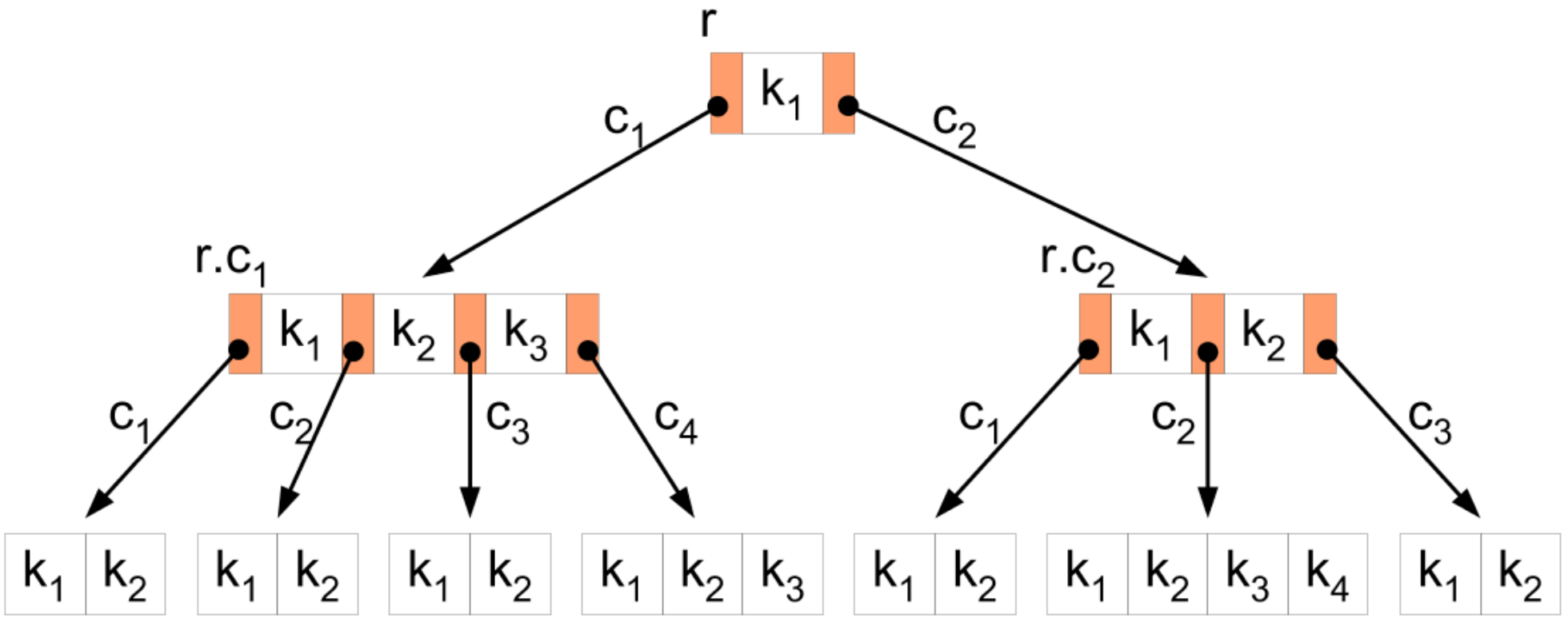
VLDB 2008

A practical and scalable distributed B-tree

Marcos K.Aguilera
HP Laboratories

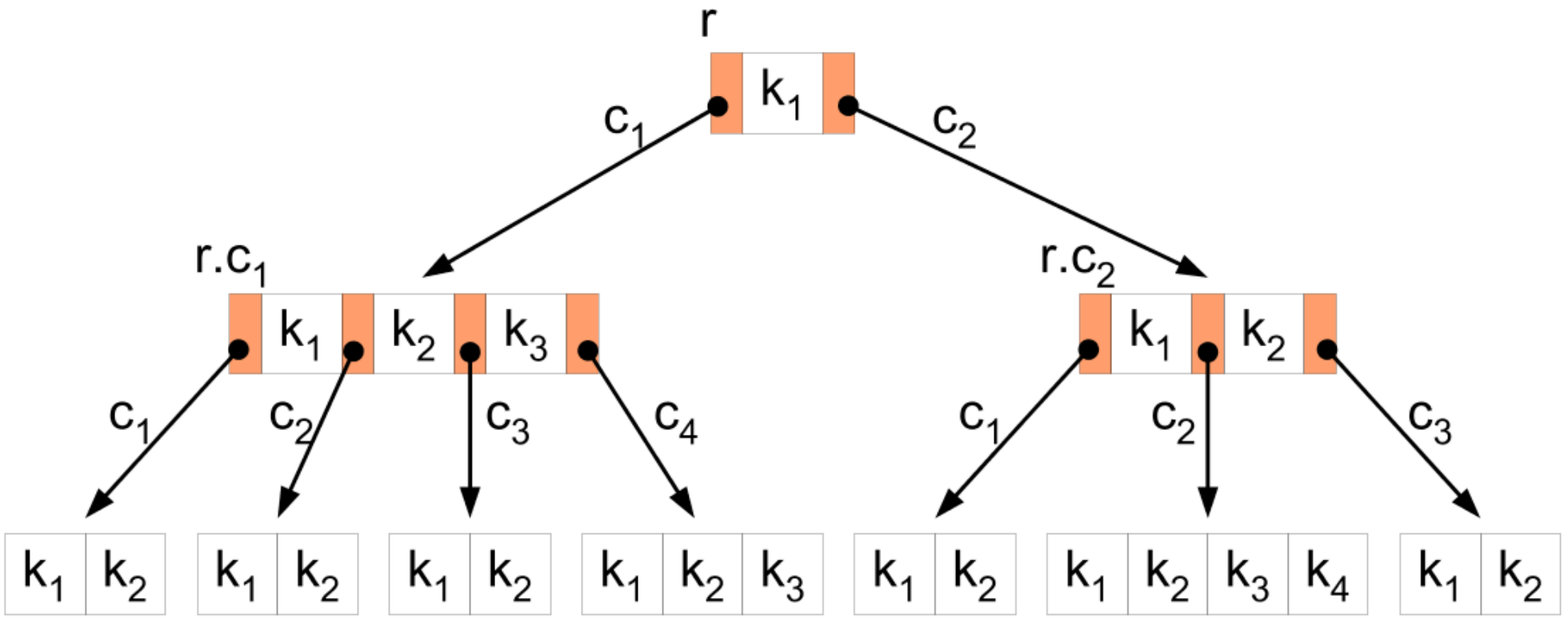
Wojciech Golab
University of Toronto

B+drzewo



- Przechowuje dane tylko w liściach
- Nie potrzebuje częstego wyważania
- Idealne do przechowywania dużych ilości rekordów

Węzeł B+drzewa



- Każdy węzeł (prócz korzenia) ma od rząd do 2*rząd+1 dzieci
- Węzły posiadają wartości, dzięki którym rozdzielają wartości przechowywane w dzieciach

Cel autorów pracy

- Napisać algorytm działania rozproszonych B-drzew, których węzły są rozproszone po różnych maszynach
- Algorytm powinien być odporny na błędy, skalowalny i wydajny (przeważnie do 1-2 „przekierowań” w sieci)

- Powinien też oferować:
- Transakcyjny dostęp do węzłów (kilku jednocześnie)
- Migrację węzłów drzewa (między serwerami, żeby zrównoważyć ich obciążenie)
- Dynamiczne tworzenie i usuwanie serwerów

Do tej pory

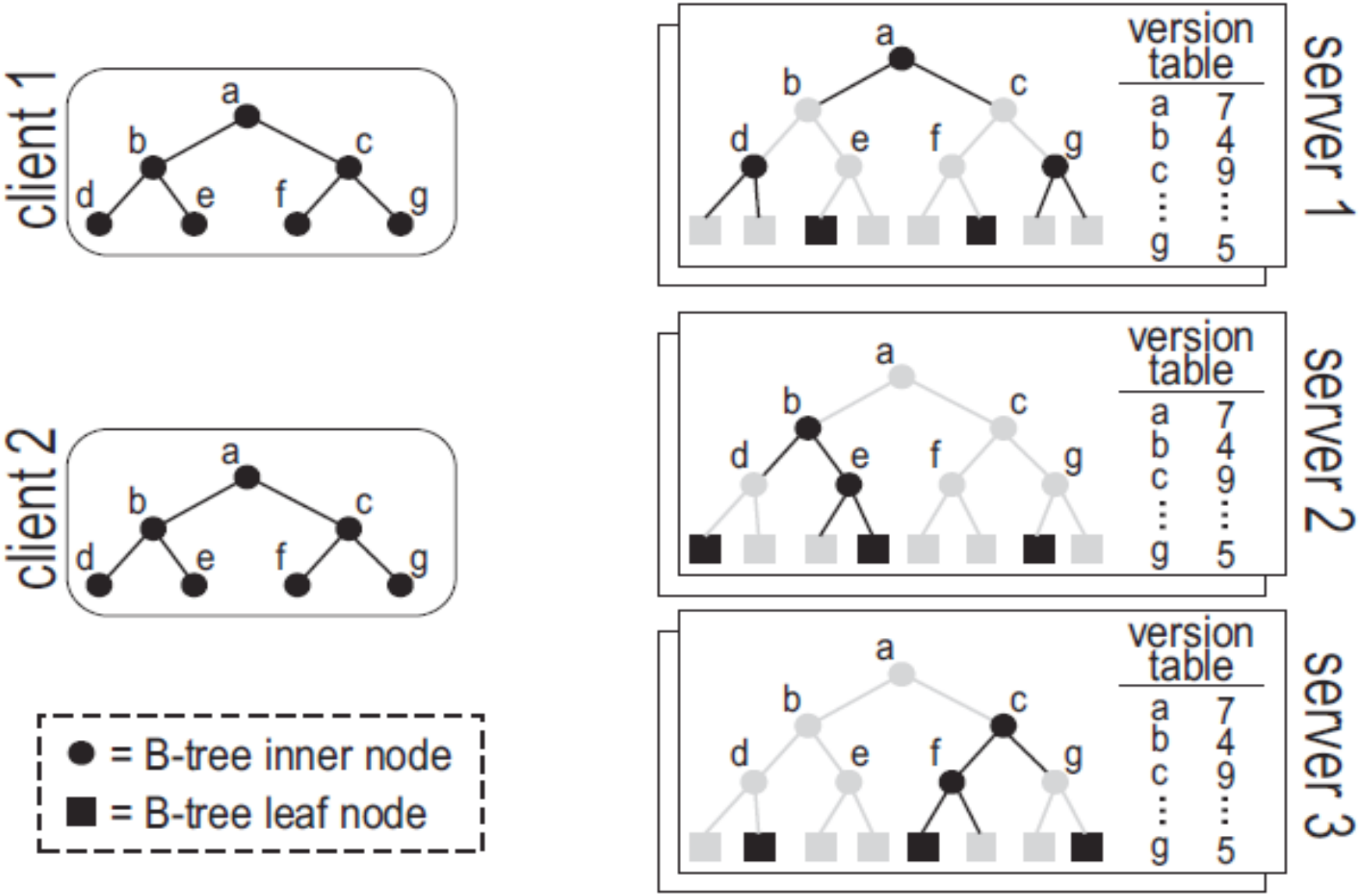
- B-link trees – z dowiązaniem do prawego rodzeństwa
- Distributed hash tables – zazwyczaj nie uwzględniają porządku rekordów. Dobrze radzą sobie z dynamiczną strukturą sieci (P2P), gorzej z zapewnieniem poprawności danych
- Powstały DHT przechowujące poprawne dane, ale bez obsługi złożonych transakcji

- Przesłanką do użycia blokad i pamięci dzielonej jest architektura wieloprocesorowa. Autorzy pracy zakładają raczej system *message-passing*, gdzie komunikacja kosztuje sporo

Model

- Procesy: *serwery* i *klienci*
- Serwery są połączone z sobą i z klientami
- Połączenia są niezawodne
- Serwery po awarii są w stanie się odtworzyć z kopii zapasowej. Klienci nie muszą
- Komunikacja jest synchroniczna (tak działają centra danych; ułatwione wykrywanie błędów)

Mniej więcej



B-drzewo - operacje

- Insert(k,v) – dodaje (k,v) do drzewa
- Lookup(k) – zwraca v, gdy (k,v) należy do drzewa, wpp błąd
- Delete(k) – usuwa (k,v)
- getNext(k) – zwraca najmniejsze k' > k, że (k, _) w drzewie
- getPrev(k) – zwraca największe k' < k, że (k, _) w drzewie

- Przy dodawaniu nowych węzłów może nastąpić dzielenie istniejących węzłów, a przy usuwaniu – scalanie
- Te operacje powinny być „*linearizable*”, czyli wykonywać się bez opóźnień

Dodatkowe operacje

- addServer(s)
- removeServer(s)
- Migrate(x, s) – przenosi węzeł x na serwer s
- firstNode(s) – *enumeruje* węzły na serwerze s w pewnym porządku
- nextNode(x, s)

- B-drzewo działa nieprzerwanie podczas wykonywania tych operacji

Techniki: odwlekanie blokady

- Węzły drzewa nie są blokowane podczas wykonania transakcji
- Blokada na węzłach zakładana tylko podczas *commit*'a
- Wtedy klient sprawdza, czy węzły odczytywane podczas transakcji nie zmieniły się
- Aby ułatwić sprawdzanie, dla każdego węzła trzymamy jego numer wersji
- Jeśli numer jakiegoś węzła się zmienił, cała transakcja jest ponawiana

- To działa, bo zwykle jest mało zmian na węzłach B-drzewa

Techniki: kopia u klienta

- Klient trzyma kopie węzłów, z których skorzystał do tej pory
- Z racji dużej liczby dzieci węzłów (zwykle ok. 200) – ilość przechowywanych danych nie jest znacząca
- Kopie są trzymane leniwie – aby ograniczyć ruch w sieci
- Nowa wersja węzła jest pobierana, gdy klient chce go odczytać i nie jest on aktualny

- Uniknięcie konieczności odświeżenia dużej ilości klientów w krótkim czasie, gdy zachodzi jakaś zmiana

Techniki: serwery znają numery wersji

- Transakcje często będą sprawdzać wersje poszczególnych węzłów
- Wszystkie serwery znają wersje wszystkich węzłów
- Wersje są odświeżane na bieżąco
- Dzięki temu wystarczy się skontaktować z 1 serwerem, aby sprawdzić aktualność węzłów

- Generowany ruch w sieci jest znośny, ponieważ liczba węzłów we wnętrzu B-drzewa nie jest imponująca

Wszystkie 3 techniki są potrzebne

- Bez optymistycznego przeprowadzania transakcji – potrzeba dużego ruchu w sieci ze strony klientów, aby blokować węzły na serwerach
- Bez leniwych kopii drzewa u klientów – potrzeba by było za każdym razem wielu *pogadanek* z serwerami aby przejść drzewo
- Bez zachłannego liczenia wersji węzłów na serwerach – serwery trzymające korzeń drzewa bądź jego dzieci stałyby się szybko wąskim gardłem systemu

- W algorytmie unika się skomplikowanych (i przez to podatnych na błędy) protokołów

Transakcje

- BeginTx() - czyści zbiory *read* i *write*
- Read(n) – czyta obiekt n i dodaje (n, val) do zbioru *read*
- Write(n, val) – dodaje (n, val) do zbioru *write*
- Commit()
- Abort()
- IsAborted()
- EndTx() - czyści struktury użyte w transakcji

- (de)alokacja obiektów jest transakcyjna
- Transakcje mają zbiory *read* i *write*, które przechowują obiekty

Transakcje

```
Function BeginTx
  status ← pending

Function EndTx
  if status = pending then Abort ()
  readSet ← ∅
  writeSet ← ∅

Function Abort
  if status = pending then
  |   status ← aborted
  |   foreach (i, obj) ∈ readSet do
  |   |   localReplicas[i] ← ⊥
  |   end
  end
  return OK

Function IsAborted
  Output: true if status is aborted, otherwise
          false
  return status = aborted
```

Transakcje

```
Function Commit
  Output: true if readSet successfully validated,
          otherwise false
  if status ≠ pending then
  |  return false
  end
  Ret ← ValidateUpdate(readSet, writeSet)
  if Ret = ⊥ then
  |  Abort ()
  |  return false
  else
  |  status ← committed
  |  foreach (i, obj) ∈ writeSet do
  |  |  if obj is an inner tree node then
  |  |  |  localReplicas[i] ← obj
  |  |  end
  |  end
  |  return true
  end
```

Transakcje

- Commit standardowo składa się z dwóch faz:
- Blokujemy zmieniane obiekty – rozsyłanie zbiorów *read* i *write* do serwerów ...
- ... serwery zwracają OK, gdy obiekty zablokowane i można przeprowadzić transakcję
- Klient każe serwerom zatwierdzić transakcję

- Gdy zachodzi konflikt, wznowienie po losowym czasie
- Gdy awaria podczas commita – odtwarza się stan pojedynczego serwera
- Gdy wszystkie zmieniane węzły są na jednym serwerze – można szybciej wywołać commit - 1-fazowo

Transakcje

Function ValidateUpdate(V, U)

Input: V – objects to validate
Input: U – objects to update
Output: if objects in V are up to date then
return OK, otherwise return $\perp$
Atomically execute the following:
foreach $(i, obj) \in V$ **do**
  if object i at server differs from obj **then**
    return $\perp$
  end
end
foreach $(i, obj) \in U$ **do**
  write obj to object i at server
end
return OK

Transakcje

Function Write(n, obj)

Input: i – object ID
Input: obj – object
writeSet$[i] \leftarrow obj$
return OK

Function Read(i)

Input: i – object ID
Output: object with ID i, or else $\perp$ if transaction aborted

```
if writeSet[i] ≠ ⊥ then
    return writeSet[i]
else if localReplicas[i] ≠ ⊥ then
    readSet[i] ← localReplicas[i]
    return localReplicas[i]
else
    Ret ← ValidateFetch(readSet, i)
    if Ret = ⊥ then
        Abort ()
        return ⊥
    else
        readSet[i] ← Ret
        if Ret is an inner tree node then
            localReplicas[i] ← Ret
        end
        return Ret
    end
end
```

Transakcje

Function ValidateFetch(V, i)

Input: V – objects to validate
Input: i – ID of object to fetch
Output: if objects in V are up-to-date then
return object i, otherwise return $\perp$
Atomically execute the following:
foreach $(j, obj) \in V$ **do**
  if object j at server differs from obj **then**
    return $\perp$
  end
end
$Ret \leftarrow$ value of object i at server
return Ret

***

- Zazwyczaj operacje mają mały zbiór *write* na pojedynczym serwerze i duży zbiór *read* na wielu serwerach
- Na przykład Insert zwykle pisze tylko do liścia, ale czyta kilka węzłów, zaczynając od korzenia

- Dołożenie wartości do niepełnego liścia to bardzo popularna operacja
- Może się ona wykonać w jedno- (a nie dwu-) fazowym commit

- Read i Write wykonujemy wywołując atomowe odpowiedniki
- Alokacja też przebiega atomowo – żeby uniknąć wycieków pamięci i wyścigów

***

- GetRoot – trzymamy w meta-danych. Obiekt zawierający meta-dane także należy do zbioru *read* transakcji
- Gdy serwer jest usuwany z systemu, zaznaczamy odpowiednią flagę w meta-danych

- Serwery posiadają gotowe pule pamięci pod obiekty
- Klienci trzymają (leniwie odświeżane) maski bitowe – one oznaczają, które miejsca pod obiekty są wykorzystane
- Przy alokacji maska bitowa jest dodawana zarówno do zbioru *write* jak *read* transakcji

Zarządzanie systemem

- Migracja – wywołanie transakcji, która czyta węzły z serwera s1 i zapisuje je na serwerze s2
- Dodanie serwera:
- Wypełniamy tablicę wersji węzłów
- Zbieramy - bez transakcji – numery od innych węzłów
- Zapisujemy max dla każdego węzła

- Usunięcie serwera
- Oznaczamy serwer jako usunięty
- Przenosimy węzły na inne serwery, np. losowo
- Transakcje uprzednio rozpoczęte na tym węźle się nie wykonają, bo meta-dane nie będą aktualne

Złożoność pamięciowa

- Pamięć w przybliżeniu równo podzielona między serwerami
- O(liczba_serwerów^2) na meta-dane
- O(liczba_lisci*rozmiar_lisci) na przechowywane dane; ułamek tego na tablice z numerami wersji węzłów i maski bitowe alokatora

Złożoność komunikacji

- Lookup potrzebuje jednego „*round-trip*”, aby odczytać wartość liścia i jednocześnie sprawdzić aktualność węzłów po drodze
- Insert i Delete potrzebują 2 takich „*wycieczek*”, aby wpierw odczytać wartość liścia, a następnie zatwierdzić jego zmianę
- W najgorszym przypadku wszystkie węzły u klienta na ścieżce do liścia są nieaktualne i potrzeba h (wysokość B-drzewa) „*wycieczek*”
- Może się zdarzyć duża gęstość transakcji – wtedy je powtarzamy do skutku (nie zdarza się to często)
- Liczba komunikatów podczas jednego „round-trip” jest stała (koszt zamortyzowany)

Testy

- Klaster 24 maszyn, 1GHz PentiumIII, Gb karty sieciowe
- Węzły po 4KB, klucze/wartości po 10/8 bajtów (ok 200/węzeł)
- Drzewa wypełnione 100,000 elementami
- Klienci wywołali po 10,000 operacji Insert, Lookup i Delete

- Średnia liczba „round-trips” (na próbce 10,000) wyniosła
- 2.0 – 2.2 dla Insert()
- 1.000 – 1.001 dla Lookup()
- 2.1 – 2.6 dla Delete()

- Wyniki bardzo sensowne – w Delete i Insert zawsze trzeba najpierw pobrać liść, potem go zmieniać

Testy

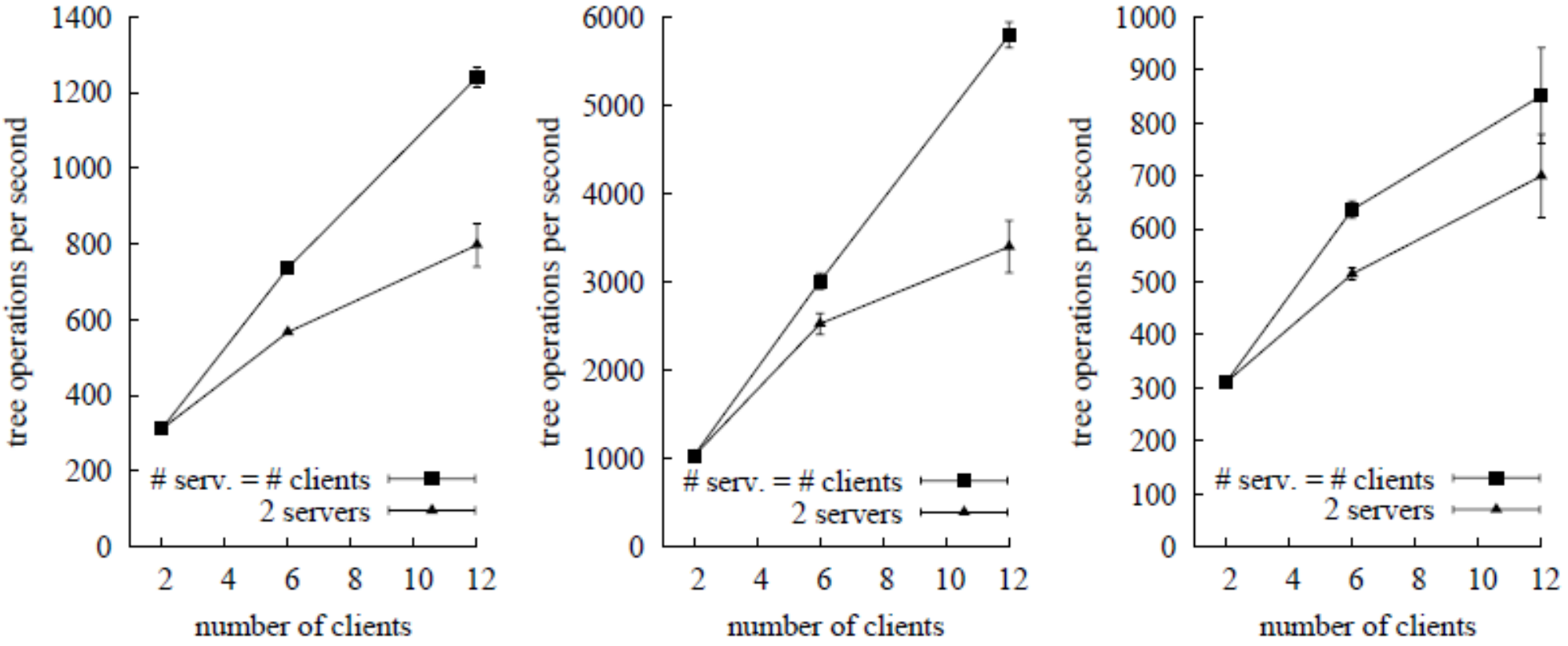


Figure 7: Aggregate throughput for Insert (left), Lookup (middle), and Delete (right) operations.

Więcej?

Insert (key, value)

```
rootNum <- getRoot()
(ret, modified, root) <- insertHelper(rootNum, key, value, -1)
if IsAborted() then return NULL
if root has too many keys then
    split root into children child and child2 and new root root
    c <- Alloc()
    c2 <- Alloc()
    Write(rootNum, root)
    Write(c, child)
    Write(c2, child2)
else if modified then
    Write(rootNum, root)
end
```

Search (node, key)

```
if node.numKeys == 0 then
    return NULL
else
    return index of the largest key in
    node.keys[1..numKeys] that does not exceed
    key, or else 0 if no such key
end
```

InsertHelper (n, key, value, depth)

```
node <- Read(n)
if node == NULL or node.depth <= depth then Abort()
i <- Search(node, key)
if node.isLeaf then
    if i <> 0 and node.keys[i] == key then
        node.values[i] <- value
        return (false, true, node)
    else
        insert key and value into node
        return (true, true, node)
    end
else ...
```

InsertHelper (n, key, value, depth)

```
... else // not node.isLeaf
    c <- node.children[i]
    (ret, modified, child) <- InsertHelper(c, key, value,
      node.depth)
    if IsAborted() then return NULL
    if child has too many keys then
        split child into child and child2, update node as needed
        c2 <- Alloc()
        Write(c, child)
        Write(c2, child2)
        return(ret, true, node)
    else if modified then
        Write(c, child)
    end
    return(ret, false, node)
end
```