune parsarea tranzactiilor de input pentru a identifica care dintre clasele respective va fi utilizata, fapt ce determina limitarea imediata a numarului de tranzactii pe care middleware-ul le poate ruta per secunda.

Metoda de distribuire a incarcarii folosita intr-un system de tip DBFarm constituie consecinta directa a tuturor restrictiilor enumerate mai sus. Pentru evitarea acestora, metoda propusa face distinctie intre tranzactiile de tip readonly si cele de update de date. Tranzactiile de update sunt cele in urma carora baza de date se updateaza, in timp ce tranzactiile readonly sunt cele care nu schimba starea datelor existente. O asemenea distribuire a incarcarii datelor poate fi implementata efficient la nivelul middleware-ului fara a fi necesara parsarea tranzactiilor si fara a mentine informatiile pe schema fiecarei instante a bazei de date de la nivelul cluster-ului.

Avantejele distribuirii:

1. Update-ul tranzactiilor determina statusul fiecarei instante si asigura corectitudinea datelor, fiind necesar doar fluxul de tranzactii pentru aceste determinari.
2. Tranzactiile readonly sunt folosite pentru asigurarea scalabilitatii prin rerutarea lor catre diferite noduri de calcul ale cluster-ului. Acest lucru a avut la baza modelele de referinta in materie de baze de date (exemplu: TPC-W sau RUBBoS), care spun ca incarcarea datelor este determinata, in cele mai multe cazuri, de tranzactii readonly (cel putin 50% dintre tranzactii)

Aceasta metoda de distribuirea a incarcarii sistemelor de baze de date, bazate pe tranzactiile de tip read/write, determina scalabilitatea sistemului, care poate fi limitat numai de proportia tranzactiilor read si write. Astfel, un DBFarm nu va asigura scalabilitatea bazei de date atunci cand proportia de tranzactii update este de 100% (asa cum o astfel de incarcare nu asigura scalabilitatea nici in cazul bazelor de date replicate, aceasta afectand serios engine-urile unui system de baza de date paralele). Dar, asa cum am aratat mai sus, intrucat majoritatea tranzactiilor de la nivelul bazelor de date sunt de tip readonly, un system DBFarm asigura o scalabilitate semnificativa.

Serverul master si satelitii

Ca o prima consecinta a folosirii metodei de distribuire a incarcarii unui system de baze de date, DBFarm adopta strategia lazy propagation. Copia principal a unei instante se numeste baza de date Master, acestea fiind localizate intotdeauna la nivelul unui master server. Un astfel de server proceseaza toate tranzactiile de update ale tuturor bazelor de date master pe care le gazduieste, fiind astfel updatat la zi (up-to-date).

Fiecare baza de date master este responsabila pentru mentinerea corectitudinii datelor proprii, folosind pentru asta propriile sisteme de control al tranzactiilor concurente si a mecanismelor de refacere a datelor (recovery mechanism).

Copiile bazelor de date master (asa numitele copii satelitare) sunt pozitionate la nivelul serverelor-satelit, care au rolul de a executa doar tranzactiile de tip readonly, astfel ca consistenta continutului datelor este dictat de catre serverele-master.

Tinand cont de aceste lucruri, repunerea in functiune (recovery) a serverelor-satelit cazute si transferul copiilor noi ale bazei de date nu reprezinta o problema complexa in cadrul unui system de tip DBFarm (spre deosebire de celelalte arhitecturi unde crearea sau stergerea unei copii implica operatii de sincronizare foarte complexe).

In plus, chiar baza de date master reprezinta solutia de siguranta, chiar si atunci cand toate celelalte copii sunt cazute, serverul-master asigura utilizatorilor posibilitatea a-si continua munca avand la dispozitie o versiune actualizata si consistenta a datelor (bineinteles, cu o reducere a performantei utilizarii sistemului).

In general, se presupune ca sistemele master sunt computer de dimensiuni considerabile ce au la dispozitie resurse suficiente (memorie si capacitate pe disc) pentru a permite rularea unui numar mare de baza de date. De asemenea, unor astfel de sisteme li se asigura disponibilitate foarte mare, asigurata atat de componentele hardware cat si de cele software. Capacitatea unui system master va determina numarul de baze de date care poate fi suportat si care este limita in ceea ce priveste totalul incarcarii tranzactiilor.

Un system DBFarm ajunge la limita superioara doar daca toate sistemele master ating limita proprie de procesare a fluxurilor de tranzactii de tip update. In acel moment, si doar daca tranzactiile viitoare nu vor fi de tip readonly, sistemele-client nu pot creste rata de transmitere a tranzactiilor atata timp cat viteza de procesarea a update-urilor este determinata de sistemul master care, la saturatie, nu mai dispune de capacitate de procesare. Scalabilitatea poate fi, in acest caz, crescuta usor prin adaugarea mai multor sisteme master si redistribuirea bazelor de date catre aceste noi masini.

Serverele-satelit pot fi mai putin puternice decat cele master si, de asemenea, nu foarte reliable, la acest nivel putand exista un numar arbitrar de copii ale bazei de date. Scalabilitatea unui singure baze date master este asigurata de existenta copiilor acesteia la nivelul multor sisteme satelit. Limita scalabilitatii, pe langa limita impusa de proportia celor 2 tipuri de tranzactii, este atinsa cand fiecare tranzactie readonly concurenta se executa la nivelul propriului server-satelit. Mai mult de atat, copiile aditionale vor ramane inactive.

Un asemenea grad extrem de distributie a datelor poate interveni in cazul unui cluster, care nu are pierderi de eficienta, prin plasarea copiilor instantelor la nivelul aceluiasi satelit. Din moment ce satelitii pot fi folositi de catre diverse copii ale bazei de date, nicio resursa nu este folosita deficitar, intrucat fiecare copie executa o singura tranzactie de tip readonly. La cealalta extrema, intr-un system DBFarm exista posibilitatea pentru o instant de a rula la nivel centralizat, fara a avea nicio alta copie. In astfel de cazuri, sistemul master proceseaza atat fluxurile de tranzactii readonly cat si pe cele de update.

Un avantaj al DBFarm il constituie faptul ca asigura o solutie fiabila pentru aplicatiile care utilizeaza bazele de date. Functiile, triggeri, procedurile stocate, etc pot fi definite la nivelul serverului master si nu necesita duplicarea la nivelul serverelor-satelit. Majoritatea solutiilor de replicare a bazelor de date reclama astfel de functionalitati, dar acestea ori nu sunt folosite (exemplu: atunci cand controlul verionarilor si al concurentei se manifesta la nivelul middleware-ului) ori trebuie replicate la nivelul tuturor copiilor – lucru nu tocmai fezabil in cazul trigger-ilor (exemplu: atunci cand update-urile sunt facute prin group communication).

Planificarea tranzactiilor in DBFarm

In timp ce principiul separarii incarcarii datelor, in cazul DBFArm, ofera posibilitatea scalabilitatii si ofera garantii ca baza de date master este intotdeauna la zi (up-to-date), nu asigura accesul consistent la date atunci cand sunt accesate copii ale bazei de date. Astfel, atata timp cat nu sunt luate alte masuri, consistenta datelor privita din punctual de vedere al serverului-client va depinde de cum incarcarea tranzactiilor generate de client sunt impartite in fluxuri de tip readonly si update si de cum aceste fluxuri sunt transmise catre diferitele masini ale cluster-ului.

Din perspectiva clientului, se pune accent doar pe *strong session serializability*, metoda prin care fiecare client are acces intotdeauna la propriile update-uri. Cu alte cuvinte, o tranzactie readonly trebuie sa vada nu numai propriul status actualizat dar si toate update-urile inregistrate de acel client. Acest fapt previne clinetul de experienta *calatoriei in timp* (*travel-in-time*) cand o interogare a bazei de date intoarce date corecte dar vechi.

Desi principiul *strong session serializability* ar trebui sa fie sufficient, abordarea DBFarm merge un pic mai departe, asigurandu-se ca o tranzactie readonly executata la nivelul unei copii va vedea toate update-urile executate la nivelul bazei de date master pana la momentul cand tranzactia readonly intra in system. In acest sens, sistemul DB Farm devine efectiv transparent pentru serverele-client din moment ce acestea vor citi exact ceea s-ar fi citit utilizand un singur server de baze de date.

Pentru simplitatea expunerii si pentru a pastra un nivel cat mai general, vom descrie detaliile planificarii tranzactiilor intr-un system DBFarm ce foloseste un singur server de baze de date master.

Atata timp cat serverele-client comunica doar cu cele master, acestea nu sunt vizibile la nivelul clientilor. Prin urmare, tranzactiile de intrare de tip readonly trebuie sa fie transmise de catre master catre sateliti intr-un mod prin care sa se asigure consistenta datelor.

Tranzactiile de tip update de la nivelul unei anumite instante a bazei de date sunt executate local pe baze de date a sistemului master. Rezultatul constituie, conceptual vorbind, o serie de statusuri actualizate ale fiecarei baze de date care contin update-urile tuturor tranzactiilor inregistrate pana in acel moment. Sistemul master profita de acest lucru prin identificarea writeset-urilor tranzactiilor de update in ordinea in care acestea au fost inregistrate, un writeset constituind o colectie a modificarilor (exemplu: “ID” si “Valoare noua”).

Dupa aceste operatii, sistemul master transmite writeset-urile catre toti satelitii implicati, folosindu-se de cozile de tip FIFO (first in – first out). Apoi, satelitii proceseaza aceste seturi de date in ordinea in care le-au receptionat.

In acest fel se poate observa si demonstra cu usurinta ca, daca executarea tranzactiilor la nivelul bazei de date master a fost efectuata cu success, aplicarea modificarilor pe o copie a bazei de date in ordinea inregistratrilor stabilita de master, garanteaza ca respective copie va trece prin acelasi process de update ca si masterful.

Conceptual, procesul din DBFarm este:

Pentru fiecare tranzactie inregistrata cu success, master-ul trimite catre client un mesaj de confirmare (acknowledgment message) – cel mai nou mesaj trimis reflecta ultimul status actualizat pe care orice client il poate vedea in momentul in care inregistreaza urmatoarea tranzactie

Prin urmare, cand DBFarm master trimite un mesaj de confirmare unui server-client care a executat tranzactia Tk, toate tranzactiile viitoare de tip readonly ale oricarui client vor vedea statusul actualizat determinat de writeset-ul W Sk (writeset-ul care contine modificarile optinute prin rularea tranzactiei Tk)

Astfel, din moment ce o cerere pentru o tranzactie readonly ajunge in sistemul master, acesta poate deduce (prin urmarirea tuturor mesajelor de confirmare) stadiul minim al bazei de date pe care tranzactia de tip readonly o poate observa.

De asemenea, managementul transmiterii mesajelor de confirmare si asigurarea consistentei tranzactiilor readonly trebuie realizat de catre sistemul master pentru fiecare baza de date, in mod separat.

Pentru a ne asigura ca o tranzactie readonly nu are acces la date invechite ar trebui sa fie blocata la nivelul master-ului si apoi acesta sa transmita toate operatiile sale catre satelitii tinta, numai dupa ce raporteaza cu success aplicarea tuturor writeset-urilor.

Aceasta abordare are, insa, si unele dezavantaje:

Logica aditionala, fluxuri comunicationale si complexitate sporita

La nivelul master-ului trebuie implementate cozi de mesaje

Timpul de procesare crescut ca urmare a transmiterii mesajelor de confirmare, fapt ce poate determina blocarea inutila a tranzactiilor de tip readonly.

In realitate, modul in care sistemul functioneaza atinge aceleasi rezultate dar fara a bloca tranzactiile, mai mult decat trebuie, si fara a impune o incarcare a serverului master.

Solutia se bazeaza pe un mechanism de etichetare (tagging mechanism), care, de fiecare data cand tranzactie este inregistrata, sistemul master nu numai ca extrage writeset-urile, dar si incrementeaza automat tranzactia care urmeaza a fi transmisa.

Vom denumi acest numar incrementat automat ca “the change number for database DB”.

Nota: de retinut faptul ca acest numar nu este specific sistemului-client, dar se aplica tuturor tranzactiilor executate la nivelul bazei de date.

Algorithm 1: Master Transaction Handling

```
1:  INPUT DB: Database Name
2:  INPUT T: Incoming Transaction
3:  INPUT M: Mode of T, M ∈ {Read-Only, Update}

4:  if M == 'Update' then
5:      /* T must be executed on the master */
6:      Start a local update transaction in database DB
7:      for all Incoming statements S in T do
8:          if S == 'ROLLBACK' then
9:              Abort the local transaction
10:             Send abort response back to client
11:             RETURN
12:         end if
13:         if S == 'COMMIT' then
14:             ENTER CRITICAL SECTION
15:             Commit the local update transaction
16:             if Commit operation failed then
17:                 LEAVE CRITICAL SECTION
18:                 Abort the local update transaction
19:                 Send error response back to client
20:                 RETURN
21:             end if
22:             Determine T's commit number CN
23:             Set CUR-CN(DB) := CN
24:             LEAVE CRITICAL SECTION
25:             Send successful commit reply to client
26:             Retrieve T's encoded writeset WS from DB
27:             Forward (DB, WS, CN) to all satellites that
                host a copy of database DB
28:             RETURN
29:         end if
30:         Execute S locally
31:         if Execution of S fails then
32:             Abort the local transaction
33:             Send error response back to client
34:             RETURN
35:         end if
36:         Send result of S back to client
37:     end for
38: end if
39: /* T is Read-Only, try to execute on a satellite */
40: if ∃ satellite N with a copy of DB then
41:     Use load balancing to select a satellite N
42:     MIN-CN := CUR-CN(DB)
43:     Forward (DB, T, MIN-CN) to N
44:     Relay all incoming statements S in T to N
45:     RETURN
46: end if
47: /* Need to execute T locally */
48: Start a local read-only transaction in database DB
49: for all Incoming statements S in T do
50:     if S ∈ {COMMIT, ROLLBACK} then
51:         Commit the local read-only transaction
52:         Send response back to client
53:         RETURN
54:     end if
55:     Execute S locally
56:     if Execution of S fails then
57:         Abort the local read-only transaction
58:         Send error response back to client
59:         RETURN
60:     end if
61:     Send result of S back to client
62: end for
```

Algorithm 2: Satellite Transaction Handling

```
1:  INPUT DB: Database Name
2:  INPUT T: Incoming Transaction
3:  INPUT MIN-CN: Min. committed State needed to exe-
    cute T

4:  Wait until CUR-CN(DB) ≥ MIN-CN
5:  /* Execute T locally in read-only mode */
6:  Start a local read-only transaction in database DB
7:  for all Incoming statements S in T do
8:      if S ∈ {COMMIT, ROLLBACK} then
9:          Commit the local read-only transaction
10:         Send response back to client
11:         RETURN
12:     end if
13:     Execute S locally
14:     if Execution of S fails then
15:         Abort the local read-only transaction
16:         Send error response back to client
17:         RETURN
18:     end if
19:     Send result of S back to client
20: end for
```

Algorithm 3: Satellite Writeset Application

```
1:  INPUT DB: Database Name
2:  INPUT WS: Writeset
3:  INPUT CN: Committed State produced by WS

4:  Turn WS into a set of SQL statements
5:  Wait until CUR-CN(DB) == (CN-1)
6:  Start a local update transaction in database DB
7:  Apply the produced SQL statements
8:  Commit the local transaction
9:  if Application of WS failed then
10:     Report ERROR to the master
11:     Disable further processing for database DB
12:     RETURN
13: end if
14: Set CUR-CN(DB) := CN
```

In momentul in care se reruteaza o tranzactie readonly catre sistemele-sateliti, masterful eticheteaza inceputul acelei operatii cu numarul incrementat automat (change number) corespunzator respectivei Baze de date. Acest numar este de asemenea mentinut pentru toate copiile ale bazei de date DB.

Un system-satelit care receptioneaza o eticheta a unei tranzactii readonly va incepe executia acesteia doar dupa ce ii aplica writeset-ul corespunzator. In acest mod ne asiguram ca o tranzactie readonly vede toate update-urile inregistrate pana in acel moment, nu numai ale unui system-client specific. Blocarea unei tranzactii readonly este delegate, prin urmare, catre sistemele-satelit, unde poate fi manipulate efficient.

Este important de mentionat faptul ca, in afara de verificarea ca fiecare tranzactie readonly vede intotdeana cel mai recent status al datelor, nu este necesar controlul concurentei in afara bazei de date si, prin urmare, planificarea tranzactiilor presupune un timp de procesare redus.

Detaliile algoritmilor folositi in DBFarm pentru manipularea tranzactiilor la nivelul sistemului master si a satelitilor, la fel ca si utilizarea writeset-urilor, sunt dati in algoritmii 1, 2 si 3.

1. Algoritmul 1 descrie rutarea la nivelul sistemului master.
2. Liniile 4-38 manipuleaza tranzactiile de update. Acestea sunt executate local la nivelul masterului, pana cand systemul client decide rollback (renuntarea la tranzactie, la linia 8) sau inregistrarea datelor (commit, la linia 13).
    a. In caz de rollback, tranzactia poate fi pur si simplu anulata la nivelul masterului, nicio actiune ulterioar nefiind necesara.
    b. In caz de commit, sistemul master inresitreaza tranzactia in baza de date master *DB,* extrage writeset-ul si il transmite mai departe catre satelitii care detin o copie a bazei sale de date. De asemenea, masterful updateaza valoarea *CN (DB).*
3. Manipularea tranzactiei readonly este descrisa la liniile 39-62. In principiu, astfel de tranzactii sunt intotdeauna etichetate cu *CN (DB)* si rerutate, daca este posibil.
4. Algoritmul 1 si 2 descriu gestionarea tranzactiilor readonly si a writeset-urilor la nivelul satelitilor.
5. Linia 4 din algoritm-ul 2 marcheaza modul prin care ne asiguram ca tranzactiile readonly nu au acces la date invechite. Tranzactiile rerutate catre sateliti sunt intotdeauna startate in modul readonly, astfel ca, daca un client, intr-o tranzactie readonly declarata incearca sa updateze elementele bazei de date, atunci respective baza de date anuleaza automat respectiva tranzactie.

Implementarea

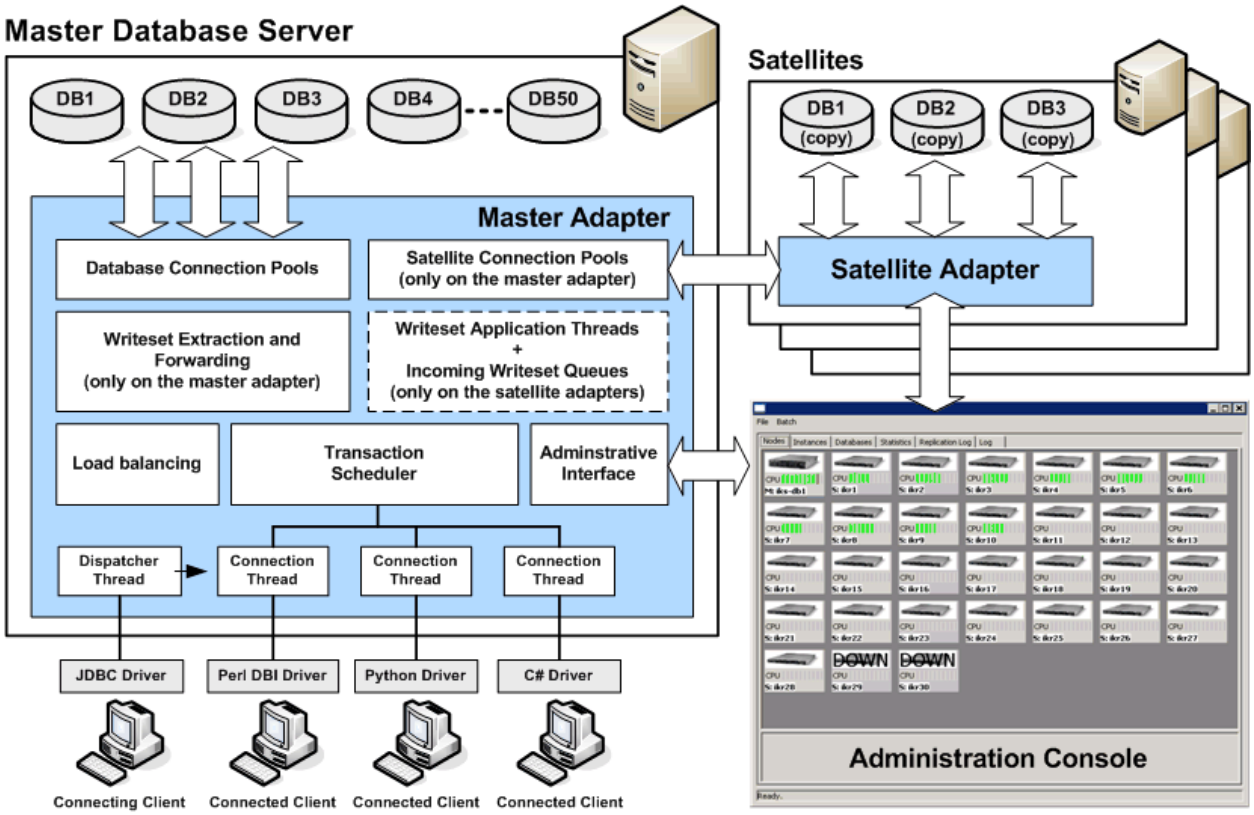
Implementarea curenta a DBFarm ruleaza pe PostgreSQL 8.1. Partea de baza a lui DBFarm o reprezinta adaptoarele, care sunt componentele middleware distribuite folosite pentru a integra conceptele prezentate in sectiunea anterioara. Cum adaptoarele reprezinta componentele principale ale implementarii noastre le vom descrie primele.

Abordarea folosind Adaptoare

Clientii acceseaza DBFarm stabilind o conexiune la un adaptor la un server master. Daca instanta bazei de date cautata nu este gazduita local pe baza de date master, conexiunea este transmisa mai departe catre severul master corect. Copiile bazelor de date gazduite pe masinii satelit nu sunt direct accesibile clientiilor. In *figura 2* adaptorul de pe master intercepteaza toate conexiuniile de la clienti si routeaza tranzactiile read-only catre sateliti. Routarea in sine este mai complicata, deoarece clientii nu trimit niciodata tranzactiile ca un block intreg. Deci adaptorul master trebuie sa inspecteze fluxul de date pentru fiecare conexiune client si sa trateze operatiile corespunzator. Pentru a putea identifica tranzactiile read-only adaptorul master presupune ca clientii folosesc mecanismul standard SQL pentru a declara intreaga sesiune read-only sau sa marcheze inceputul tranzactiilor inregistrate cu atributul read-only. In clientii implementati in Java, de exemplu, acest lucru poate fi realizat prin executarea metodei *Connection.setReadOnly()*. Daca clientul nu ofera aceasta informatie, atunci adaptorul presupune ca tranzactia inceputa este de tipul update.

De asemenea adaptorul principal (master) extrage ultimele modificari produse de fiecare operatie de update (seturi de scriere - writesets) din baza de date principala (master) si le trimite catre adaptoarele satelit corespunzatoare. In contrast cu [6] folosim seturi de scriere in locul query-ului de update original deoarece a fost demonstrat ca reprezinta un mod mai eficient de a propaga modificarile in sisteme replicate [14]. Seturile de scriere sunt formate dintr-o descriere codata si compacta a tuplet-elor ce trebuie inserate, modificate sau sterse. Odata ajunse la adaptorul tinta acestea sunt decodate si executate sub forma unui set minim de query-uri SQL. Extragerea seturlor de scriere pentru o anumita tranzactie are loc dupa ce tranzactia respectiva este comisa si este realizata pentru toata tranzactia. Astfel cand adaptorul principal (master) propaga schimbarile, face acest lucru pentru toate modificarile realizate in timpul unei tranzactii.

Adaptoarele de pe masiniile satelit primesc operatii de la tranzactiile read-only si le executa copiile instalate local asigurandu-se de pastrarea consistentai datelor. Pentru a pastra consistenta datelor, adaptoarele satelit aplica in mod constant seturile de scriere primite si respecta tag-urile de la inceputul tranzactiilor retransmise.



La pornire, fiecare adaptor configureaza si ruleaza instanta locala de PostgreSQL (fiecare instanta contine in mod normal cateva baze de date master sau satelit). Toate fiserele de configurare PostgreSQL necesare sunt generate dinamic astfel incat baza de date PostgreSQL locala se lega la interfata de retea locala. Ca rezultat, toate instantele PostgreSQL nu accesibile clientiilor direct din retea. Dupa ce instanta locala de PostgreSQL este pornita, adaporul se conecteaza la ea si cauta bazele de date instalate. Ca un ultim pas, adaptorul asculta pe interfata de retea externa. Dar, daca nu a fost configurat din consola de administrare, acesta respinge toate cererile primite de la clienti – pana cand nu stie daca ruleaza ca master sau ca satelit. In acest stagiu, informatii legate de intreg DBFarm-ul sunt centralizate si oferite de consola de administrare. Numai dupa ce un master a fost

informat de catre consola de rolul sau si despre alti adaptori master si sateliti diponibili, acesta poate stabili toate seturile de scriere si conexiuniile de retransmitere a tranzactiilor si este gata pentru a procesa tranzactiile primite de la clienti.

In momentul de fata, numai adaptoarele master pot sa retransmita tranzactiile read-only catre sateliti, pentru echilibrarea incarcarii folosindu-se un mecanism de asignare de tipul round-robin. Intr-o dezvoltare ulterioara se doreste explorarea unui mecanism de asignare mai sofisticat pentru a imbunatatii performantele generale. Necesitatea acestui mecanism sofisticat de asignare se naste din faptul ca unele tranzactii read-only pot avea un timp foarte mare de executie, de ordinul orelor. Daca asignarea taskurilor ar lua in calcul si incarcarea, ar distribui mai eficient tranzactiile primite.

Pentru eficienta, conexiuniile de la un master la celelalte adaptoare sunt organizate in pool-uri: pentru fiecare baza de date locala si pentru fiecare adaptor cunoscut este creat un pool. Motivul pentru folosirea unui pool pentru fiecare baza de date locala (in loc sa folosim un pool pentru fiecare instanta de PostgreSQL) este dat de o limitare a protocolului on-wire a PostgreSQL-ului: nu se poate modifica schema si baza de data dupa ce o conexiune a fost stabilita su autentificata. Conexiuniile catre alte adaptoare sunt folosite in 2 scopuri: unu, pentru a timite tranzactii read-only si doi, pentru a trimite seturi de scriere catre sateliti. Cand o conexiune nu este folosita, aceasta este intoarsa in pool. Daca nu sunt folosite pentru o perioada de timp conexiuniile din pool sunt inchise.

Adaptoarele sunt implementate ca un nivel Java. Software-ul epntru adaptoarele master si satelit este acelas, dar in functie de configurare un adaptor se comporta fie ca un master fie ca un satelit. Avantajul de a avea adaptoarele astfel este ca poermite mutarea ulterioara a unui master pe o masina satelit. Acest lucru face ca Dbfarm sa fie mai dinamic dar, de asemenea, schimba proprietatiile arhitecturii rezultate pentru ca satelitii, in principiu, nu sunt fiabili.

Asigurarea Consistentei

Continuand munca facuta anterior despre consistenta bazelor de date replicate [22], folosim izolarea prin instantanee (snapshot isolation - **SI**) ca criteriu de corectitudine pentru bazele de date master si satelit. Produse comune care folosesc SI sunt Oracle, PostgreSQL si Microsoft SQL Server 2005. SI este folosit pentru a preveni conflicte intre operatiile complexe de citire si update. Acesta functioneaza prin atribuirea unui instantaneu al bazei de date (care contine toate modificarile comise pana la acel punct) fiecarei tranzactii. Cum fiecare tranzactie se executa pe un instantaneu diferit, nu au loc conflictele intre citiri si scrieri concurente. In definitia originala a SI-ului, verificarea conflicterlor se realizeaza la momentul commitului: daca tranzactii concurente incearca sa modifice o inregistrare comuna este aplicata regula *primului commit* (prima tranzactie care realizeaza commitul realizeaza modificarea, celelalte sunt anulate). Insa implementariile reale se bazeaza pe metode mai eficiente de detecatre a conflictelor incrementale. Tranzactiile read-only nu sunt verificate pentru conflicte.

Folosirea SI face relativ simpla implementarea necesitatilor de consistenta in DBFarm oferind clientiilor o vedere consistenta a datelor. Cum o tranzactie read-only se executa pe o copie folosind SI, iar copia ofera un instantaneu consistent, o tranzactie read-only va citi un instantaneu care a existat in baza de date. Acest lucru ofera un avantaj important deoarece permite unei copi sa aplice in mod constant update-uri fara a interfera in vre-un fel cu tranzactiile concurentiale read-only venite din partea clientiilor.

In implementarea curenta a DBFarm-ului un adaptor master, vazut de client, arata ca o instanta normala de PostgreSQL (adaptorul asculta pe portul TCP 5432). Am implementat suport atat partea de client cat si partea de server pentru protocolul de nivel jos al PostgreSQL-ului. Partea de server este folosita pentru a implementa frontend-ul PostgreSQL, iar partea de client este folosita pentru a comunica cu instanta locala de baza de date PostgreSQL. Cand routeaza tranzactii intre diferitele adaptoare, acestea folosesc o varianta extinsa a protocolului client.server PostgreSQL.

Cum adaptoarele master implementeaza interfata standard a serverului PostgreSQL, DBFarm poate fi folosit de o multitudine de aplicatii si platforme: C, C++, Java, Pearl, Python, .NET, etc – de fapt orice aplicatie care este conceputa pentru a folosi PostgreSQL. Mai mult poate fi folsit cu aplicatii deja existente fara ca acestea sa necesite modificari cat timp aplicatiile folosesc interfata standard PostgeSQL.

Extragerea seturilor de scriere

Am implementat si testat diferite variantede extragere a seturilor de scriere. In momentul de fata DBFarm suporta doua abordari. Abordarea generica, de baza, este similara cu ce este folosit in alte sisteme: colectam toate cererile DML ale tranzactiilor din adaptorul master si le folosim ca seturi de scriere. Aceasta metoda a fost folosita pentru testare, deoarece are multe probleme nerezolvate inca. De exemplu, nu se poate folosi pentru update-uri facute de triggere, deoarece acestea nu sunt vizibile adaptorului. De asemenea are probleme cu cereri care instruiesc baza de date sa insereze valori generate de functii.

A doua abordare este bazata pe triggere: am implementat o biblioteca comuna care poate fi incarcata in PostgreSQ la pornire. Libraria contine functii care vor fi atribuite de adaptorul master ca triggere pe toate tabelele care necesita replicare. Cand se produce o modificare asupra unei tabele, indiferent daca este facuta de user sau de o procedura, triggerul captureaza modificarile. Setul de scriere este colectat in memorie. La finalul tranzactiei adaptorul master poate apela o alta functie pentru a extrage setul de scriere. Acest lucru se realizeaza foarte rapid ne fiind necesar cititrea de pe disk. Implementarea noastra este capabila sa prinda si modificari ale schemei cauzate de comenzi DDL si sa produca seturi de scriere care sa reproduca modificarile.

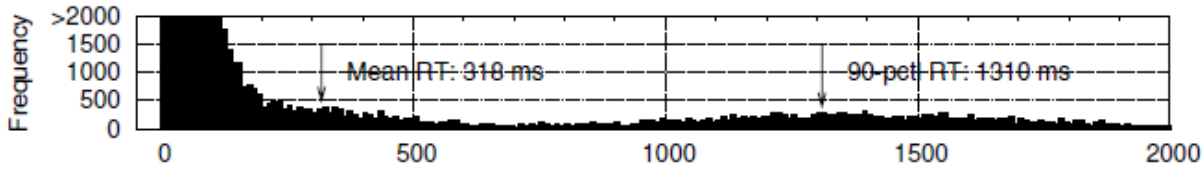
Consola de administrare

Consola de administrare a fost implementata ca o aplicatie grafica java independenta de platforma. Este folosita pentru a porni, opri si configura de la distanta adaptoarele. Mai mult ajuta la inspectarea fiecarui nod cluster. Toate comunicariile intre consola de administrare si nodurile cluster DBFarm sunt codate. Tot ce este nevoie este ca fiecare nod sa ruleze un daemon OpenSSH. Pentru a putea folosi protocolul SSH-2 din Java, folosim propria librarie open source SSH-2.

Evaluarea performantelor

In general abordarea noastra nu restrictioneaza modul in care resursele pot fi impartite intre bazele de date. Dar, in aceast document, evaluam performantele statice ale setup-ului unde un singur server puternic gazduieste toate bazele de date master si un set de sateliti mai putin puternici care gazduiesc copiile read-only. Prezentam rezultatele unei instlari de DBFarm care contine 360 de baze de date gazduite pe serverul master si pana la 30 de sateliti care sa ofere performante imbunatatite pentru clienti.

Pentru a produce masuratori realistice, am folosite setup-uri de baze de date bazate pe pe 2 standarde de testare diferite: TCP-W si RUBBoS. TCP-W modeleaza cumparatorii care acceseaza o librarie online, in timp ce RUBBoS modeleaza un site de stiri (bulletin board). Pentru TCP-W s-a folosit incarcarea default care este compusa in proportie de 80% de interactiunii read-only. In cazul RUBBoS incarcarea cu interactiuni read-only este de 85%.



Am instalat 300 baze de date TCP-W (folosind un factor de scalare de 100/10000, care rezulta in 497MB pe baza de date) pe serverul master, precum si 60 de baze de date RUBBoS(folosind datasetul extins rezultand intr-un consum de 2440MB). Dimensiuniile specificate include tot spatiul pe disk folosit de bazele de date. Totalul spatiului ocupat de bazele de date pe disk depaseste 417GB. Toate bazele de date include indecsi.

Serverul master are 2 procesoare Intel(R) Xenon 3.0 Ghz cu 4GB RAM cu 5 HDD Hitachi HDS722525VLAT80 de 250GB care rezulta intr-un spatiu de stocare de 931GB, ruland un sistem de operare Fedora Core 4. Masiniile satelit au 2 procesoare AMD Opteron™ 250 la 2.4Ghz, 4GB RAM si un HDD Hitachi de 120GB. Aceste masini ruleaza Red Hat Enterprise Linux AS versiunea 4. Toate masiniile sunt conectate printr-o retea de 100Mbit. Adaptoarele au fost rulate pe o masina virtuala de java Java-Blackdown 1.4.2-02. Am folosit o versiune nemodificata de PostgreSQL 8.1.

Pentru a masura performantele pentru setup-ul nostru am folosit un client de testare al incarcarii dezvoltat in Java care este capabil sa reproduca incarcariile generate de TCP-W si RUBBoS. Trebuie sa amintim ca nu facem toate testele ci numai cele ce tin de baza de date.

Clientul foloseste threaduri pentru a simula un numar mare de clienti. La pornire clientul genereaza un pool de conexiuni catre baza de date master. Daca numarul de thread-uri este mai mic decat numarul maxim de conexiunii atunci se mai creeaza o conexiune. Altfel xlientul genereaza conexiunii folosind metoda round-robin. De asemenea pentru fiecare conexiune exista o masina de stare care dicateaza urmatoarea tranzactie care va fi executata. Fiecare thread ruleaza intr-o bucla infinta si executa tranzacti conform masinii de stare.

Cand testeaza sistemul, clientul de test variaza numarul de threaduri. Cand se schimba acesta clientul foloseste un timp de “incalzire” de cateva minute pentru a stabiliza sistemul, apoi urmeaza perioada de teste timp de 2 min. In timpul perioadei de test clientul masoara timpul de raspuns pentru toate tranzactiile efectuate.

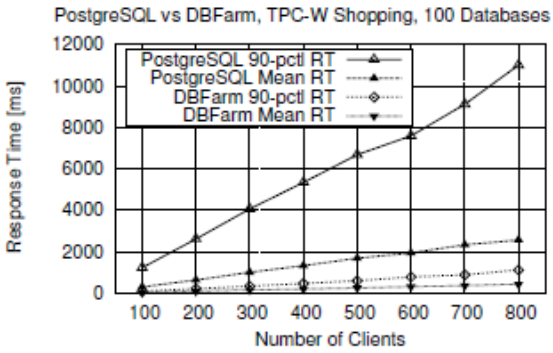


Fig. 4. TPC-W Results for 100 Databases.

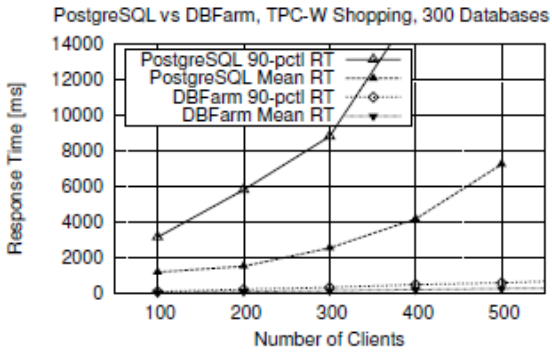


Fig. 5. TPC-W Results for 300 Databases.

Partea A: Accesul Concurest la bazele de date

In experimentul ce urmeaza vom arata ca DBFarm este capabil sa se descurce in cazul in care mai multe baze de date sunt accesate concurent. Folosim un set de satelti pentru a executa tranzactiile read-only, reducant numarul de slecturi facute de master. In acelas timp mai multe resusrse sunt disponibile pentru executarea tranzactiilor de update. Pentru acest experiment am folosit un setup simplu, cate un satelit pentru fiecare baza de date master.

Rezultate pentru TCP-W: In acest experiment am comparat performantele obtinute pentru un numar mare de baze de date TCP-W accesate concurent. Mai intai am folosit clientul pentru a stresa baza de date principal. Apoi am pornit sistemul DBFarm si am masurat performantele din nou.

In primul experiment am folosit 100 de baze de date TCP-W concurente. Se poate observa ca serverul principal a ajuns la capacitate maxima cu 300 de conexiunii TCP-W concurente moment in care timpul de raspuns al serverului ajunge la 4 secunde, lucur neacceptat in cazul unei aplicatii interactive.

Prin aplicarea aceleias incarcari pe DBFarm cu 10 sateliti atasati (fiecare ruland 10 copii ale bazei de date), se poate observa ca ssitemul se poate scala la un numar mult mai mare de conexiunii concurente si in acelas timp sa ofere timpi de raspuns acceptabili. Aceste rezultate sunt cu atat mai importante deoarece ruleaza o singura instant a bazei de date master. Performantele pot fi imbunatatite adaugand mai multi sateliti si avant 2 copi a fiecarei baze de date master.

Cu setupul DBFarm, fiecare satelit gazduieste 10 copii ale bazei de date TCP-W. Acest lucru inseamna ca fiecare satelit are un data sed de aproximativ 5GB. Avand in vedere ca cererile asupta bazei de date TCP-W sunt axate in principal pe o parte limitata a bazei de date (query-uri pentru cele mai vandute carti din librarie), cei 4GB de memorie sunt suficienti pentru a limita la un minim citirile de pe disk. De asemenea operatiile de update care apar in baza de date TCP-W (in general operate asupra caruciorului clientului) necesita doar o arte foarte mica din baza de date. Astfel, baza de date master din DBFarm (executa numai tranzactii de update) poate face fata cu usurinta tranzactiilor de update pentru toate bazele de date accesate. Majoritatea accesului la disk se face pentru a scrie modificarile, acest lucru fiind valabil si pentru sateliti, care dupa stagul initial, in mare parte acceseaza disk-ul pentru a scrie modificarile.

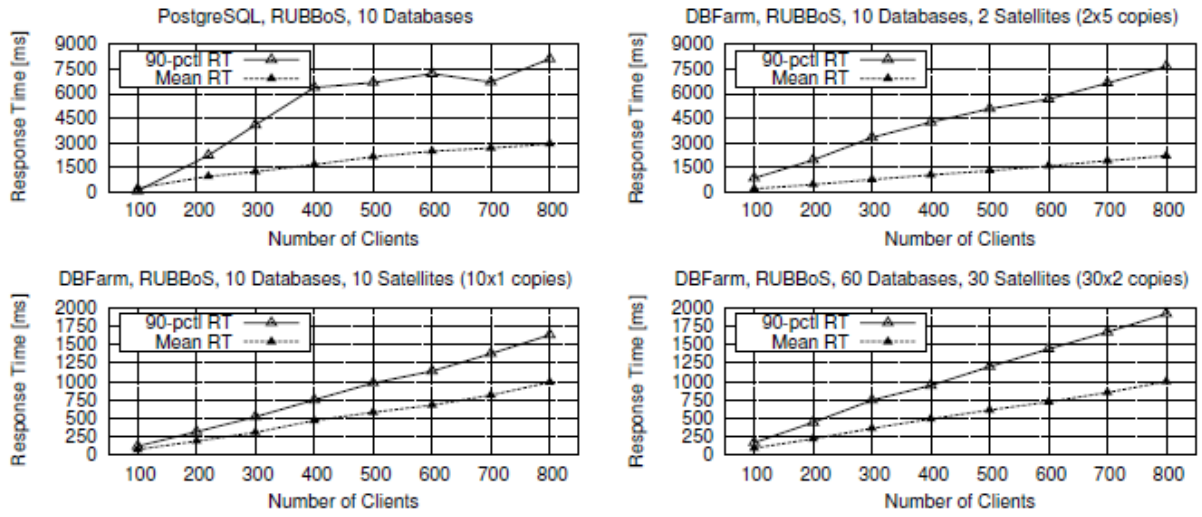


Fig. 6. RUBBoS Results.

Incurajat de rezultatele bune pentru 100 de baze de date concurente am incercat folosirea a 300 de baze de date TCP-W. Rezultatele sunt afisate in figura 5. Se poate observa foarte clar ca in cazul nefolosiri DBFarm-ului la 300 de conexiunii concurente asupra baei de date TCP-W timpi de asteptare nu sunt acceptabili. Pentru ca masina realizeza in mare parte operation de scriere si citire pe disk, rezultatele sunt irelevante, performantele find dictate de controlerul RAID-5.

Acelas experiment executate pe DBFarm cu 300 de baze de date folosind 30 de sateliti (fiecare avand cate 10 baze de date) arata ca DBFarm este capabil sa ofere timpi de raspuns acceptabili pentru acest scenario.

Rezultatel pentru RUBBoS: Bazele de date RUBBoS nu sunt numai mai mari (peste 2GB) decat bazele de date TCP-W, volumul de munca este mai complex, iar tranzactiile rezultate nu folosesc numai o parte din baza de date ci aproape pe toata.

Ca si pana acum mai intai am verificat performantele bazei de date master singura. Rezultatele se pot observa in figura 6. Se poate observa ca baza de date master nu poate suporta multe baze de date RUBBoS concurente, 10 baze de date concurente ducand deja la problem de performanta.

Uitandu-ne la rezultatele pentru DBFarm, se poate observa ca este crucial numarul de copii de pe fiecare satelit sa nu depaseasca un anumit numar. In experimental in care am folosit numai 2 sateliti (fiecare continand 5 instante ale bazei de date RUBBoS) imbunatatirilie asupra performantelor au fost insignifiante. Acest lucru se datoreaza faptului ca DBFarm are aceeas problema ca si un singur server master: setul de lucru pentru 5 baze de date este pre amare pentru memoria disponibila, asa ca distributia muncii pe sateliti este limitata de banda de transfer a disk-uliui. Se poate considera ca am mutat constrangerea de pe serverul master pe sateliti. Acest lucru poate fi privit ca irosirea resurselor, dar trebuie luat in considerare ca setup-ul general a fost imbunatatit: preluand din munca bazei de date master, exista o capacitate crescuta pentru alte conexiunii concurente sa acceseze baza de date. Vom evidentia acest lucru in sectiune urmatoare. Pentru a verifica faptul ca sateliti sunt acum constrangerea, am testat acelas volum de munca dar folosind 10 sateliti, fiecare gazduind o

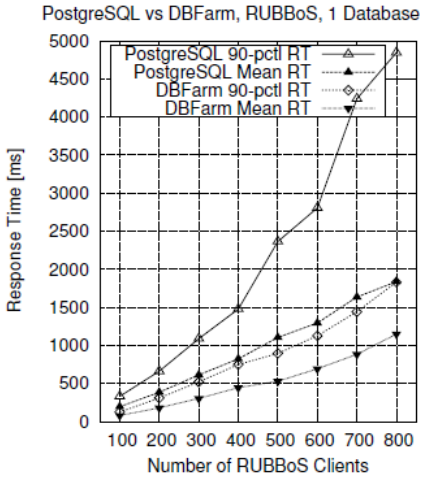
singura instant a abazei de date RUBBoS. Se poate observa o imbunatatire seminificanta fata de setup-ul cu 2 sateliti.

In ultimul experiment am incercat sa gazduim 60 de baze de date RUBBoS concurente. Pentru acest setup am folsoit 30 de sateliti fiecare gazduind 2 instante ale bazei de date. A fost imposibil sa executa acest experiment cu un singur server master, deoarece ar fi fost blocata cu operatii de citire si scriere pe disk. Rezultatele din figura 6 arata ca DBFarm poate sa suporte si acest scenario. Interesant este faptul ca performantele sunt ceva mai mici fata de cele obtinute pentru 10 sateliti. Sunt 2 motive pentru aceste rezultate: unu, avant 2 instante ale bazei de date RUBBoS, bufferul cache al satelitiilor nu este sufficient de mare pentru a tine ambele baze de date in memorie. Dar acest lucru nu este un factor foarte important daca se observa numarul de operatii de citire si scriere pe disk din timpul experimentului. Cele dea-l doilea este faptul ca, avand de gestionat 60 de baza de date concurente , serverul master devine o constrangere, avand in vedere ca daele pentru fiecare tanzactie de update nu se afla in memorie tot timpul (masina executa mai multe operatii de citire/scriere pe disk decat satelitii). Din acest experiment putem invata urmatorul lucru: cand se foloseste un system ca DBFarm, este foarte important sa se optimizeze strucutra tanzactiilor de update, trebuie incercat sa se minimizeze operatiile de citire, altfel baza de date master devine o constrangere a sistemului. Acest lucru se poate realiza prin introducerea indecsi specifici pentru baza de date master.

Partea B: Scalarea bazelor de date selectate

In experimentele anterioare am folosit sateliti cu copii ale bazelor da date pentru a extinde capacitatea severului master.

In ultimul experiment aratam cum o singura baza de date RUBBoS poate beneficia de folosirea sistemului DBFarm. De exemplu, ne putem imagina existent unui client cu proritate mare care necesita un timp de acces maxim garantat printr-un contract. Pentru a rezolva aceasta problema putem atribui un numar de sateliti, fiecare gazduind o copie a bazei de date client de pe serverul master. In acest fel tranzactiile read-only ale clientului pot fi impartie intre sateliti care in acelas timp nu vor fi accesati de alti client. Din nou am facut 2 masuratori: mai intai am masurat performantele unui server PostgreSQL instalat pe masina master, apoi am masurat perfomantele cu setup-ul DBFarm. Dar, in acest caz, incarcarea asupra serverului a fost mai mare, in parallel pe langa RUBBoS 200 de baze de date TCP-w au fost accesata de 100 de conexiuni. Copii ale celor 200 de baze de date TCP-w au fost impartite pe 20 de sateliti. In cazul RUBBoS am folosit 3 sateliti fiecare gazduind o copie a bazei de date master.



RUBBoS Clients	DBFarm RT Mean	DBFarm RT 90-pctl	DBFarm TPM	PostgreSQL RT Mean	PostgreSQL RT 90-pctl	PostgreSQL TPM
100	82	127	72,526	197	333	30,854
200	183	311	65,279	387	663	30,895
300	306	523	58,726	614	1,090	29,176
400	448	751	54,981	825	1,478	27,426
500	531	897	56,456	1,106	2,365	25,673
600	695	1,128	51,961	1,296	2,807	23,921
700	884	1,445	47,480	1,637	4,239	22,736
800	1,152	1,832	41,770	1,839	4,845	20,485

Fig. 7. Detailed Scale-Out Results for the RUBBoS Database. Note that DBFarm had to handle simultaneously 100 clients that randomly accessed 200 TPC-W databases (not included in TPM).

Rezultate celor 2 experimente sunt evidentiate in figura 7. Se poate observa clar ca baza de date prioritara RUBBoS are performante mai bune decat in cazul unui singur server, desi DBFarm trebuie sa se ocupe si de cele 200 de conexiuni concurente pentru cele 200 de baze de date TCP-W. In table se poate observa ca trnzactiile pe minut pentru RUBBoS s-au dublat.

Lucrari conexe

DBFarm este construita pe ideii dezvoltate anterior in cateva proiecte [15,16,22] precum si pe o multime de lucrari middleware bazate pe teplicarea bazelor de date. In [22] prezentam un system pentru replicarea bazelor de date folosind iszolarea prin instantanee (snapshot isolation). Versiunea curenta de DBFarm foloseste aceasta implementare pentru a oferi consistenta clientiilor.

Pe partea teoretica, [8] a studiat in detaliu problema consistentei sesiuniilor ca un criteriu mai correct pentru replicarea bazelor de date. Algoritmi incearca sa ofere serialibilizarea pentru o singura baza de date, complet replicate si pana acum au simulati cu success. DBFarm ofera o notiune puternica de consistenta folosind un mechanism care nu ar trebui sa cauzeze o scadere a performantelor ca in cazul [8] si [9]. Din nou sistemul nostru ofera scalabilitate fara a face rabat de la oferirea de rezultate consistente pentru toti clientii. In implementarea noastra folosim de asemenea izolarea prin instantanee ca un mecanism de control concurential.

In termen de sisteme implementate, [1] aplica tehici prezentate in [2] pentru a oferi posibilitatea de a *calatori in timp* pentru clientii care doresc sa ceara instantanee mai vechi. Desi si aceasta tehnica poate fi implementata in DBFarm, scopul este sa support consistenta totala si “On Line Transaction Processing”. De notat este ca odata ce verificarea de consistenta e mai “relaxata”, scalabilitatea poate creste exponential. Munca descrisa in [2] se bazeaza pe o tehnica numita verzionare distribuita. Ideea principal este folosirea unui middleware contralizat care sa se ocupe de salvare versionarii pentru fiecare tabela din fiecare replica. Fiecare tanzactie de update executata pe o tabela creste numarul versiunii. La inceputul fiecarei tranzactii , client trebuie sa informeze

middleware-ul de tabelele pe care le vor modifica, iar acesta foloseste informatia pentru a asigna numere de versiune pentru tranzactii.

Exista un numar de sisteme care folosesc comunicarea de grup pentru a implementa replicarea [15,16,17]. Aceste sisteme nu iau in calcul impartirea volumului de munca si impugn resctricitii severe asupra incarcarii trnazactiilor. De exemplu este necesar ca tranzactiile sa fie executate ca un bloc deoarece sistemul poate executa decat tranzactii complete. Aceste lucru este in contrat cu DBFarm unde clientii pot trimite tanzactiile cerere cu cerere cum se intampla in majoritatea sistemelor de baze de date. Din punctual de vedere al bazelor de date cluster cu mai multe instante, cel mai mare dezavantaj al replicarii la comunicare in grup este volumul de date suplimentar necesar pentru comunicare in grup.

Oracle RAC (Real Application Cluster) este o aplicatie comerciala care ofea de asemenea izolarea prin instantanee. Se bazeaza pe folsirea unui hardware special (toate nodurle au acces la la un set de disk-uri comune) sau folosesc sisteme de stocare in retea. Deci, spre deosebire de abordarea noastra, sistemul nu paote fi instalat pe un set de servere normale.

Concluzii

Aceasta lucrare prezinta arhitectura si implementarea DBfarm (ferma de baza de date), o solutie de cluster de baze de date cu instante multiple care poate suporta sute de conexiuni concurente din partea clientiilor. Mai mult, suporta scalabilitate pentru anumite baza de date ale clientiilor. DBFaram ofere consistenta datelor tot timpul, si pentru client arata ca o baza de date obisnuita. Nu este nevoie sa se schimbe clientul folosit pentru a folosi acest system. Abordarea noastra folosind un adaptor ofera multe avantaje asupra folosiri unui middleware. Experimentele arata ca aceasta abordare este fesabila sic a sistemul este capabil sa gestioneze tranzactii pentu un volum mare de date in acelas timp oferind performante pentru un numar mare de conexiuni client concurente.

Dezvoltariile ulterioare se voar concentra pe aspectul dinamic al sistemului. Alocand sateliti si stabilind conexiunii la baze de data copii in functie de cerere.

Bibliografie

1. F. Akal, C. T¨urker, H.-J. Schek, Y. Breitbart, T. Grabs, and L. Veen. Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees. In Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005, pages 565–576.
2. C. Amza, A. L. Cox, andW. Zwaenepoel. A Comparative Evaluation of Transparent Scaling Techniques for Dynamic Content Servers. In ICDE ’05: Proceedings of the 21st International Conference on Data Engineering (ICDE’05), pages 230–241.
3. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil. A Critique of ANSI SQL Isolation Levels. In Proceedings of the SIGMOD International Conference on Management of Data, pages 1–10, May 1995.
4. P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.
5. E. Cecchet. C-JDBC: a Middleware Framework for Database Clustering. IEEE Data Engineering Bulletin, Vol. 27, No. 2, June 2004.

6. E. Cecchet, J. Marguerite, andW. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware.
In USENIX Annual Technical Conference, FREENIX Track, pages 9–18, 2004.
7. C. Coulon, E. Pacitti, and P. Valduriez. Consistency Management for Partial Replication in a High Performance Database Cluster. In Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS 2005), Fuduoka, Japan, July 20-22, 2005.
8. K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In Proceedings of the 20th International Conference on Data Engineering (ICDE 2004), 30 March - 2 April 2004, Boston, MA, USA, pages 424–435.
9. S. Elnikety, F. Pedone, and W. Zwaenepoel. Database Replication Using Generalized Snapshot Isolation. In SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems.
10. A. Fekete. Allocating Isolation Levels to Transactions. In PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 206–215, June 2005.
11. A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making Snapshot Isolation Serializable. ACM Trans. Database Syst., 30(2):492–528, 2005.
12. R. Jim´enez-Peris, M. Pati˜no-Mart´ınez, and G. Alonso. An Algorithm for Non-Intrusive, Parallel Recovery of Replicated Data and its Correctness. In 21st IEEE Int. Conf. on Reliable Distributed Systems (SRDS 2002), Oct. 2002, Osaka, Japan, pages 150–159.
13. . R. Jim´enez-Peris, M. Pati˜no-Mart´ınez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. In IEEE 22nd Int. Conf. on Distributed Computing Systems, ICDCS'02, Vienna, Austria, pages 477–484, July 2002.
14. B. Kemme. Database Replication for Clusters of Workstations. PhD thesis, Diss. ETH No. 13864, Dept. of Computer Science, Swiss Federal Institute of Technology Zurich, 2000.
15. B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication. In Proceedings of the 26th International Conference on Very Large Databases, 2000.
16. B. Kemme and G. Alonso. A New Approach to Developing and Implementing Eager Database Replication Protocols. ACM Transactions on Database Systems, 25(3):333–379, 2000.
17. Y. Lin, B. Kemme, M. Pati˜no-Mart´ınez, and R. Jim´enez-Peris. Middleware based Data Replication
providing Snapshot Isolation. In SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, pages 419–430.
18. J. M. Milan-Franco, R. Jim´enez-Peris, M. Pati˜no-Mart´ınez, and B. Kemme. Adaptive Distributed Middleware for Data Replication. In Middleware 2004, ACM/IFIP/USENIX 5th International Middleware Conference, Toronto, Canada, October 18-22, Proceedings, 2004.
19. OpenSSH. A free version of the SSH protocol suite. http://www.openssh.org/.
20. T. Ozsu and P. Valduriez. Principles of Distributed Database Systems. Prentice Hall, 1999.
21. C. Plattner. The Ganymed SSH-2 Library. http://www.ganymed.ethz.ch/ssh2.
22. C. Plattner and G. Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In Middleware 2004, 5th ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada, October 18-22, Proceedings, 2004.
23. R. Schenkel and G.Weikum. Integrating Snapshot Isolation into Transactional Federation. In Cooperative Information Systems, 7th International Conference, CoopIS 2000, Eilat, Israel, September 6-8, 2000, Proceedings.
24. The ObjectWeb Consortium. RUBBoS: Bulletin Board Benchmark. http://jmob.objectweb.org/rubbos.html.
25. . The Slashdot Homepage. http://slashdot.org/.
26. The Transaction Processing Performance Council. TPC-W, a Transactional Web E-Commerce Benchmark. TPC-C, an On-line Transaction Processing Benchmark. http://www.tpc.org.

27. S. Wu and B. Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control Based on Snapshot Isolation. In ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05), pages 422–433.

(Text realizat de Iacob Mihai Vlad, grupa ABD master, ianuarie 2011, pentru cursul "Sisteme avansate de baze de date". În construirea textului, pe lângă bibliografia indicată, autorul a folosit mai multe surse, dintre care menționez **DBFarm: A Scalable Cluster for Multiple Databases**,de Christian Plattner, Gustavo Alonso, și M. Tamer ¨Ozsu, Department of Computer Science. ETH Zurich, 2006. Această ultimă sursă face parte din lista de articole propusă în New Folder 1, la cursul de "Sisteme avansate de baze de date").

I. În sfârșit, am rugămintea ca în continuare, să fie introduse enunțurile unor exerciții, pentru "Baze de date 2". Acum, voi da un singur enunț, urmând ca altele să le propunem mai târziu:

Problema P1, "Baze de date 2", 1 Octombrie 2012:

Fie o întreprindere INTR cu entitățile A, B, C. Asoc iați cu A, B, C atribute, așa cum doriți, dar unul din atribute trebuie să fie cheie. Între A și B există asocierile (a b) de tip m la n și (b a) de tip n la 1. Entitatea B are o asociere proprie (b b), părinte – copil, de tip n la 1. Între B și C există asocierea (b c) de tip 1 la 1. Entitatea A posedă o asociere "depinde de" cu o entitate slabă, căreia îi puteți adăuga propriile atribute.
Se cere: a/ să se reprezinte INTR printr-o diagramă entitate – asociere convențională; b/să se transfere imaginea obținută, într-un model cu perechi (min, max).

Notă. Vă rog ca în soluționarea problemei să vă simțiți liberi în a face ipotezele pe care le veți considera potrivite. Vă recomand să folosiți textul propus pentru textul " modelul entitate asociere extins". Soluțiile trebuie transmise la adresa bazededate2@gmail.com.