Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Jarosław Wódka
Nr albumu: 262583

Analiza i pomiar wydajności rozproszonego systemu bazodanowego Gemius BigTable

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janina Mincer-Daszkiewicz
Instytut Informatyki

Czerwiec 2012

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora pracy

Streszczenie

System bazodanowy Gemius BigTable jest konfigurowalny na wielu poziomach. Wartości parametrów konfiguracyjnych przekładają się na takie czynniki jak czas wykonania operacji bazodanowych, zajmowany obszar dyskowy, a także dostępność danych. Dlatego odpowiednia konfiguracja jest niezbędna dla efektywnego wykorzystania systemu.

W pracy opiszę realizację modułu do testowania wydajności i skalowalności systemu Gemius BigTable. Przedstawię parametry konfiguracyjne systemu i opiszę ich wpływ na zachowanie systemu. Zdefiniuję miary wydajności oraz zbadam ich zachowanie dla różnych wartości parametrów konfiguracji.

Słowa kluczowe

rozproszone systemy baz danych, wydajność, pomiar, analiza, Gemius BigTable, Google BigTable, HBase, Cassandra

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.4 Operating Systems
D.4.8 Performance

H. Information Systems
H.2 Database Management
H.2.4 Systems

Tytuł pracy w języku angielskim

Analysis and measurement of distributed database system Gemius BigTable

Spis treści

Wprowadzenie 7

1. Testowanie wydajności bazodanowych systemów rozproszonych 9
1.1. Ogólna charakterystyka systemów NoSQL 9
1.2. Decyzje projektowe 10
1.2.1. Odczyt/zapis 10
1.2.2. Opóźnienie/trwałość 10
1.2.3. Synchroniczna replikacja/asynchroniczna replikacja 10
1.2.4. Ułożenie danych 11
1.3. Dostępne rozwiązania NoSQL 11
1.3.1. Google BigTable 11
1.3.2. Cassandra 13
1.3.3. HBase 14
1.4. Dostępne środowiska testujące 14
1.4.1. DieCast 14
1.4.2. DiPerF 15
1.5. Yahoo! Cloud Serving Benchmark 15
1.5.1. Testowanie wydajności 16
1.5.2. Testowanie skalowalności 16
1.5.3. Model testowania 16
1.5.4. Rozkłady 17
1.5.5. Architektura systemu 17

2. Gemius BigTable 19
2.1. Architektura 19
2.2. Biblioteka CommonLib 20
2.3. Moose File System 21
2.4. Logiczny model danych 23
2.5. API 25
2.6. Fizyczny model danych 26
2.7. Odporność na awarie 29

3. Narzędzia testujące 31
3.1. Wymagania 31
3.2. Integracja z YCSB 31
3.2.1. Interfejs rozszerzenia 32
3.2.2. JNI 32
3.2.3. Implementacja interfejsu Gemius BigTable 33

3.3. Symulator obciążenia tabletu 34
3.3.1. Interfejs symulatora 35
3.3.2. Polityki łączenia 35
3.3.3. Symulacja operacji bazodanowych 37
3.3.4. Ewaluacja kosztu odczytu danych 38

4. Wyniki testów 39
4.1. Testy z wykorzystaniem YCSB 39
4.1.1. Środowisko testowe 39
4.1.2. Ogólny opis 40
4.1.3. Testowanie opóźnienia 40
4.1.4. Skalowalność 48
4.1.5. Wnioski 50
4.2. Testowanie polityki łączenia 51
4.2.1. Ogólny opis 51
4.2.2. Wyniki testów 52
4.2.3. Wnioski 52

5. Podsumowanie 53

A. Zawartość płyty 55

Bibliografia 57

Spis rysunków

1.1. Przydział kluczy do serwerów w systemie Cassandra 13
1.2. Mechanizm testowania DieCast 14
1.3. Przegląd architektury systemu DiPerF 15
1.4. Architektura aplikacji klienckiej YCSB Client 18

2.1. Gemius BigTable — komponenty i komunikacja 19
2.2. Architektura Gemius BigTable — warstwy systemu 20
2.3. Struktura pliku RTL 21
2.4. Podział tabeli na tablety w Gemius BigTable 24
2.5. Odczyt danych w Gemius BigTable 28

4.1. Intensywna aktualizacja i odczyt, wykres opóźnienia 42
4.2. Intensywny zapis, wykres opóźnienia 43
4.3. Krótkie przedziały, wykres opóźnienia 44
4.4. Długie przedziały, wykres opóźnienia 45
4.5. Operacje modyfikujące, wykres opóźnienia 46
4.6. Operacje niemodyfikujące, wykres opóźnienia 46
4.7. Opóźnienie operacji dla różnych rozkładów, GBT 47
4.8. Opóźnienie operacji dla różnych rozkładów, Cassandra 47
4.9. Wykres opóźnienie w zależności od ilości serwerów 48
4.10. Wykres przepustowość w zależności od liczby serwerów 49
4.11. Wykres opóźnienia odczytu przy podłączeniu dodatkowego serwera 50
4.12. Opóźnienie odczytu ze względu na politykę łączenia 52

Wprowadzenie

W ostatnich latach można było zaobserwować bardzo dynamiczny rozwój rozproszonych systemów bazodanowych. Opublikowanie przez firmę Google dokumentu opisującego rozproszony system bazodanowy BigTable [Cha08] zainspirowało wiele firm do stworzenia własnych implementacji rozproszonych systemów bazodanowych. Dało to początek takim systemom jak MongoDB, Cassandra, czy HBase. Firma Gemius S.A. także rozpoczęła projekt Gemius BigTable bazujący na tej publikacji.

Dostępne rozwiązania bazodanowe wolnego oprogramowania (ang. *open source*) prezentują różny model danych, a niektóre z nich korzystają z rozproszonych systemów plików. Wszystkie te cechy mają wpływ na czas wykonywania różnych operacji na systemie bazodanowym. Autorzy systemów dokonują oceny wydajności swoich produktów. Jednakże porównanie różnych systemów między sobą wymaga platformy pozwalającej testować rozproszone systemy bazodanowe na równych warunkach. Możliwość przeprowadzania automatycznych testów wydajnościowych Gemius BigTable na tle konkurencyjnych rozwiązań jest kluczowa dla wyznaczania dalszych kierunków rozwoju projektu.

Oferowany przez Gemius BigTable interfejs programistyczny jest bardzo podstawowy. Nie oznacza to jednak, że sam system nie może być dostrajany do potrzeb użytkownika. Wręcz przeciwnie, Gemius BigTable oferuje szerokie spektrum parametrów konfiguracyjnych. Administrator bazy określić może wysokopoziomowe polityki, ograniczenia systemu, a nawet sięgające modelu danych metody kompresji i serializacji. Wszystkie te parametry mają bezpośredni wpływ na efektywność wykonywania operacji bazodanowych. Odpowiedzenie na pytanie, które parametry zwiększają efektywność w poszczególnych zastosowaniach jest szalenie istotne, gdyż przekłada się to bezpośrednio na szybsze przetwarzanie procesów biznesowych.

Cel pracy

Celem mojej pracy jest dobranie odpowiednich narzędzi niezbędnych do wykonania testów mierzących wydajność systemu Gemius BigTable. W ramach tego planuję zintegrować system Gemius BigTable z dostępnym środowiskiem testowym. Z użyciem wybranego środowiska zamierzam opracować i przeprowadzić szereg testów uwypuklających silne i słabe strony systemu Gemius BigTable. Aby otrzymane wyniki były bardziej wiarygodne planuje przeprowadzone testy wykonać także na jednym z ogólnodostępnych rozwiązań bazodanowych.

Struktura dokumentu

Dokument podzielony jest na trzy rozdziały. W następnym rozdziale przedstawiłem ogólną charakterystykę systemów NoSQL oraz opisałem najpopularniejsze, dostępne rozwiązania na rynku. Przedstawiłem również aspekty systemów NoSQL mające wpływ na wydajność oraz środowisko do testowania wydajności systemów NoSQL. W rozdziale drugim opisałem system Gemius BigTable oraz systemy, których wymaga do pracy w środowisku rozproszonym.

Opisałem sposób w jaki zintegrowałem ten system ze środowiskiem testowym oraz symulator obciążenia. W trzecim rozdziale przedstawiłem wyniki testów jakie przeprowadziłem na systemie Gemius BigTable oraz Cassandra. Następnie opisałem testy przeprowadzone z użyciem symulatora oraz powtórzone te same testy w środowisku rozproszonym.

Podziękowania

Pragnę podziękować Pani dr Janinie Mincer-Daszkiewicz za wsparcie i prowadzenie pracy. Dziękuję również firmie Gemius S.A. za umożliwienie mi zrealizowania pracy magisterskiej na temat związany z wewnętrznymi systemami oraz dostarczenie architektury niezbędnej do przeprowadzenia testów. Podziękowania składam także na ręce moich Współpracowników, z którymi miałem przyjemność współpracować podczas rozwoju Gemius BigTable: Mariusza Gądarowskiego, Jakuba Bogusza, Kamila Nowosada, Tomasza Wekseja, Konrada Witkowskiego, Tomasza Żołnowskiego oraz wszystkich pracowników firmy Gemius, którzy zaangażowani byli w rozwój systemów i bibliotek opisanych w tym dokumencie.

Oświadczenie

Wszystkie fragmenty kodów źródłowych systemów Gemius BigTable, BigTableLib, CommonLib zawarte w pracy są wyłączną własnością Grupy Gemius S.A.

Rozdział 1

Testowanie wydajności bazodanowych systemów rozproszonych

W rozdziale tym opiszę ogólną charakterystykę systemów NoSQL oraz decyzje podjęte przy ich projektowaniu, które mają znaczący wpływ na wydajność systemów. Następnie przyjrzę się najpopularniejszym rozwiązaniom NoSQL i temu, jak optymalizacja jednego parametru systemu może mieć wpływ na inny aspekt wydajnościowy. Pod koniec rozdziału przybliżę YCSB [Coo10] (rozwiązanie firmy Yahoo!) oraz sposób w jaki autorzy tego produktu starają się usystematyzować testowanie wydajności bazodanowych systemów rozproszonych.

1.1. Ogólna charakterystyka systemów NoSQL

Termin NoSQL określa klasę systemów bazodanowych, które składują nierelacyjne dane [Pok11]. Czasem termin ten jest używany także w odniesieniu do baz XML, czy do baz danych reprezentujących grafy. W dalszych rozważaniach zajmować się będę jednak tylko najpopularniejszymi systemami NoSQL, tj. systemami przechowującymi dane w postaci klucz-wartość.

Systemy NoSQL zwykle mają uproszczony interfejs w porównaniu z klasycznymi relacyjnymi systemami zarządzania baz danych (RDBMS). Dane w takich systemach wprawdzie są zwykle organizowane w tabele, jednak dostęp do nich jest tylko przy pomocy klucza. Ponadto systemy NoSQL zwykle nie implementują operacji złączenia ani `ORDER BY`. Jest to powodowane tym, że dane są rozmieszczone zwykle na wielu serwerach. Ponadto nie dają one zwykle pełnej gwarancji ACID. Najczęściej zamiast spójności gwarantowana jest tylko spójność ostateczna. Oznacza to, że spójność jest osiągana, gdy na bazie nie są wykonywane modyfikacje przez dostatecznie długi okres czasu.

Systemy NoSQL zazwyczaj mają architekturę rozproszoną na wiele serwerów. Pozwala to na tworzenie większych baz danych niż w przypadku tradycyjnych RDBMS. Często dane trzymane są redundantnie na kilku serwerach. Dzięki temu system taki może być odporny na awarie, czyli usterka jednego z serwerów nie zakłóca prawidłowego działania systemu.

Model danych w systemach bazodanowych NoSQL zwykle stanowi zbiór par klucz-wartość. Baza danych jest więc zbiorem nazwanych wartości, które są jednoznacznie identyfikowane przez unikatowy klucz. Wartościami zwykle jest napis, ustruktyryzowany obiekt lub pozbawiony struktury BLOB.

Systemy NoSQL są zatem szczególnie użyteczne w sytuacjach, gdzie przetwarzane są ogromne ilości danych, które nie są ze sobą relacyjnie powiązane. Przykładem takich danych

mogą być dane do analizy statystycznej.

1.2. Decyzje projektowe

Projektując system bazodanowy trzeba podjąć szereg decyzji projektowych. Część z nich będzie miało wpływ na efektywność wykonywania poszczególnych operacji. Jeśli system optymalizowany jest pod pewne funkcjonalności zwykle pociąga za sobą większą kosztowność wykonania innych. Główne decyzje związane z wydajnością, jakie należy podjąć przy projektowaniu bazodanowego systemu rozproszonego przedstawię w kolejnych sekcjach [Coo10].

1.2.1. Odczyt/zapis

Operacje na danych w systemach bazodanowych często wymagają odczytu lub zapisu pojedynczych rekordów. Podczas przetwarzania trudno jest przewidzieć, na których rekordach wykonywane będą następne operacje. Skoro nie wiadomo, które rekordy będą następnie używane, system nie może wcześniej zlecić szeregu asynchronicznych operacji odczytu z dysku. W konsekwencji prowadzi to do wykonywania sekwencyjnych operacji wejścia/wyjścia. Losowe odczyty lub zapisy są szczególnie nieefektywne na standardowych dyskach twardych, na których opiera się obecnie architektura najpopularniejszych systemów NoSQL. Dlatego też system może zostać zoptymalizowany pod kątem losowych odczytów lub zapisów.

Znacznie większa efektywność losowego zapisu rekordów do bazy danych może zostać osiągnięta przez dopisywanie logów z różnicami jakie zostały wprowadzone. Zapis ten jest wykonywany w sposób ciągły, co znacząco podnosi efektywność modyfikacji rekordów. Nie mniej jednak, ma to wpływ na efektywność odczytu danych. Podczas pobierania rekordu trzeba odczytać zarówno sam rekord, jak i różnice, które zostały na niego naniesione. Dopiero rekord z naniesionymi zmianami jest przekazywany użytkownikowi. Powszechnie stosowanym rozwiązaniem, mającym na celu zniwelowanie narzutu na odczyt, jest utrzymywanie procesu, który w tle łączy rekordy z naniesionymi wcześniej zmianami. To jednak z kolei może mieć wpływ na efektywność innych operacji.

1.2.2. Opóźnienie/trwałość

Podczas modyfikacji rekordów dane mogą być synchronizowane z trwałym nośnikiem przed przekazaniem użytkownikowi kodu wykonania operacji, jak również mogą być trzymane w pamięci aż do momentu, gdy synchronizacja z dyskiem jest konieczna. Pierwsze podejście gwarantuje, że wszystkie zmiany, jakie wprowadzi użytkownik, zostaną zachowane w systemie nawet w przypadku awarii serwera. Nie mniej jednak ma to istotny wpływ na efektywność operacji zapisujących. Leniwe odsyłanie zmian na dysk jest jeszcze bardziej efektywne w sytuacji, gdy wykonywanych jest wiele modyfikacji na tym samym rekordzie. Niestety efektywność jest uzyskana kosztem bezpieczeństwa. W chwili awarii serwera, zostają utracone wszystkie zmiany, które były tylko w pamięci operacyjnej.

1.2.3. Synchroniczna replikacja/asynchroniczna replikacja

Replikacja w rozproszonych systemach bazodanowych jest procesem redundantnego powielenia danych mającym na celu:

- **zapewnienie odporności na awarie**: W przypadku awarii jednego z serwerów, brakujące repliki są odzyskiwane z serwerów posiadających kopie danych.

- **zwiększenie dostępności danych**: W przypadku awarii serwera dane, które się na nim znajdowały dostarczane są z serwera przechowującego odpowiednie repliki.
- **poprawienie wydajności**: Repliki mogą być wykonywane na serwery znajdujące się w różnych miejscach na świecie. W ten sposób dane mogą być odczytywane z serwera, który znajduje się najbliżej użytkownika.

W systemach rozproszonych synchronizacja między replikami może mieć charakter synchroniczny lub asynchroniczny. Synchronizacja replik w momencie modyfikacji rekordu zapewnia, że wszystkie repliki posiadają aktualne dane. Nie mniej jednak może mieć to istotny wpływ na opóźnienia podczas operacji modyfikujących. Co więcej, w sytuacji, gdy pewne repliki są niedostępne, operacja może się w ogóle nie powieść.

W przypadku replikacji asynchronicznej taki narzut wydajnościowy nie występuje. Nie mniej jednak, jeśli dojdzie do awarii serwera, który nie zdążył jeszcze wykonać kopii zapisów zatwierdzonych już zmian, może dojść do utraty danych.

1.2.4. Ułożenie danych

Dane w systemach bazodanowych mogą być ułożone wierszowo lub kolumnowo. W przypadku ułożenia danych w sposób wierszowy, kolejne rekordy zajmują ciągłą przestrzeń dyskową. Pozwala to na istotnie szybszy dostęp przy odwoływaniu się do całego rekordu. Zysk ten jest szczególnie zauważalny w przypadku odwoływania się do pojedynczych rekordów, ponieważ w takiej sytuacji najdłużej trwa wyszukiwanie danych na dysku. To wyszukiwanie zaś musi być wykonane tylko raz.

W przypadku systemów zorientowanych kolumnowo różne kolumny mogą być trzymane w różnych plikach, a nawet na różnych serwerach. Podejście to daje istotnie lepsze wyniki w przypadku, gdy użytkownika interesuje tylko pewien podzbiór kolumn. Różnica jest szczególnie wyraźna w przypadku odczytywania większej ilości danych, ponieważ w takiej sytuacji odnalezienie danych na dysku stanowi zaniedbywalny koszt. Popularną techniką realizacji tego podejścia jest pogrupowanie danych w rodziny kolumn (ang. *column family*). Pomysł ten polega na grupowaniu w rodzinę kolumn te kolumny, które są w tabeli ze sobą logicznie powiązane (np. imię i nazwisko). Dzięki temu dostęp do podzbioru kolumn, które są ze sobą logicznie powiązane jest istotnie szybszy, niż czytanie całego rekordu.

1.3. Dostępne rozwiązania NoSQL

W ostatnich latach powstało wiele rozproszonych systemów składujących dane. W każdym z nich zostały podjęte specyficzne decyzje projektowe, które mają bezpośredni wpływ na ich wydajność. W rozdziale tym opiszę kilka najbardziej popularnych rozwiązań na rynku. Dla każdego z nich wskażę pod jakim kątem zostało zoptymalizowane oraz jakie decyzje projektowe musiały zostać podjęte w celu osiągnięcia zamierzonego celu. Przyjrzę się następującym rozwiązaniom: BigTable, Cassandra, HBase.

1.3.1. Google BigTable

BigTable [Cha08] jest rozproszonym systemem składowania danych wytworzonym przez firmę Google w 2006 roku. System został stworzony do przechowywania petabajtów danych na tysiącach serwerów. Firma Google traktuje BigTable jako projekt wewnętrzny i nie upubliczniła jego kodu źródłowego. Produkt jest jednak szeroko wykorzystywany wewnątrz firmy. Został wdrożony w takich produktach jak: Google Analytics, Google Finance, Google Earth, czy

App Engine. W tym ostatnim można pośrednio korzystać z BigTable, ponieważ został on użyty do składowania danych w projekcie App Engine [Bar10].

Modelem danych w Google BigTable jest zbiór tabel. Tabele te różnią się istotnie od modelu relacyjnego. Schemat takiej tabeli może być rozumiany jako rzadki, rozproszony, wielowymiarowy, posortowany słownik. Sygnatura tego modelu danych jest następująca:

```
(row:string, column:string, time:int64) -> string
```

Dane w tabeli utrzymywane są w rosnącym porządku leksykograficznym kluczy. Przestrzeń kluczy jest podzielona na spójne przedziały kluczy. Jeden taki przedział w systemie to *tablet*. Podział ten jest modyfikowany dynamicznie podczas działania systemu.

BigTable działa w architekturze rozproszonej. BigTable nie wymaga dedykowanych maszyn. Procesy tego systemu mogą pracować na serwerach razem z innymi aplikacjami. W systemie BigTable zarówno dane, jak i logi składowane są w rozproszonym systemie plików GFS (Google File System)[Ghe03]. W systemie znajduje się serwer zarządzający (ang. *master server*) oraz serwery obsługujące dane (ang. *tablet server*). Serwer zarządzający jest odpowiedzialny za zarządzanie tabletami (przydział tabletów do serwerów danych, podział tabletów, łączenie tabletów), zarządzanie serwerami tabletów (dodanie serwera tabletów, awaria serwera tabletów) oraz zarządzanie schematem danych. Serwery danych są natomiast odpowiedzialne za obsługę tabletów, które zostały im przydzielone przez serwer zarządzający. Serwery te wykonują fizyczną modyfikację danych oraz obsługują zapytania klientów.

Tabele w systemie podzielone są na tablety. Tabela po utworzeniu składa się tylko z jednego tabletu. Gdy rozmiar tabletu się zwiększy, dochodzi do jego podziału na dwa mniejsze. Serwer zarządzający zleca podział tabletu, gdy jego rozmiar będzie wynosić 100–200 MB. Gdy rozmiar tabletów zmaleje, serwer zarządzający zleca łączenie tabletów.

Tablety fizycznie składają się z plików zapisanych w formacie *SSTable*. Jest to format reprezentujący uporządkowany, niemodyfikowalny słownik z kluczy w wartości. Format ten dostarcza operacji odczytu wartości dla zadanego klucza oraz skanowanie danych z zadanego zakresu kluczy. Dane fizycznie podzielone są na bloki. Podczas odczytu danych z pliku, z dysku czytany jest cały blok. Na końcu pliku znajduje się indeks, który pozwala na szybką lokalizację bloku. Podczas otwierania pliku, cała zawartość indeksu jest wczytywana do pamięci. Dzięki temu zapytania o kolejne klucze wymagają tylko jednego żądania odczytu danych z dysku.

BigTable trzyma swój trwały stan w GFS. Zmiany, które zostają wykonane w tabeli są reprezentowane w pliku w postaci logu. Najnowsze zmiany przetrzymywane są w posortowanym schowku (ang. *memcache*) w pamięci operacyjnej. Starsze zmiany są zrzucane do GFS, do pliku SSTable. Dane w schowku oraz plikach SSTable logicznie reprezentują jeden zbiór posortowanych danych. Podczas zapytania serwer tabletów łączy w pamięci dane z tych źródeł. Złączeniem tym jest suma danych ze schowka oraz plików SSTable.

Wraz z kolejnymi operacjami wykonywanymi na tablecie, liczba plików SSTable w tablecie rośnie. Nadmierna liczba plików w tablecie powodowałaby sytuację, w której odczyt stawałby się mniej efektywny, ponieważ trzeba by wykonać operacje dyskowe na wielu plikach. Dlatego też w tle wykonywane jest okresowe łączenie plików SSTable. W systemie występują małe upakowania (ang. *minor compaction*), upakowania łączące (ang. *merging compaction*) oraz duże upakowania (ang. *major compaction*). Małe upakowania polegają na zrzuceniu zawartości schowka do formatu SSTable. Po zrzuceniu danych, aktualny schowek oraz odpowiadający mu log zmian nie są już potrzebne i zostają zwolnione. Upakowanie łączące wykonuje przepisanie kilku plików SSTable do jednego pliku. Duże upakowanie polega na złączeniu danych z wszystkich plików SSTable występujących w tablecie w jeden plik. Jest ono istotne, ponieważ podczas usuwania danych z tabeli, w pliku SSTable zapisywany jest rekord o zadanym kluczu

ze znacznikiem informującym, że rekord zostaje usunięty. Duże złączenie pozwala na odzyskanie zasobów dyskowych, ponieważ tylko łączenie wszystkich plików pozwala na fizyczne usunięcie rekordu z tabeli. Podczas wykonywania każdej z opisanych wcześniej operacji dane wciąż są dostępne dla klienta do odczytu oraz zapisu.

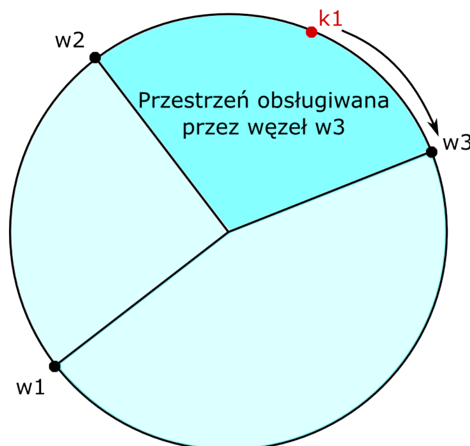
1.3.2. Cassandra

Cassandra to rozproszony system zarządzania bazą danych, którego powstanie zostało zainicjowane w firmie Facebook [Lak10]. Głównym celem projektowym było zarządzanie ogromną ilością danych rozproszonych na tysiącach serwerów.

Tabelą w systemie Cassandra jest rozproszony, wielowymiarowy słownik. Mapowanie indeksowane jest kluczem o typie string, którego długość nie przekracza zwykle 36 bajtów. Wartością w słowniku są natomiast ustrukturyzowane obiekty. Odczyt danych wykonywany jest atomowo w obrębie pojedynczego wiersza.

Podobnie jak w Google BigTable, dane w Cassandrze zorientowane są kolumnowo. Kolumny łączone są w rodziny kolumn. W systemie dostępne są dwa rodzaje rodzin kolumn: *Super Column family* oraz *Simple Column family*. Super Column family może być rozumiane jako rodzina kolumn rodzin kolumn. W odróżnieniu jednak od Google BigTable, dane w systemie mogą być sortowane albo po nazwie, albo po czasie.

W systemie Cassandra przestrzeń kluczy jest zarządzana cyklicznie. Z kluczy wyliczane są hasze przez funkcję haszującą zachowującą porządek na kluczach. Każdemu z węzłów systemu przypisywany jest losowy punkt w tej przestrzeni cyklicznej. Reprezentuje on pozycję tego węzła w przestrzeni cyklicznej. Węzeł ten odpowiedzialny jest za zarządzanie kluczami, których hasze są mniejsze niż jego własny hasz, ale nie mniejsze niż hasz wartości poprzedniego węzła systemu (por. rys. 1.1). Zaletą takiego podejścia jest fakt, że dodanie lub usunięcie węzła ma wpływ tylko i wyłącznie na poprzedni oraz następny węzeł w systemie.



Rysunek 1.1: Schemat reprezentujący przydział kluczy do serwerów. Klucz k1 jest zarządzany przez węzeł w3

Cassandra ma wbudowany system replikacji. Dla każdej instancji ustala się liczbę kopii danych, które muszą znajdować się w systemie. Dla klucza k serwerem odpowiedzialnym za utrzymywanie replik jest serwer zarządzający kluczem k. Mechanizm ten parametryzowany jest polityką wyznaczania serwera do składowania repliki. System może wyznaczyć serwer w trybie "Rack Unaware", gdzie repliki trafiają do kolejnych serwerów w cyklicznej przestrzeni kluczy. W przypadku polityk "Rack Aware", czy "Datacenter Aware" algorytm selekcji

serwera jest bardziej wyrafinowany i dba o to, aby repliki nie znajdowały się w tej samej lokalizacji.

Fizyczny model danych w systemie jest podobny do tego z Google BigTable. W momencie zapisu danych do tabeli, serwer zapisuje na dedykowanym dysku twardym plik ze zmianami oraz modyfikuje w pamięci odpowiadające struktury danych. Okresowo zawartość pamięci jest zrzucana do pliku na jeden z dysków z danymi. Aby ograniczyć liczbę plików na dysku, pliki są okresowo przepisywane i łączone w większe pliki.

1.3.3. HBase

HBase jest projektem wolnego oprogramowania (ang. *open source*) rozwijanym obecnie przez firmę Apache [ApaHB]. Projekt ten jest otwartą implementacją napisaną w języku Java systemu Google BigTable.

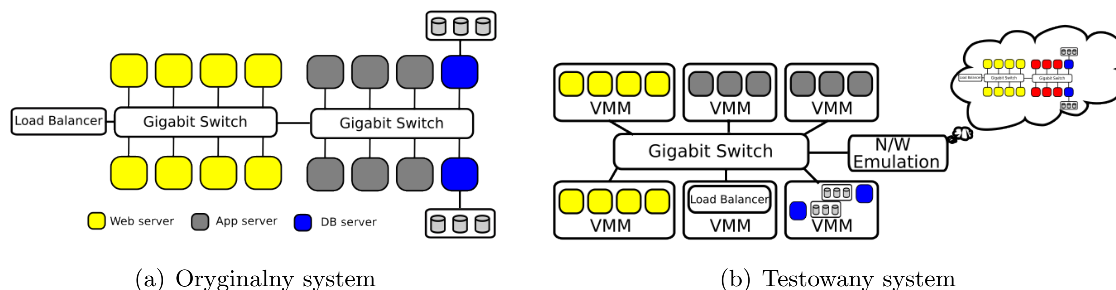
System ten jest odporny na awarie, jednak aby zapewnić tę własność musi on w środowisku rozproszonym składować swoje dane na rozproszonym systemie plików HDFS. HDFS jako system plików nie zapewnia szybkiego dostępu do dowolnych rekordów. Dopiero HBase działający z wykorzystaniem HDFS wykonuje efektywnie operacje na rekordach, nawet dla bardzo dużych wolumenów danych. HBase optymalizuje dostęp do danych za pomocą pamięci podręcznej (ang. *cache*) bloków danych oraz filtrów Blooma. Praca z danymi odbywa się przy pomocy API dostarczonego dla języka Java. Nie mniej jednak wspiera równoległy model programowania MapReduce.

1.4. Dostępne środowiska testujące

W tym podrozdziale przedstawię pokrótce dostępne środowiska do testowania rozproszonych systemów bazodanowych. Z kolei podrozdział 1.5 poświęciłem opisowi rozwiązania wybranego do przeprowadzenia testów: YCSB.

1.4.1. DieCast

DieCast to system stworzony na Uniwersytecie Kalifornijskim [Gup11]. Głównym zadaniem tego środowiska jest ocena wydajności dużego systemu przy wykorzystaniu znacznie mniejszych zasobów. Krok w kierunku oszczędnego testowania i przewidywania zachowania dużych systemów rozproszonych został zrealizowany poprzez symulowanie serwerów maszynami wirtualnymi (por. rys. 1.2).



(a) Oryginalny system

(b) Testowany system

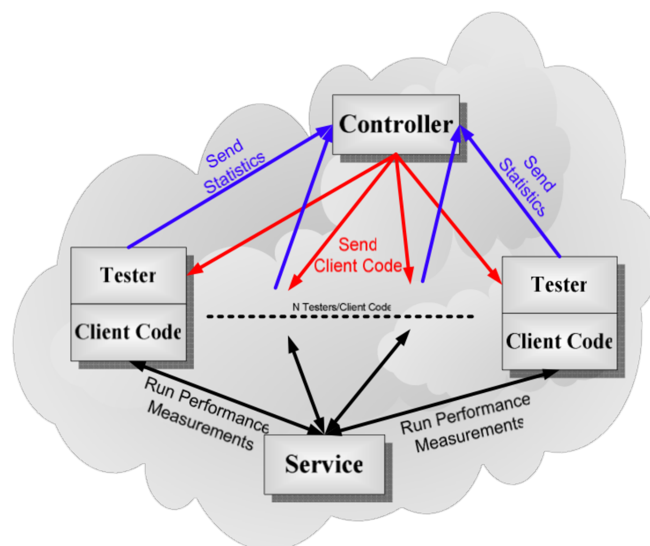
Rysunek 1.2: Mechanizm testowania DieCast [Gup11]

Rysunek 1.2(a) przedstawia przykładową architekturę systemu, którego wydajność ma zostać oceniona. DieCast wykonuje to zadania w sposób przedstawiony na sąsiednim rysunku. Serwery są symulowane na maszynach wirtualnych(*VMM*), natomiast pierwotna architektura sieci symulowana jest przez emulator sieci (*N/W Emulation*).

DieCast nie mógł zostać użyty do oceny systemu Gemius BigTable, ponieważ system plików MooseFS (więcej o tym systemie można przeczytać w podrozdziale 2.3) nie działa dobrze na wirtualnych maszynach.

1.4.2. DiPerF

Według autorów DiPerF jest narzędziem do testowania systemów rozproszonych [Dum04]. Celem stworzenia produktu było dostarczenie prostego środowiska do przeprowadzania automatycznych testów systemów rozproszonych. Narzędzie to zarządza zbiorem dostępnych serwerów użytych do przetestowania systemu, zbiera dane związane z wydajnością, a na ich podstawie generuje zbiorcze statystyki.



Rysunek 1.3: Przegląd architektury systemu DiPerF [Dum04]

Środowisko to składa się z dwóch głównych komponentów (por. rys. 1.3), tj. kontrolera (ang. *controller*) oraz testerów (ang. *testers*). Na każdej dostępnej maszynie uruchomiony jest tester, który odpowiada za koordynację obliczeń na serwerze. Użytkownik dostarcza kontrolerowi adres usługi, która ma zostać przetestowana, a także kod, który ma zostać wykonany na dostępnych serwerach. Kontroler jest odpowiedzialny za dystrybucję kodu, pobranie od testerów pomiarów wydajności na poszczególnych węzłach oraz agregowanie otrzymanych wyników.

System był z powodzeniem testowany na ponad 100 maszynach w środowisku PlanetLab. Nie wybrałem go jednak do przeprowadzenia testów, ponieważ jest narzędziem bardziej ogólnego przeznaczenia. Co za tym idzie, nie dostarcza abstrakcji bazodanowych.

1.5. Yahoo! Cloud Serving Benchmark

W ostatnich latach powstało wiele systemów NoSQL. Twórcy każdego z tych systemów prezentowali swoje wyniki testów wydajnościowych. Systemy te znacząco różnią się od systemów,

dla których zostały stworzone scenariusze testowe do porównywania wydajności, jak na przykład TPC-C. Dlatego też brakowało spójnego środowiska do testowania ich w jednakowych warunkach. To też było motywacją do zrealizowania w firmie Yahoo! projektu *Yahoo! Cloud Serving Benchmark* (YCSB) [Coo10]. Wraz z systemem dostarczony jest podstawowy zestaw scenariuszy testowych oraz interfejsów do popularnych systemów NoSQL. Ponadto system jest łatwo rozszerzalny, co zdaniem autorów jest jedną z kluczowych funkcjonalności. Istotnie, w dokumencie opisującym system oraz w dokumentacji YCSB jest dokładnie opisane jakie czynności trzeba wykonać, aby wzbogacić system o obsługę kolejnych baz danych oraz dodać nowe scenariusze testowe. Aby wykorzystać te zalety oraz promować dalszy rozwój produktu, YCSB został wydany na licencji open-source.

1.5.1. Testowanie wydajności

Warstwa YCSB odpowiedzialna za testowanie wydajności koncentruje się wokół testowania opóźnienia wykonywania zapytań na bazie danych względem żądanej przepustowości. Jest to szczególnie interesująca zależność, ponieważ wraz z rozrostem bazy danych, często zwiększa się liczba żądań na sekundę. Z kolei użytkownicy, którzy chcą uzyskać dane z bazy są w stanie zaakceptować tylko pewne, ustalone opóźnienia, szczególnie w sytuacji, gdy użytkownikiem jest człowiek. Niestety, najczęściej te dwie wartości są ze sobą skorelowane. Warstwa testowania wydajności ma za zadanie zbadać kompromis między przepustowością a opóźnieniem. Podczas testu badane jest opóźnienie w warunkach zwiększającego się obciążenia systemu. Przepustowość jest zwiększana, aż do momentu wysycenia możliwości systemu.

1.5.2. Testowanie skalowalności

Bardzo ważnym aspektem systemów NoSQL jest sposób, w jaki reagują na zwiększającą się liczba serwerów w systemie. Warstwa testująca skalowalność pozwala zbadać to zachowanie. Twórcy YCSB definiują dwie własności określające zdolność systemu do skalowania:

- *scaleup*: Własność ta określa, jak zachowuje się system, gdy liczba serwerów w instancji się zwiększa. Załóżmy, że w systemie zostaje zmierzona wydajność (np. opóźnienie). Następnie test jest powtarzany, ale przepustowość jest zwiększana proporcjonalnie do zwiększenia liczby serwerów. Jeżeli system ten osiągnie taką samą wydajność, to ma on wysoką wartość *scaleup*.

- *speedup*: Własność ta określa, jak system dostosowuje się do podłączenia do instancji kolejnego serwera. W teście na system nakładane jest pewne obciążenie. Następnie, w trakcie działania systemu, podłączany jest kolejny serwer. System, który ma dużą wartość speedup powinien zareagować zwiększeniem wydajności. Może nastąpić to natychmiast lub po pewnym okresie dostosowywania się systemu do nowych zasobów.

1.5.3. Model testowania

W YCSB proces testowania systemu odbywa się dwuetapowo:

1. *faza ładowania*: Podczas tego etapu YCSB zapełnia tabelę danymi, aby właściwy pomiar odbywał się na niepustej tabeli.

2. *faza pomiaru*: Na tym etapie odbywa się właściwy pomiar wydajności bazy danych. Faza ta jest zwykle dość krótka (kilkadziesiąt minut), w odróżnieniu od fazy ładowania, która może trwać kilka godzin.

Podczas wykonywania pomiarów na bazie danych wykonywane mogą być następujące operacje:

- *Zapis*: zapis rekordu do tabeli.
- *Aktualizacja*: aktualizuje wartość jednego z atrybutów rekordu.
- *Odczyt*: odczytuje albo cały rekord, albo pojedynczy atrybut rekordu.
- *Skanowanie*: wykonywany jest odczyt przedziału rekordów. Przy skanowaniu czytana jest pewna liczba danych poczynając od pewnego miejsca.
- *Usunięcie*: usuwa pojedynczy rekord w bazie danych.

1.5.4. Rozkłady

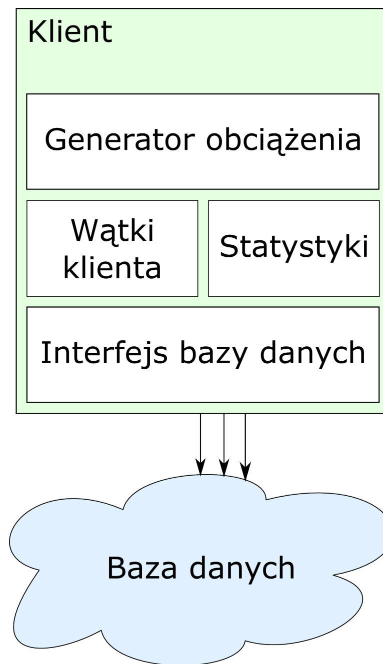
W systemach bazodanowych istotnym czynnikiem mającym wpływ na wydajność jest kolejność w jakiej rekordy są modyfikowane. Z punktu widzenia fizycznego modelu danych może być zupełnie czymś innym wykonywanie zawsze modyfikacji ostatnio dodanych rekordów, a czymś innym równomierne ich modyfikowanie. Aby umożliwić przetestowanie jaki ma to wpływ na system, YCSB udostępnia zestaw wbudowanych rozkładów określających, na których rekordach wykonywać operacje. Dostępne są następujące rozkłady:

- *jednostajny*: w rozkładzie tym podczas wyboru rekordu do modyfikacji, każdy wybierany jest z jednakowym prawdopodobieństwem;
- *zipfian*: w rozkładzie tym wybierana jest niewielka grupa elementów, która ma bardzo duże prawdopodobieństwo wylosowania, nieco większa grupa z dużym prawdopodobieństwem, jednak ogromna większość elementów ma bardzo małe prawdopodobieństwo wylosowania. Rozkład ten ma za zadanie modelować popularność występowania słów w języku, jak również wielkość miast na świecie;
- *najnowsze*: rekordy, które zostały dodane najpóźniej mają największe prawdopodobieństwo wylosowania.

Istotne jest, że popularność w przypadku rozkładu *najnowsze* zdobywają rekordy, które mają najpóźniejszy czas wrzucenia, a nie największą wartość klucza. Istotną różnicą między rozkładem *najnowsze* a *zipfian* jest zmiana prawdopodobieństwa w czasie. W *zipfian* przypisane do elementu prawdopodobieństwo utrzymuje się niezmiennie, natomiast w *najnowsze* w momencie powstania jest duże i wraz z pojawianiem się kolejnych rekordów maleje. Rozkłady te mają za zadanie symulować dwa zupełnie inne scenariusze dostępu do danych. Rozkład *najnowsze* symuluje na przykład wiadomości pojawiające się na serwisie informacyjnym. Początkowo nowe wiadomości są często czytane, natomiast z czasem tracą na znaczeniu. Z kolei *zipfian* może być użyty do symulowania popularności stron internetowych. Jest ich bardzo dużo, jednak tylko niewiele z nich zyskuje ogromną popularność.

1.5.5. Architektura systemu

Twórcy systemu dostarczają klienta YCSB (ang. *YCSB Client*) do wykonania testu na bazie danych. Jest to program napisany w języku Java, którego zadaniem jest wygenerowanie obciążenia na bazie danych, a także wykonania pomiarów.



Rysunek 1.4: Architektura aplikacji klienckiej YCSB Client [Coo10]

Workload Executor to część aplikacji odpowiedzialna za zapełnienie bazy danych (faza ładowania) oraz przeprowadzenie części pomiarowej. Autorzy dostarczają podstawową implementację `CoreWorkload`. Użytkownik jednak może sam dostosować generowanie obciążenia do swoich potrzeb. Może to zostać wykonane na dwa sposoby. Pierwszym z nich jest dostarczenie pliku konfiguracyjnego, który będzie opisywać inne parametry obciążenia do wygenerowania. Drugim sposobem jest dostarczenie własnej implementacji modułu.

Obciążenie generowane przez Workload Executor wykonywane jest równolegle przez wiele wątków klienckich. Wątki te współdzielą obiekt typu `Workload`, dzięki czemu wszystkie wątki generują ruch o żądanym rozkładzie. Praca wątków może być ograniczona do żądanej intensywności, dzięki czemu możliwe jest przetestowanie opóźnienia na bazie podczas zmieniającego się obciążenia.

Podczas wykonywania operacji zbierane są statystyki odnośnie uzyskanej przepustowości, czy opóźnienia. Moduł zbierający statystyki dostarcza 99. i 95. percentyl opóźnienia, jak również histogram.

Jednym z głównych założeń twórców YCSB było stworzenie systemu do testowania, który byłby łatwo rozszerzalny na interfejsy kolejnych systemów NoSQL. Dlatego też YCSB tłumaczy ogólne wywołania `read()`, czy `scan()` na funkcje specyficzne dla konkretnych baz danych. Kod ten jest ładowany dynamicznie w momencie wykonywania testu. Aby dodać konkretną implementację bazy danych, należy dostarczyć implementację wszystkich metod wymaganych przez YCSB, tj. `read()`, `write()`, `scan()`, `update()`, `delete()`, które opisałem w podrozdziale 1.5.3.

Rozdział 2

Gemius BigTable

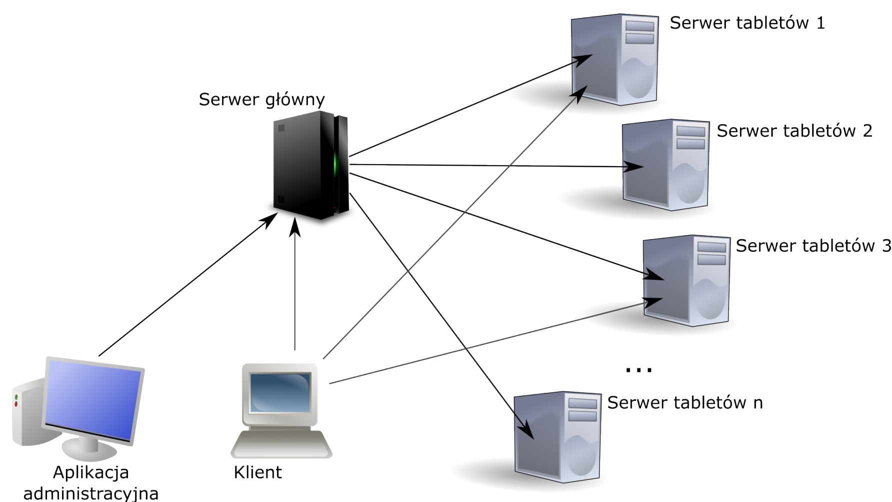
W rozdziale tym opiszę architekturę, kluczowe dla systemu biblioteki, wymagany system plików, model danych oraz narzędzia do testowania tytułowego systemu. Gemius BigTable (GBT) jest implementacją popularnego systemu BigTable (BT) stworzonego w firmie Google. Nie mniej jednak implementacja firmy Gemius różni się od pierwowzoru.

Motywacjami do stworzenia systemu były:

- W firmie Gemius szeroko wykorzystywany jest format RTL z biblioteki CommonLib (Więcej o CommonLib w podrozdziale 2.2). Oferuje on jednak pliki niemodyfikowalne. Gemius BigTable miał za zadanie obejść te ograniczenia.
- GBT miał budować abstrakcję bazodanową na dostęp do plików w formacie RTL.
- W wielu systemach w firmie powtarza się schemat zarządzania plikami RTL. BigTable miał zwalniać użytkownika z obowiązku zarządzania plikami.

2.1. Architektura

Gemius BigTable to rozproszony system bazodanowy. Serwery zarządzające danymi dzielą wspólną przestrzeń rozproszonego systemu plików.

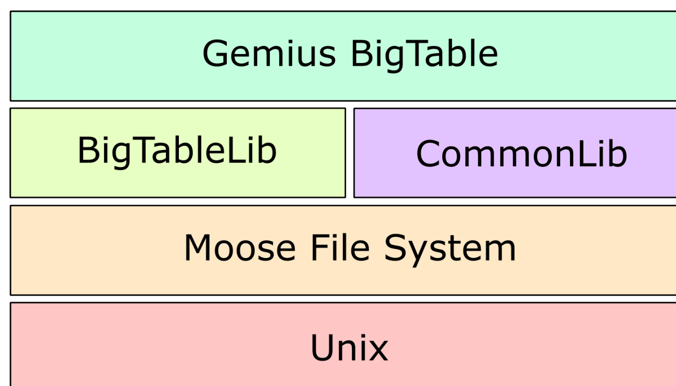


Rysunek 2.1: Architektura Gemius BigTable — komponenty systemu i schemat komunikacji

System składa się z następujących komponentów:

- *serwer główny* (ang. *master server*): serwer zarządzający systemem. Jest on odpowiedzialny za zarządzanie serwerami tabletów, równoważenie obciążenia w systemie, przydział tabletów (patrz sekcja 2.4) do serwerów danych, wykrywanie sytuacji awaryjnych oraz reagowanie na nie.
- *serwer tabletów* (ang. *tablet server*): serwer obsługujący dane. Jest on odpowiedzialny za realizowanie żądań klienta: wykonuje odczyty oraz transakcje.
- *BigTableLib*: biblioteka, która jest dołączana do programu klienckiego, aby komunikować się z systemem. Tłumaczy ona klienckie wywołania z API na żądania do BigTable.
- *aplikacja administracyjna*: program służący do administrowania systemem. Pozwala na zmiany parametrów tabel, dodawanie schematów tabel, wyłącza system oraz listuje istniejące obiekty w systemie.

Podczas uruchomienia systemu serwer główny wczytuje wszystkie metadane z dysku. Następnie podłączają się do niego kolejne serwery tabletów. Serwer główny rozdziela dane do obsługi pomiędzy serwery tabletów. Klient, aby wykonać operację na bazie danych, komunikuje się z serwerem głównym w celu uzyskania informacji o tym, który serwer tabletów odpowiada za jaką część danych. Następnie biblioteka kliencka kieruje żądania już bezpośrednio do odpowiednich serwerów danych.



Rysunek 2.2: Architektura Gemius BigTable — warstwy systemu

Gemius BigTable wykorzystuje dwie biblioteki: BigTableLib oraz CommonLib. BigTableLib implementuje API klienta, protokół komunikacyjny oraz typy danych wykorzystywane w systemie. Bibliotek CommonLib wymagana jest w systemie do fizycznego reprezentowania danych, serializacji i kompresji.

System składuje dane na rozproszonym systemie plików Moose File System (MooseFS, MFS). Każdy serwer tabletów powinien mieć zamontowaną instancję tego systemu plików. Dzięki temu każdy z nich ma dostęp do danych BigTable. MooseFS zapewnia także replikację danych. System pracuje na systemach z rodziny Unix, ponieważ MooseFS wymaga biblioteki FUSE. Więcej o Moose File System można przeczytać w podrozdziale 2.3.

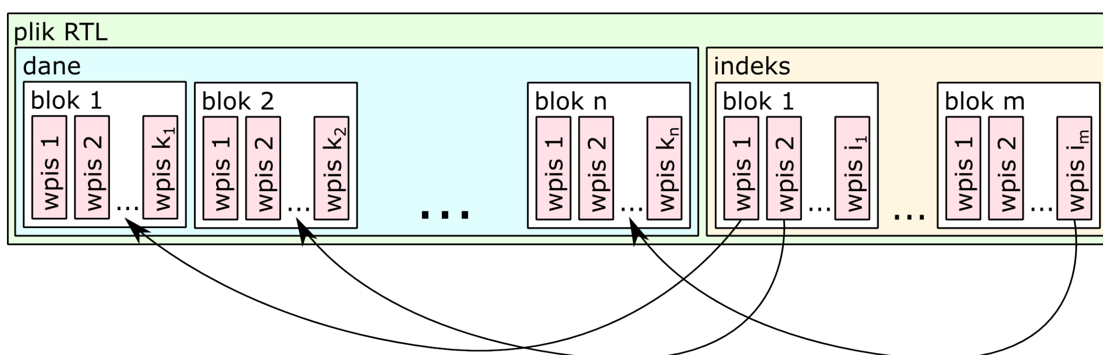
2.2. Biblioteka CommonLib

CommonLib to biblioteka rozwijana w firmie Gemius. Zawiera ona szereg ogólnych funkcji i klas. Z punktu widzenia projektu Gemius BigTable najistotniejsza, a zarazem najbardziej

rozbudowana, jest część RTL. Zawiera ona definicje szablonów reprezentujących między innymi:

- *strumienie* — obiekty tego typu reprezentują sekwencyjne przetwarzanie danych. Szablon `RecordStream` dostarcza operacji inkrementacji, dereferencji oraz zapytanie o koniec strumienia (`endOfStream()`). Jest on opakowaniem na abstrakcyjny szablon `RecordStreamImpl`, po którym należy odziedziczyć w celu dostarczenia konkretnych implementacji.
- *bazy danych* — w RTL baza danych jest rozumiana jako niemodyfikowalny plik zawierający dane. W takiej bazie przechowywana mogą być dane o typie prostym, kontenery STL oraz obiekty klas ze zdefiniowaną metodą szablonową `forEachAttribute(F &f)` do odczytu i zapisu danych. W klasie składowanego obiektu część pól musi zostać wyróżniona jako pola kluczowe. Dane w pliku składowane są w leksykograficznym porządku rosnącym.
- *multigrupy* — multigrupa w RTL jest tworem pozwalającym na jednoczesny odczyt danych z wielu plików w formacie RTL. Obiekt pobiera listę plików do odczytania, a następnie przekazuje wynikowy strumień zawierający posortowane dane z plików wejściowych.
- *indeksy* — indeksy dopisywane są na końcu pliku RTL w celu szybkiego rozpoczęcia odczytu pliku od zadanego klucza. Indeksy mają strukturę B-drzewa.

Na rysunku 2.3 przedstawione jest ułożenie danych w pliku w formacie RTL.



Rysunek 2.3: Struktura pliku RTL

Dane podzielone są na dwie części: część z właściwymi danymi oraz indeks. Zarówno rekordy danych, jak i rekordy indeksu podzielone są na bloki. Bloki są najmniejszą porcją danych, która jest wysyłana oraz czytana z systemu plików. Pojedyncze rekordy indeksu wskazują na początki bloków danych poprzez wskazanie ich przesunięć. W przypadku najefektywniejszego indeksu, B-drzewa, bloki danych tworzą węzły drzewa. Domyślnie wpisy w węźle drzewa grupowane są po 128 elementów.

2.3. Moose File System

Moose File System (MooseFS) to odporny na awarie, rozproszony system plików [Gad10]. System został opublikowany na licencji open-source w 2008 roku. Jego stworzenie było zainspirowane opublikowaniem przez firmę Google dokumentu opisującego Google File System ([Ghe03]).

Ogromną zaletą MooseFS jest prostota jego użytkowania. Systemu tego używa się praktycznie tak samo, jak każdego popularnego systemu plików z rodziny Unix. Moose File System oferuje:

- drzewiastą strukturę katalogów,
- przechowywanie atrybutów plików, zgodnie ze standardem POSIX,
- tworzenie odnośników twardych oraz symbolicznych,
- obsługę specjalnych urządzeń.

Dzięki temu skrypty shellowe zazwyczaj można przenieść bez modyfikacji na rozproszony system plików. MooseFS dostarcza użytkownikowi nowe możliwości:

- Dostępność — dane z systemu plików dostępne są wszędzie tam, gdzie zostanie zamontowana instancja systemu.
- Łatwa rozbudowa — w trakcie wykonywania operacji na systemie plików można go rozbudować o dodatkowe, dostępne miejsce. Wystarczy podłączyć do systemu kolejny dysk lub serwer składujący dane.
- Replikacja — MooseFS ma wbudowany mechanizm replikacji. Dla każdego pliku można ustawić parametr *goal*, który określa w ilu kopiach plik ma być utrzymywany w systemie. W przypadku awarii dysku, na którym składowana była jego replika, system sam dba o powielenie pliku na inny dysk, aby osiągnąć wymaganą liczbę replik.
- Odporność na błędy użytkownika — w systemie MooseFS usunięcie pliku nie skutkuje natychmiastowym usunięciem go z dysków, ale przeniesieniem go do kosza. Dla każdego pliku określony jest parametr *trashtimeout*, który określa ile sekund plik będzie przebywał w koszu. Po upływie tego czasu, informacje o pliku są usuwane. Jednak jeśli plik został usunięty w wyniku błędu użytkownika, to można go odzyskać z kosza przez trashtimeout sekund od usunięcia.

W systemie istnieją trzy typy procesów. Pierwszym, a zarazem najważniejszym, z nich jest *serwer główny* (ang. *master server*). Jest to proces odpowiedzialny za zarządzanie całym systemem plików. Przechowuje on w pamięci operacyjnej wszystkie metadane, jak na przykład opis plików, strukturę katalogów, czy atrybuty obiektów. Aby informacje te istniały w systemie w sposób trwały, serwer główny okresowo zapisuje do pliku obraz pamięci w postaci binarnej. Od momentu zapisu takiego pliku, serwer główny zapisuje także plik z logiem zmian. Dzięki temu, na podstawie tych dwóch plików, struktura systemu plików istnieje na nośniku danych w sposób trwały.

Kolejnym procesem obecnym w systemie jest *serwer kawałków* (ang. *chunk server*). Jest to proces uruchomiony na każdym z serwerów zawierających dyski do składowania danych w MooseFS. Procesy te są bezpośrednio odpowiedzialne za zapis danych w systemie plików oraz ich odczytywanie. Procesy te muszą także spełniać żądania serwera zarządzającego, na przykład w przypadku awarii innego serwera kawałków serwer główny może zlecić powielenie danych, które znajdowały się na uszkodzonym dysku. Wówczas to posiadacz ważnych replik jest odpowiedzialny za powielenie je na inne serwery kawałków.

Ostatnim komponentem systemu jest proces *mfsmount*. Procesy te działają na komputerach użytkownika i odpowiedzialne są za komunikację z serwerem głównym oraz serwerami

kawałków. Zostały stworzone z wykorzystaniem biblioteki FUSE do komunikacji z jądrem systemu operacyjnego. Dlatego też MooseFS może zostać zamontowany tam, gdzie dostępna jest biblioteka FUSE, czyli na systemach operacyjnych z rodziny UNIX.

Moose File System został stworzony w firmie Gemius, aby rozwiązać problem składowania i replikacji ogromnych plików. Mocno skompresowany plik zawierający dane gromadzone w firmie z jednej doby ma rozmiar przekraczający 120 GB. Z tego powodu MooseFS jest zoptymalizowany pod obsługę dużych plików. Aby zachować elastyczność w obsłudze tak dużych plików, są one dzielone na kawałki (ang. *chunk*). Rozmiar kawałka wynosi około 64 MB.

2.4. Logiczny model danych

BigTable oferuje bazę danych, w której rekordy przechowywane są w tabelach. Tabele te nie są relacjami dobrze znanymi z relacyjnych baz danych. Są one odwzorowaniem z unikatowego klucza w wartość. Pomysł jest podobny do opisanego w dokumencie firmy Google, jednak w Gemius BigTable sygnatura jest bardziej ogólna:

$$key \rightarrow value$$

Dane przechowywane w tabelach to ustrukturyzowane rekordy. Część z atrybutów kluczowych wyróżniona jest jako pola kluczowe. Pola te stanowią klucz identyfikujący rekord. Wartość stanowią pozostałe pola. Dane w tabeli ułożone są w rosnącym porządku leksykograficznym pól kluczowych. Schemat jest dość podobny do tego wykorzystywanego w bibliotece CommonLib opisanej w podrozdziale 2.2. Jest tak dlatego, że fizyczny model (podrozdział 2.6) tych danych jest silnie oparty na tej właśnie bibliotece.

W działającym systemie dodanie schematu tabeli odbywa się poprzez załadowanie przez system biblioteki dynamicznej z implementacją schematu. Biblioteka ta jest kompilowana i łączona z kodem C++ wygenerowanego przez CPPGenerator. Generator ten przyjmuje tabelę określoną przy użyciu języka do opisu tabel w systemie. Przykładowa definicja typu tabeli wygląda następująco:

```
1  CREATE_TABLE(
2      RecordExample
3  ,
4      hId     (uint64_t)          (Key)
5      lId     (std::string)       (Key)
6      visits  (int)
7      hits    (HitContainer)
8      startTs (std::list<uint32_t>)
9  ,
10     Printable
11     DefaultConstructible
12     Comparable
13 )
```

Kod definiuje tabelę o odwzorowaniu, którego sygnatura jest następująca:

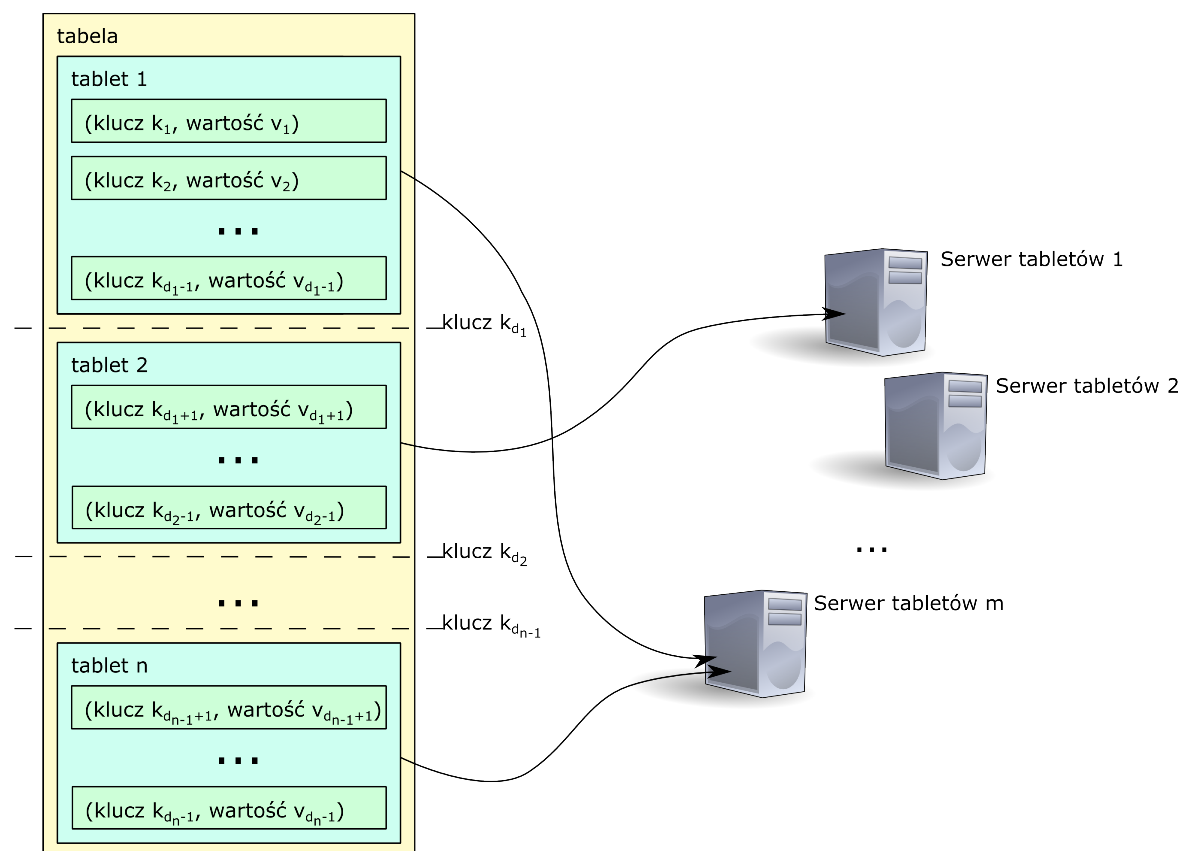
```
(uint64_t, std::string) -> (int, HitContainer, std::list<uint32_t>)
```

Tak zdefiniowany schemat tabeli wygeneruje kod dla klasy RecordExample (która reprezentuje cały rekord, czyli klucz wraz z wartością) oraz RecordExampleKey (która reprezentuje klucz rekordu).

Warto zauważyć, że atrybutami rekordu mogą być zarówno kontenery biblioteki STL, jak również obiekty dowolnych typów. W przypadku tych ostatnich, muszą one mieć zdefiniowane metody `forEachAttribute`, tak jak zostało to opisane w podrozdziale 2.2. Dane w powyższej tabeli będą uporządkowane rosnąco leksykograficznie względem klucza `(hId, lId)`.

Do definicji tabeli można dodać modyfikatory, które nadają odpowiednie własności definiowanemu typowi. Tak na przykład rekordy zawarte w tabeli o określonym powyżej typie będą posiadały zdefiniowany operator wypisania, konstruktor domyślny oraz operator porównania (po współrzędnych). Klucz musi mieć zdefiniowany operator porównywania, ponieważ dane w tabeli są posortowane rosnąco względem wartości klucza. Pełen, formalny opis języka do opisu tabel w Gemius BigTable wykracza poza ramy mojej pracy.

Dla każdej tabeli zbiór wartości atrybutów krotki będącej kluczem nazywamy *przestrzenią kluczy*. Przestrzeń ta jest dzielona na spójne fragmenty nazywane w systemie *tabletami* (por. rys. 2.4).



Rysunek 2.4: Schemat reprezentuje podział tabeli na tablety i przypisanie ich do serwerów tabletów

W momencie utworzenia tabeli, składa się ona tylko z jednego tabletu. Tablet ten obsługuje rekordy o kluczach z zakresu $(-\infty, +\infty)$. Serwer główny przydziela go jednemu z dostępnych serwerów tabletów. Serwer tabletów odpowiedzialny jest za realizację operacji na tym tablecie. W momencie, gdy rozmiar tabletu wzrośnie, serwer główny decyduje o jego podziale na dwie części pewnym kluczem k. Powstają dwa tablety: jeden obsługujący klucze

z przedziału $(-\infty, k)$, a drugi $[k, +\infty)$. Pierwszy z nich dalej jest obsługiwany przez przydzielony wcześniej serwer tabletów. Drugi natomiast zostaje przydzielony przez serwer główny do pewnego serwera tabletów. W sytuacji, gdy któryś z tabletów znów zbytnio się rozrośnie, serwer główny znów dokonuje podziału na tej samej zasadzie, co uprzednio.

2.5. API

Komunikacja użytkownika z bazą danych odbywa się poprzez API. Z punktu widzenia integracji z YCSB oraz testów przeprowadzonych z jego użyciem najistotniejsze są operacje CRUD, które opiszę w tym podrozdziale.

Operacje modyfikujące w systemie odbywają się w sposób transakcyjny. Użytkownik rozpoczyna transakcję, wykonuje modyfikacje, a następnie ją zatwierdza (`transaction.commit()`) lub odrzuca (`transaction.rollback()`).

Na szczególną uwagę zasługuje operacja `update()`, której semantyka jest odmienna od standardowo przyjmowanej. Wykonywanie `update()` w systemie powoduje leniwe wyliczanie zdefiniowanego z góry funktora `PairJoiner`.

```
1  struct PairJoiner {
2      void operator()(Record &lhs, const Record &rhs) const
3      {
4          // ...
5      }
6  };
```

Funktor ten jest wywoływany w sytuacji, gdy wykonuje się operację update na rekordzie, którego klucz już istnieje w tabeli. Wykonanie serii operacji na rekordach, które mają ten sam klucz `k`: `insert(r1)`, `update(r2)`, `update(r3)`, spowoduje, że odczytanie rekordu o kluczu `k`, będzie dawało rekord: `PairJoiner(r3, PairJoiner(r2, r1))`. Warto podkreślić, że operacją łącząca musi być operacją łączną (więcej w podrozdziale 2.6).

Przedstawię na krótkich przykładach jak wykonać operacje w Gemius BigTable na tabeli składającej rekordy typu `MyRecord` o dwóch polach numerycznych (`Key` oraz `Value`). Niech kluczem będzie rekord `MyRecordKey` o jednym polu numerycznym (`Key`). Przyjmijmy, że funktor aktualizujący będzie sumował wartości. Aby wykonywał sumowanie, zdefiniowany musi być on następująco:

```
1  struct PairJoiner {
2      void operator()(MyRecord &lhs, const MyRecord &rhs) const
3      {
4          lhs.refValue() += rhs.refValue();
5      }
6  };
```

Wówczas operacje wstawienia, aktualizacji, usunięcia, odczytu realizowane są w API następująco:

```
BigTable::Access btAccess("serverhost", hostPort);
BigTable::Transaction transaction =
    btAccess.beginTransaction(BigTable::TRANS_UNSORTED);
BigTable::Table<MyRecord> table =
    transaction.openTable<MyRecord>("tableName");
table.insert(MyRecord(1, 2));
transaction.commit();
```

Pokazany krótki fragment kodu realizuje dodanie rekordu do tabeli ”tableName”. Oczywiście operacja może się nie powieść, wówczas BigTableLib zgłosi odpowiedni wyjątek.

Wykonanie aktualizacji rekordu wymaga zastąpienia 6. wiersza w podanym przykładzie przez `table.update(MyRecord(1, 2));`. Spowoduje to zwiększenie aktualnej wartości rekordu o kluczu `1` o wartość 2. Z kolei, aby usunąć rekord o kluczu `1` z tabeli, należy zastąpić wiersz 6. przez `table.erase(MyRecordKey(1));`

Odczytanie rekordu o zadanym kluczu realizowane jest w systemie następująco:

```
BigTable::Access btAccess("serverhost", hostPort);
BigTable::Select<MyRecord> select =
        btAccess.createSelect<MyRecord>("tableName");
BigTable::QueryFilter<MyRecordKey> filter;
filter.addKey(MyRecordKey(32));
select.setFilter(filter);
RecordStream<MyRecord> rs = select.execute();
while (!endOfStream(rs)) {
    ++rs;
}
```

Aby wykonać odczyt przedziałowy (lewostronnie domknięty), należy zastąpić wiersz nr 5 następującym kodem:

```
filter.addRange(beginKey, endKey);
```

Aby zaś odczytać przedział obustronnie domknięty, należy użyć kodu:

```
filter.addRangeClosed(beginKey, endKey);
```

2.6. Fizyczny model danych

W Gemius BigTable tabele reprezentowane są przez utworzenie katalogu. Każdy z nich podzielony jest na podkatalogi reprezentujące tablety. W katalogach tych znajdują się pliki z danymi reprezentujące fragmenty pojedynczych transakcji, bądź też zbiory fragmentów transakcji. Cała opisana struktura katalogów znajduje się w MooseFS. Dzięki temu, serwer główny może dowolnie przydzielić tablety do serwerów danych, a te i tak będą miały dostęp do potrzebnych danych.

Jak napisałem w podrozdziale 2.5, klient modyfikuje dane w sposób transakcyjny. Jedna transakcja może obejmować modyfikację danych należących do różnych tabletów. W pliku zawartym w tablecie zapisywane są tylko modyfikacje dotyczące tej części tabeli. Dlatego

jedna transakcja reprezentowana jest fizycznie przez zbiór plików; zazwyczaj po jednym w każdym tablecie, którego dotyczyła transakcja. Jeśli natomiast podczas wykonania transakcji dojdzie do podziału tabletu, zapisywana jest ona jako kilka *plików częściowych*.

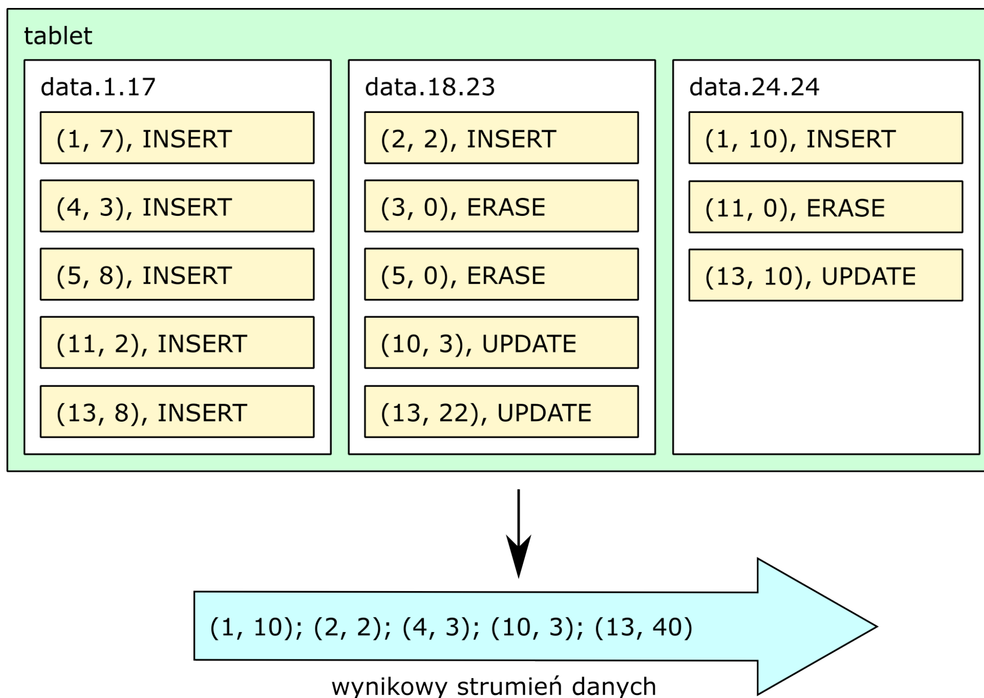
Wraz z rosnącą liczbą operacji wykonywanych na tablecie, liczba plików znajdujących się w jego katalogu znacząco by rosła. Dlatego okresowo serwer główny zleca serwerom tabletów wykonanie operacji przepisania zbioru plików do jednego, większego pliku. Przepisywane są pliki reprezentujące zbiór kolejnych transakcji. Po wykonaniu tego procesu pliki, które były przepisywane zostają zastąpione plikiem wynikowym i jeśli nikt z nich już nie korzysta, zostają usunięte z systemu. Warto zauważyć, że serwer główny może wybrać do złączenia dowolny zbiór kolejnych plików. W momencie łączenia aktualizowane są rekordy zgodnie z operatorem `PairJoiner`. Aby wyliczenie, bez względu na kolejność łączenia plików, zawsze dawało ten sam wynik, operator `PairJoiner` musi być operację łączną.

Pliki zawarte w katalogach tabletów są tworzone w formacie RTL zdefiniowanym w bibliotece CommonLib, którą opisałem w podrozdziale 2.2. Biblioteka CommonLib oferuje zapis plików w postaci niemodyfikowalnej. Z drugiej jednak strony w rozdziale 2.5 opisałem operacje aktualizacji (`update()`) oraz usunięcia (`erase()`) rekordu. BigTable łączy te dwie, z pozoru sprzeczne, koncepcje przy pomocy opakowania rekordów i dopisywania informacji o tym, jaka operacja została na nich wykonana. Rekordy w plikach RTL składowane są fizycznie w następującej postaci:

```
1  CREATE_CLASS(
2      RecordWrapper
3  ,
4      Record (RecordType)
5      Operation (int)
6  ,
7      TEMPLATE("typename RecordType")
8      DEFAULTCONSTRUCTIBLE
9      PRINTABLE
10     COMPARABLE
11     TEMPLATECONSTRUCTIBLE
12 )
```

Kod `CREATE_CLASS` jest makrem z biblioteki CommonLib generującym definicję klasy lub szablonu. `RecordWrapper` to szablon parametryzowany typem `RecordType`, dla którego definiuje opakowanie. Dla konkretnych implementacji schematów tabel, szablon `RecordWrapper` instancjonowany jest konkretnym typem.

Pole `Record` reprezentuje faktyczną zawartość rekordu. Pole `Operation` natomiast reprezentuje operację wykonaną na danym rekordzie, tj. `INSERT`, `UPDATE` lub `ERASE`. W momencie odczytu danych z tabeli otwierane są wszystkie pliki tabletu i odczytywane fragmenty z zadanego przedziału kluczy. Z danych z tych plików składany jest strumień wynikowy, wyliczany w momencie wykonania odczytu. Strumień ten nie zawiera usuniętych rekordów, a aktualizowane rekordy mają wartość wyliczoną za pomocą funktora `PairJoiner`. Sytuację tę zobrazowałem przykładem z rysunku 2.5. Przyjmijmy, że składowane tam są rekordy typu `MyRecord` zdefiniowanego w podrozdziale 2.5.



Rysunek 2.5: Odczyt danych w Gemius BigTable

W tablecie z rysunku 2.5 znajdują się trzy pliki: `data.1.17`, `data.18.23`, `data.24.24`. Pierwsze dwa z nich zawierają dane z wielu transakcji. Pierwszy z nich powstał w wyniku łączenia plików reprezentujących transakcje od 1. do 23.. Drugi natomiast z łączenia transakcji od 18. do 23.. W wyniku odczytu danych z tabletu, użytkownik otrzymał strumień z danymi zapisanymi w strzałce. Warto zauważyć, że:

- Rekord o kluczu `5` został dodany w pierwszym pliku. Nie został jednak przekazany do strumienia wynikowego, ponieważ w drugim pliku istnieje ze znacznikiem do usunięcia.
- Rekord o kluczu `3` został usunięty w drugim pliku, mimo, że nie istniał wcześniej w tabeli. Jest to poprawna sytuacja, a usunięcie to nie ma efektu.
- Dodanie rekordu (`INSERT`) o kluczu `1` w pliku trzecim, który już wcześniej istniał w tabeli, nadpisuje jego poprzednią wartość.
- Rekord z kluczem `13` ma wartość `40` ze względu na operator `PairJoiner` zdefiniowany w podrozdziale 2.5, który sumuje wartości rekordów.
- Rekord o kluczu `10` jest wyliczony poprawnie, ponieważ operator łączący w przypadku operacji `UPDATE`, gdy brakuje klucza o danej wartości, przekazuje do funktora rekord z domyślnymi wartościami atrybutów.

Jak przedstawiłem w tym podrozdziale, spora część obliczeń w systemie GBT wykonywana jest leniwie w momencie odczytu danych z kilku plików RTL. Dane zostają fizycznie zmaterializowane na dysku w momencie łączenia wszystkich plików. Moment, w którym dochodzi do złączenia, definiuje polityka łączenia. Aby zbadać czas dostępu do danych w zależności od strategii łączenia, stworzyłem symulator obciążenia tabletu, który opisałem w podrozdziale 3.3.

2.7. Odporność na awarie

Gemius BigTable jest systemem, w którym odporność na awarie w dużej mierze zapewnia warstwa systemu plików: Moose File System (więcej o tym będzie powiedziane w podrozdziale 2.3). BigTable nie musi sam dbać o utrzymywanie replik plików, ponieważ MooseFS udostępnia możliwość określenia w ilu kopiach (parametr *goal*) ma być trzymany dany plik.

W przypadku awarii serwera tabletów, serwer główny przydziela tablety uszkodzonego serwera pozostałym serwerom tabletów. Po przydzieleniu, serwer główny wznawia operacje wykonywane wcześniej na tablecie. Uzgadnia on uprzednio z serwerem tabletów stan w jakim znajduje się dany tablet. Awaria taka najczęściej jest niewidoczna dla użytkownika. Jednak, jeśli była wykonywana transakcja na tablecie obsługiwanym przez uszkodzony serwer, może zakończyć się ona niepowodzeniem.

Najsłabszym ogniwem systemu jest serwer główny, ponieważ jego działanie jest konieczne dla dostępności usługi. Serwer główny nie posiada swojej repliki w systemie, więc stanowi pojedynczy punkt awarii (ang. *single point of failure*).

Szczegółowy opis odporności systemu GBT na awarie można znaleźć w pracy magisterskiej Tomasza Wekseja [Wek10].

Rozdział 3

Narzędzia testujące

Rozdział ten podzielony jest na dwa podrozdziały. W pierwszym z nich opisuję proces integracji systemu Gemius BigTable z narzędziem YCSB opisanym w podrozdziale 1.5. W drugiej części natomiast zawarłem opis symulatora obciążenia.

3.1. Wymagania

Celem projektu magisterskiego jest w pierwszej kolejności przygotowanie narzędzi testowych. Wybrane bądź wytworzone narzędzia powinny jak najlepiej spełniać kryteria, które Karl Huppler w pracy [Hup09] podaje jako niezbędne warunki do stworzenia dobrego testu wydajności. Mianowicie test powinien być:

- **znaczący**: parametry, które zbadam za pomocą stworzonych narzędzi bezpośrednio określają efektywną pracę z systemem, jak również badają jego zdolność do skalowania się;
- **powtarzalny**: na załączonej do pracy płycie zamieściłem wszystkie kody źródłowe oraz skrypty niezbędne do powtórzenia testów;
- **sprawiedliwy**: systemy przetestuję na tej samej architekturze, przy pomocy spójnego środowiska YCSB;
- **weryfikowalny**: osoby zaznajomione ze środowiskiem YCSB mogą zweryfikować, czy dostarczona przeze mnie implementacja jest prawidłowa;
- **ekonomiczny**: symulator obciążenia pozwoli na bardzo szybkie przeprowadzanie wstępnych ocen polityk łączących w Gemius BigTable.

3.2. Integracja z YCSB

Yahoo! Cloud Serving Benchmark to platforma do testowania wydajności rozproszonych systemów bazodanowych. W projekcie tym duży nacisk położono na rozszerzalność rozumianą jako możliwość integracji z różnymi bazami danych. W ramach mojej pracy magisterskiej dostosowałem projekt YCSB do współpracy z Gemius BigTable.

3.2.1. Interfejs rozszerzenia

Aby przeprowadzić testy z użyciem YCSB, należy dostarczyć implementację klasy dziedziczącej po abstrakcyjnej klasie DB:

```
1  public abstract class DB
2  {
3      public void setProperties(Properties p);
4      public Properties getProperties();
5      public void init() throws DBException;
6      public void cleanup() throws DBException;
7
8      public abstract int read(String table, String key, Set<String> fields, ↩
          HashMap<String,ByteIterator> result);
9      public abstract int scan(String table, String startkey, int recordcount, ↩
          Set<String> fields, Vector<HashMap<String,ByteIterator>> result);
10     public abstract int update(String table, String key, HashMap<String,↩
          ByteIterator> values);
11     public abstract int insert(String table, String key, HashMap<String,↩
          ByteIterator> values);
12     public abstract int delete(String table, String key);
13 }
```

Pierwsze dwie metody pozwalają na zarządzanie opcjami przekazanymi z pliku konfiguracyjnego oraz wiersza poleceń. Implementacja kolejnych dwóch metod jest opcjonalna. Pozwalają na dostarczenie kodu wykonującego specjalne akcje odpowiednio przed oraz po wykonaniu testu. Implementacja pozostałych metod jest obowiązkowa. Nazwy metod bezpośrednio opisują akcje, jakie należy wykonać. Komentarza wymagają natomiast sygnatury metod:

- Funkcje czytające przekazują odczytaną wartość przez parametr wywołania. Dzieje się tak dlatego, że wynikiem wywołania jest kod błędu (zero wtedy i tylko wtedy, gdy wywołanie zakończono sukcesem).

- Przekazywana oraz zwracane wartości atrybutów rekordu są typu `HashMap<String , ByteIterator>`, ponieważ jest to odwzorowanie z nazwy kolumny w jej wartość (ciąg znaków). W przypadku metody `scan`, jako że czytany jest ciąg rekordów, jest to wektor takich odwzorowań.

- Metody czytające mają parametr `fields` o typie `Set<String>`. Umożliwia to odczytywanie wartości tylko niektórych z atrybutów.

- W systemach NoSQL czytając przedział danych, zwykle podaje się początkowy oraz końcowy rekord. Z kolei metoda `scan` pobiera początkowy rekord oraz liczbę rekordów do odczytania. Jest tak dlatego, że chcąc odczytać pewną liczbę rekordów trudno jest podać klucz ostatniego z nich.

3.2.2. JNI

BigTable udostępnia interfejs programisty w dużej mierze zgodny z wymaganiami YCSB. Głównym utrudnieniem jest to, że YCSB jest napisany w języku Java, natomiast jedyny interfejs Gemius BigTable jest napisany w języku C++. Dlatego komunikacja między GBT a YCSB odbywa się przy pomocy technologii JNI (Java Native Interface).

JNI pozwala na wywoływanie z programu napisanego w języku Java kodu napisanego w innym języku programowania. Użyłem go do wygenerowania szkieletu funkcji w języku C++, które będą korzystać z interfejsu Gemius BigTable. W tym celu musiałem zdefiniować klasę w języku Java, z której JNI wygenerował funkcje C++.

```
1  class GemiusBTImpl
2  {
3    public native void init(String hostname, int port, boolean ↩
         truncateTable);
4    public native void cleanup();
5
6    public native int read(String key);
7    public native int scan(String key, int recordcount);
8    public native int update(String key, String[] values);
9    public native int insert(String key, String[] values);
10   public native int delete(String key);
11
12   static
13   {
14     System.loadLibrary("gemiusbt");
15   }
16 }
```

W podanej deklaracji klasy istotne są dwa aspekty. Pierwszy z nich to sygnowanie metod słowem kluczowym `native`. Dla każdej metody oznaczonej tym słowem kluczowym JNI wygeneruje szkielet funkcji w języku natywnym.

Wygenerowany w ten sposób plik nagłówkowy na początku załącza definicje typów z pliku `jni.h`, a następnie zawiera funkcje reprezentujące poszczególne metody klasy `GemiusBTImpl`. Na przykład nagłówek metody `public native int scan(String key, int recordcount);` zostanie wygenerowany następująco:

```
1  /*
2   * Class:     com_yahoo_ycsb_db_GemiusBTImpl
3   * Method:    scan
4   * Signature: (Ljava/lang/String;I)I
5   */
6  JNIEXPORT jint JNICALL Java_com_yahoo_ycsb_db_GemiusBTImpl_scan
7    (JNIEnv *, jobject, jstring, jint);
```

Druga rzecz, na którą warto zwrócić uwagę, to zawartość wiersza nr 14 we fragmencie kodu deklarującego klasę `GemiusBTImpl`. Powoduje on załadowanie biblioteki dynamicznej `libgemiusbt.so`. Zawiera ona definicje wygenerowanych funkcji C++ oraz wymagane przez nie symbole. Jest to niezbędne, aby móc w kodzie napisanym w języku Java stworzyć i używać obiektów typu `GemiusBTImpl`. Definicja metod klasy `GemiusBTImpl` jest określona przy pomocy kodu napisanego w języku C++.

3.2.3. Implementacja interfejsu Gemius BigTable

Klasa języka Java, która implementuje interfejs GBT to `GemiusBTClient`. Główne zadania jakie wykonuje opisałem w kolejnych punktach.

1. Stworzenie instancji obiektu klasy `GemiusBTImpl` oraz zainicjowanie go parametrami przekazanymi z wiersza poleceń.

2. Przekonwertowanie formatu argumentów wywołań na typy danych łatwo obsługiwane przy pomocy JNI. Wszystkie argumenty wywołań tłumaczone są z typu `HashMap<String,ByteIterator>` na wbudowany w JNI typ `String[]`.

3. Przekazanie argumentów wywołania do metod obiektu klasy `GemiusBTImpl`.

4. Zaraportowanie kodu błędu wykonania metody.

Faktyczne wykonanie zadań żądanych przez YCSB zaimplementowane jest w pliku `com_yahoo_ycsb_db_GemiusBTImpl.cpp`. Zawiera on kod w języku C++ metod klasy `GemiusBTImpl`. W pliku tym zawarte są funkcje implementujące wszystkie natywne metody klasy `GemiusBTImpl`. Sygnatury funkcji są analogiczne do tej przedstawionej w poprzednim podrozdziale.

W implementacji utrzymywane są połączenia do bazy GBT w dwóch kontenerach typu `ConnectionsContainer`. Jeden zbiór połączeń przydzielony jest do wykonywania operacji modyfikujących dane, drugi natomiast do operacji odczytu. Podczas wywołania funkcji inicjującej, kontenery te wypełniane są obiektami utrzymującymi połączenie z bazą. Liczba połączeń w każdym z kontenerów równa się liczbie wątków zadeklarowanych podczas wywoływania programu. Wątek, który ma za zadanie wykonać operację na tabeli, pobiera wskaźnik na obiekt klasy `BigTableConnection` z odpowiedniej puli i zwraca go po wykonaniu zadania. Synchronizację między wątkami zapewniłem wykorzystując bibliotekę Boost Thread. Synchronizacja odbywa się na poziomie kontenerów z połączeniami.

Gemius BigTable realizuje zapis do tabel w postaci transakcji. Dostarczona implementacja grupuje zapisywane rekordy w transakcje po 32768 rekordów. GBT nie oferuje możliwości odczytów danych z niezatwierdzonych jeszcze transakcji, więc odczytywane dane będą pochodziły ze stanu po ostatniej udanej transakcji. W chwili niszczenia obiektu `BigTableConnection` wszystkie niezakończone transakcje zostają zatwierdzane.

Szczegóły implementacyjne zawarte są w kodach źródłowych na załączonej do pracy płycie.

3.3. Symulator obciążenia tabletu

Dane w tablecie składowane są w postaci plików w formacie RTL. Plik taki początkowo reprezentuje część pojedynczej transakcji obsługiwanej przez dany tablet (więcej o fizycznym modelu danych: podrozdział 2.6). Wraz z kolejnymi transakcjami liczba plików w tablecie nieustannie by rosła. Dlatego w Gemius BigTable działają w tle procesy łączące pliki RTL. Jest to istotne z kilku względów, mianowicie:

- **Odczyt danych**: W momencie odczytu danych z tabletu, serwer tabletów wyszukuje odpowiednie dane we wszystkich plikach, które się w nim znajdują. Operacje na systemie plików są kosztowne czasowo, dlatego im mniej plików w tablecie, tym szybszy dostęp do danych.

- **Fizyczne wyliczanie rekordów**: Operacje modyfikujące rekordy zapisywane są ze znacznikami `UPDATE`, czy `ERASE`, a odczytanie rzeczywistej wartości rekordu wymaga dodatkowej pracy: złożenia wpisów ze znacznikami. Aby efektywniej odczytać rekord, dane muszą być przepisane do postaci z rzeczywistą wartością. Może się to odbyć tylko w momencie łączenia wszystkich plików.

- **MooseFS**: System plików składujący dane zaprojektowany był do obsługi dużych plików, a nie dużej liczby małych plików. Przechowywanie każdej transakcji w osobnym pliku skutkowałoby nadmiernym zwiększeniem metadanych w MooseFS.

System jest konfigurowalny pod względem strategii wykonywania łączenia plików. Zadaniem takiej polityki jest zadecydowanie kiedy rozpocząć łączenie oraz które pliki złączyć. Przedwczesne rozpoczęcie łączenia powoduje nadmierne obciążenie serwera, jednak zbyt późna akcja powoduje powstanie w tablecie dużej liczby plików. Złączenie mniejszej liczby plików pozwala na szybsze zakończenie operacji. Z kolei złączenie wszystkich plików pozwala na zapisanie faktycznych wartości rekordów. Odpowiedni dobór zadań złączenia jest bardzo istotny, ponieważ serwery tabletów dysponują ograniczoną liczbą wątków łączących.

3.3.1. Interfejs symulatora

Ustalenie optymalnej polityki łączenia jest zadaniem nietrywialnym, a testowanie kolejnych rozwiązań i ich specyficznych konfiguracji wymaga dużo czasu. To było motywacją do stworzenia prostego narzędzia symulującego zachowanie tabletu. Program symuluje transakcje wykonane na tablecie, okresowe odczyty oraz estymacje czasu ich wykonania. Przez cały czas pracy z tabletem, program symuluje operacje złączenia określane przez testowaną politykę łączenia.

```
1  $ ./workloadSimulator -h
2  Allowed options:
3    -n [ --recordsAmount ] arg (=1000000) total number of records to simulate an
4                                          insertion
5    -t [ --recordsPerTransaction ] arg (=10000)
6                                          number of records to insert in each
7                                          transaction
8    -s [ --recordsPerSecond ] arg (=8000) number of records that system process
9                                          during a second
10   -f [ --selectFrequency ] arg (=10)    interval (in seconds) between selects
11   -p [ --policy ] arg (=MergePolicySortedTowersN)
12                                         set merge policy to test. arg =
13                                         MergePolicySortedTowersN [NUM] [NUM] |
14                                         MergePolicyAllWhenMoreThanN [NUM] [NUM]
15   -h [ --help ]                         produce help message
```

Program symuluje zapisanie `recordsAmount` rekordów do tabeli. W każdej transakcji zapisywane jest po `recordsPerTransaction` rekordów. Program symuluje pracę systemu, który zapisuje po `recordsPerSecond` na sekundę. Odczyty wykonywane są co `selectFrequency` sekund. Parametr `policy` pozwala określić politykę jakiej symulator ma użyć przy decydowaniu o łączeniu plików (konkretne polityki łączenia opisane są w podrozdziale 4.2).

Podczas wykonywania symulacji program utrzymuje w pamięci operacyjnej *reprezentatów plików* (obiekty reprezentujące w symulatorze informacje o plikach), które reprezentują zawartość tabletu. Wraz z przetwarzaniem kolejnych transakcji program modyfikuje zbiór reprezentantów plików poprzez symulowanie operacji złączenia pewnego ich zbioru.

3.3.2. Polityki łączenia

W systemie Gemius BigTable polityką łączenia (polityki łączenia zdefiniowałem w podrozdziale 2.6) są obiekty dziedziczące po abstrakcyjnej klasie `MergePolicy` o następującym interfejsie:

```
class MergePolicy
{
public:
    typedef boost::shared_ptr<MergePolicy> MergePolicyType;
    typedef std::set<MergeUnit> MergeUnitsSet;

    virtual ~MergePolicy();
    virtual std::list<MergeUnitsSet> createMergeJobs(const MergeUnitsSet &mergeUnits↩
        , const size_t mergeJobsLimit) const = 0;
    virtual std::string toString() const = 0;

    MergePolicy(int maxNumberOfFiles = 100);
    unsigned int getMaxNumberOfFiles() const;
};
```

Metodą o kluczowym znaczeniu w załączonej implementacji klasy `MergePolicy` jest `createMergeJobs`. Pierwszy parametr tej metody to zbiór obiektów typu `MergeUnit`. Kontener ten zawiera pliki, które mogą zostać złączone (tj. te, na których nie jest wykonywana obecnie operacja łączenia). `MergeUnit` to metadane reprezentujące zbiór plików, które wchodzą w tablecie w skład pojedynczej transakcji. Zbiór ten stanowi albo pojedynczy plik zawierający komplet danych, albo kilka plików częściowych. Drugi parametr określa maksymalną liczbę zadań, jaką może przekazać w wyniku ta metoda. Parametr konstruktora klasy określa maksymalną liczbę plików, jakie mogą wejść w skład pojedynczej transakcji. Ograniczenie to ma na celu redukcję liczby otwartych deskryptorów.

Symulator dostarcza możliwości testowania dwóch polityk łączenia: *Merge all* oraz *Sorted towers*. Dla dalszych rozważań określę, co to oznacza, że plik jest *starszy*. Plik jest tym starszy, im wyższe są identyfikatory transakcji, z których pochodzą dane. W szczególności plik zawierający dane z pierwszej transakcji na tabeli jest *najmłodszy*.

Merge all

Polityka *Merge all* usiłuje łączyć wszystkie pliki w jeden. Jest ona parametryzowana dwoma liczbami. Pierwsza z nich n określa maksymalną liczbę plików, która może być w tablecie bez łączenia. Gdy liczba plików będzie większa od `n` serwer główny zleca proces łączenia. Drugi argument *maxFiles* określa maksymalną liczbę plików, która może być łączona. Zawsze łączonych jest $min(n, maxFiles)$ plików. W przypadku, gdy plików jest więcej niż *maxFiles* do łączenia wybierane jest *maxFiles* najstarszych plików. Zaletą takiego podejścia jest to, że w wyniku łączenia wszystkich plików zawsze wyliczana jest ostateczna wartość rekordu, a usunięte rekordy zwalniają miejsce.

Sorted towers

Drugą z badanych polityk łączenia jest *Sorted towers*. Jak sugeruje nazwa, zachowanie polityki ma prowadzić do sytuacji, gdzie rozmiary plików stanowią posortowany malejąco ciąg. Podobnie jak poprzednia polityka *Merge all*, ta również jest parametryzowana dwoma parametrami: n i *maxFiles*. Znaczenie drugiego z nich jest identyczne jak poprzednio. Parametr n to liczba określająca jaką największą liczbę plików będzie usiłowała osiągnąć polityka w wyniku łączenia. Gdy liczba plików przekroczy n, polityka podejmuje akcję łączenia plików. Załóżmy, że w tablecie jest k plików oraz $k > n$. W pierwszej kolejności do zbioru łączonych plików dodawanych jest $k - n + 1$ najstarszych plików. Szacowany jest rozmiar wynikowego pliku. Jeśli oszacowany rozmiar jest większy od rozmiaru najstarszego pliku, który nie jest

w zbiorze plików do złączenia, to plik ten jest dodawany do zbioru. Procedura szacowania rozmiaru wynikowego pliku oraz ewentualnego rozszerzenia zbioru plików do złączenia jest powtarzana tak długo, aż zostanie osiągnięta pożądana własność, tj. rozmiary kolejnych plików (od najmłodszego) będą nierosnące. Niewątpliwą zaletą tej polityki jest to, że usiłuje ona łączyć pliki o zbliżonych rozmiarach.

3.3.3. Symulacja operacji bazodanowych

Program modeluje obciążenie na bazie danych poprzez symulowanie operacji wykonywanych w systemie. Symulowane są trzy typy operacji:

- **transakcja**: transakcje symulowane są nieustannie. Czas ich trwania wyliczany jest z wartości `recordsPerSecond` oraz `recordsPerTransaction`.
- **łączenie**: po każdej operacji transakcji oraz łączenia wywoływany jest kod (z Gemius BigTable) testowanej polityki, który określa, czy rozpocząć symulację kolejnej operacji złączenia.
- **odczyt**: co `selectFrequency` sekund symulowany jest odczyt danych i szacowany koszt wykonania odczytu.

W symulatorze istnieje jedna instancja obiektu `SimulationEnvironment`. W obiekcie tym składowane są informacje o ilości przetworzonych rekordów, parametrach wywołania oraz czasie, który zasymulowano od początku testu. Środowisko jest wykorzystywane i modyfikowane w głównej pętli programu przedstawionej w załączonym kodzie.

```
1     EventsContainer eventsToProcess;
2
3     EventVisitorExecutor eventsExecutor(environment);
4     while (environment.recordsInserted < environment.recordsToInsert) {
5         updateMerge(environment, eventsToProcess, mergePolicy);
6         updateSelect(environment, eventsToProcess);
7         updateTransaction(environment, eventsToProcess);
8
9         assert(!eventsToProcess.empty());
10
11        Event *event = eventsToProcess.begin()->second;
12        environment.simulationTime = eventsToProcess.begin()->first;
13        event->accept(eventsExecutor);
14
15        delete event;
16        eventsToProcess.erase(eventsToProcess.begin());
17    }
```

W obiekcie `eventsToProcess` przetrzymywane są informacje o aktualnie symulowanych operacjach. Operacje te opisywane są przez obiekty klas dziedziczących po klasie `Event`. Wydarzenia przechowywane są tam w porządku rosnącym względem czasu zakończenia operacji. Natomiast `eventVisitorExecutor` to obiekt implementujący wzorzec odwiedzający (ang. *visitor*), który odpowiedzialny jest za wykonania właściwego dla przetwarzanej operacji kodu.

Pętla główna wykonywana jest tak długo, aż nie zostanie zasymulowane zapisanie żądanej liczby rekordów. W każdym obiegu pętli podejmowane są decyzje, czy zainicjować kolejne operacje. Podczas każdej iteracji przetwarzane jest również zakończenie pojedynczej operacji. Wybierana jest ta z wykonywanych operacji, której czas zakończenia jest najmniejszy. Aktualizowany jest również symulowany czas.

3.3.4. Ewaluacja kosztu odczytu danych

Wynikiem działania programu jest ciąg *kosztów* związanych z odczytami kolejnych rekordów. Koszt określany jest na podstawie szacowania liczby bloków jaka byłaby konieczna do odczytania z plików tabletu. Symulator szacuje liczbę bloków jaką trzeba odczytać z pliku RTL w celu uzyskania dostępu do dowolnego rekordu przy pomocy następującej prostej funkcji `blockAccessedInFileOfSize`.

```
1  uint64_t blockAccessedInFileOfSize(uint64_t fileSize,
2          uint64_t recordsPerBlock, uint64_t entriesPerNode) {
3      uint64_t dataBlocksAmount = fileSize / recordsPerBlock;
4      uint64_t indexRecords = dataBlocksAmount;
5
6      // calculate tree height
7      uint64_t treeHeight = 1;
8      uint64_t indexRecordsIter = indexRecords;
9      while (indexRecordsIter > entriesPerNode) {
10         treeHeight++;
11         indexRecordsIter /= entriesPerNode;
12     }
13
14     return treeHeight + 1;
15 }
```

Funkcja jako parametry pobiera kolejno: rozmiar plików (mierzony w liczbie rekordów), liczbę rekordów w bloku danych oraz liczbę wpisów w wierzchołku B-drzewa stanowiącego indeks. Sumaryczna liczba bloków do odczytania to liczba bloków indeksu oraz jeden blok danych. Dlatego funkcja najpierw wylicza wysokość B-drzewa a następnie przekazuje wynik powiększony o 1.

Wyniki testów przeprowadzone z opisanym symulatorem można znaleźć w podrozdziale 4.2.

Rozdział 4

Wyniki testów

W pierwszej części rozdziału opiszę przeprowadzone przy pomocy YCSB testy wydajności systemów Gemius BigTable oraz Cassandra. Opiszę środowisko testowe, schemat testowania oraz scenariusze testowe. Dla każdego z przeprowadzonych testów zaprezentuję oraz przeanalizuję wyniki badanych systemów.

W drugiej części rozdziału opiszę przeprowadzone testy z wykorzystaniem symulatora obciążenia tabletu, a następnie powtórzę ten sam scenariusz w rzeczywistym środowisku. Opiszę polityki łączenia plików, które użyję w testach. Następnie przedstawię oraz przeanalizuję wyniki testów.

4.1. Testy z wykorzystaniem YCSB

W sekcji tej opiszę wyniki testów, które przeprowadziłem przy pomocy środowiska YCSB (szczegółowy opis zawarłem w sekcji 1.5). Aby dostarczyć Czytelnikowi punkt odniesienia do wyników testów GBT, postanowiłem przeprowadzić identyczne testy na jeszcze jednym, popularnym systemie NoSQL na licencji open source: Cassandra.

4.1.1. Środowisko testowe

Testy w środowisku rozproszonym mogłem przeprowadzić dzięki uprzejmości firmy Gemius. Do testów użyłem czterech serwerów. Nazwijmy te serwery od *server1* do *server4*. *server4* został użyty tylko do testów skalowalności systemu. Każdy z nich wyposażony był w czterordzeniowy procesor Intel(R) Xeon(R) CPU X3220 @2.40 GHz oraz 4 GB pamięci operacyjnej. Na każdym z nich zainstalowany był system operacyjny Linux Debian 5.0. Na serwerach *server1*, *server3*, *server4* system używał jądra w wersji `2.6.32-bpo.5-686-bigmem`, natomiast *server2* w wersji `2.6.26-1-686-bigmem`.

Serwery użyte do testów nie były obciążone. Wchodzą one jednak w skład jednej z instancji systemu plików MooseFS, więc uruchomiony był na nich procesy `mfsmount` oraz `mfschunkserver`. Procesy te jednak niemal przez cały czas trwania testu były nieaktywne.

Konfiguracje zarówno Gemius BigTable, jak i Cassandra nie były optymalizowane pod kątem przeprowadzonych testów. Użyłem w obu systemach konfiguracji domyślnych. Wprowadziłem jednak następujące zmiany:

- **replikacja**: W obu systemach replikacja została ustawiona na poziomie 2. Dzięki temu zbadałem jak zachowują się systemy z zachowaniem bardzo ważnej cechy systemów NoSQL: redundancji danych.

- **ograniczenie pamięci**: Ponieważ użyte systemy stanowiły część rozproszonego systemu plików, więc nie mogłem wykorzystać całej dostępnej pamięci operacyjnej. Obydwa systemy zostały uruchomione z ograniczeniem pamięci do 2 GB. Aby wymusić to ograniczenie na systemie Cassandra, ustawiłem parametr `MAX_HEAP_SIZE` na 1 GB.
- **opóźnienie**: Obydwa systemy zostały skonfigurowane pod kątem mniejszego opóźnienia, tj. synchronizują dane na dysk okresowo.

W testach tych użyłem poszczególnych systemów w następujących wersjach:

- YCSB: 0.1.4,
- Gemius BigTable: najnowsza rozwojowa (łatka: `52c8682`),
- Gemius BigTableLib: najnowsza rozwojowa (łatka: `0af71a1`),
- Gemius CommonLib: najnowsza rozwojowa (łatka: `14151d0`),
- MooseFS: 1.6.26,
- Cassandra: 0.7.0.

Testy na systemach wykonywane byłe naprzemiennie. Jednocześnie testowany i uruchomiony był tylko jeden z systemów.

We wszystkich testach z wykorzystaniem YCSB klient wykonywał operacje na czterech wątkach. W testach obu systemów proces kliencki nie posiadał osobnego serwera, a uruchomiony był na jednym z serwerów składujących dane. Proces klienta nie był wąskim gardłem systemu i generował małe obciążenie dla systemu. W przypadku systemu Gemius BigTable, proces kliencki oraz serwer główny uruchomione były na osobnych serwerach.

4.1.2. Ogólny opis

Podczas wykonywania testów starałem się badać podobne parametry jak te opisane w artykule autorów YCSB [Coo10]. Dzięki temu będę mógł odnieść swoje wyniki testów do tych ze wspomnianego artykułu.

Do testowania użyłem dostarczonej z YCSB definicji rekordu: klucz oraz dziesięć wartości typu string. Każde z pól podczas testów będzie napisem długości 100 znaków.

4.1.3. Testowanie opóźnienia

W części tej zaprezentuję wyniki testów opóźnienia. YCSB pozwala na określenie jaka ma być osiągnięta przepustowość systemu. Następnie wykonuje w systemie szereg operacji (odczyt, zapis, aktualizacja, usunięcie, odczyt przedziałowy) i raportuje jaką faktycznie przepustowość udało się osiągnąć, a także jakie było średnie opóźnienie przy wykonywaniu poszczególnych operacji. Jako przepustowość rozumiana jest tutaj suma wszystkich operacji wykonanych w ciągu sekundy.

Z dokumentacji YCSB wnioskować można, że system ten automatycznie bada wpływ obciążenia na przepustowość. Niestety jednak użyta przeze mnie wersja nie zawierała wspomnianej funkcjonalności. Dlatego w celu automatycznego badania tej zależności, rozszerzyłem projekt YCSB o skrypt `latencyByThroughput.py`. Pobiera on jako argument wywołanie klienta YCSB wraz z wymaganymi parametrami, a wyniki zapisuje do plików (po jednym dla każdego typu operacji).

```
$ ./chartsScripts/latencyByThroughput.py --help
Usage: latencyByThroughput.py [options] command

Options:
  -h, --help            show this help message and exit
  -d DELTA, --delta=DELTA
                        delta of consecutives throughputs[default=350]
  -s SECONDS, --seconds=SECONDS
                        seconds of execution with each throughput[default=120]
  -l LABEL, --label=LABEL
                        label of the test[default=test]
```

Aby zbadać jak zmienia się opóźnienie w zależności od zwiększającego się opóźnienia, skrypt uruchamia przekazane polecenie dla różnych żądanych przepustowości. Przepustowości jakie mają zostać przetestowane określa parametr `delta`. Skrypt wykona pomiar dla kolejnych żądanych przepustowości: `delta`, `2*delta`, `3*delta`, ... Żądana przepustowość będzie zwiększana tak długo, jak długo system będzie w stanie pracować z zadaną przepustowością. Czas pracy na jednej przepustowości określa w sekundach parametr `seconds`. Parametr `label` określa od jakiej nazwy mają zaczynać się pliki z wynikami pomiaru.

Przed wykonaniem pomiarów opóźnień na systemach, uruchomiłem program kliencki YCSB w trybie zapełniania bazy danych. Do obydwu systemów załadowałem po 50 milionów rekordów.

Proces ten w przypadku Gemius BigTable trwał półtorej godziny. W wyniki załadowania danych MooseFS raportuje, że GBT posiada dane o rozmiarze 47 GB. Jako, że replikacja jest ustawiona na poziomie 2, to fizycznie dane na dyskach zajmują 94 GB. Podczas procesu zapełniania w systemie powstały 4 tablety. Ich zapełnienie było równomierne po około 12 GB (nie wliczając replikacji).

Zapełnianie bazy danych Cassandra trwało około dwie i pół godziny. W przypadku Cassandry, na serwerach powstały katalogi z danymi o pojemnościach: 54 GB, 27 GB, 47 GB.

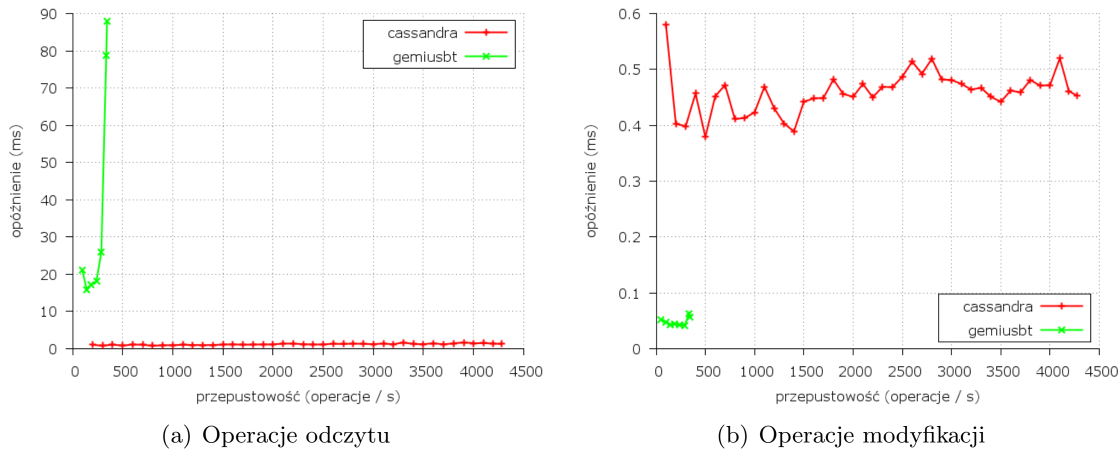
W wyniku działania niektórych testów, do zapełnionej już bazy danych, dodawane są kolejne rekordy. Ze względu na czas zapełniania bazy danych oraz fakt, że modyfikacje wynikające z testów są kilka rzędów wielkości mniejsze w stosunku do zapełnienia bazy, pomiędzy kolejnymi scenariuszami testowymi nie czyściłem baz danych. Niemniej jednak testy na systemach wykonywane były w tej samej kolejności, a także były to jedyne modyfikacje tych baz.

Intensywna aktualizacja i odczyt

Pierwszy z przeprowadzonych testów polega na intensywnym czytaniu pojedynczych rekordów (50% operacji) oraz modyfikowaniu danych (pozostałe 50% operacji). Jest to test o nazwie `workloada` dostarczony wraz z YCSB. Dlatego też, w odróżnieniu od pozostałych testów, zastosowany rozkład jest innych niż *jednostajny* stosowany domyślnie w pozostałych testach. Formalny opis testu jest następujący:

```
readallfields=true
readproportion=0.5
updateproportion=0.5
scanproportion=0
insertproportion=0
requestdistribution=zipfian
```

Podczas operacji odczytu zawsze odczytywane będą wszystkie pola. Operacje na rekordach wykonywane będą z rozkładem *zipfian* (rozkłady i ich znaczenie opisałem w podrozdziale 1.5.3). parametr `operationscount` pomijam, ponieważ skrypt `latencyByThroughput` dobiera liczbę operacji dynamicznie w czasie testu.



(a) Operacje odczytu

(b) Operacje modyfikacji

Rysunek 4.1: Opóźnienie operacji w zależności od przepustowości, scenariusz `workloada`

Wykresy przedstawione na rysunku 4.1 prezentują opóźnienie w zależności od uzyskanej przepustowości. Oś pozioma reprezentuje testowaną liczbę operacji wykonanych w ciągu sekundy. Oś pionowa to średnie opóźnienie (czas wykonania pojedynczej operacji) mierzone w milisekundach. Punkty pomiaru występują na wykresie tak długo, jak długo system jest w stanie zagwarantować żądaną przepustowość. Na im szerszym przedziale wykres jest określony, tym lepiej, ponieważ oznacza to, że system jest w stanie wytrzymać większe obciążenie. Wartości natomiast im mniejsze tym lepsze, ponieważ oznacza to, że użytkownik musi krócej oczekiwać na zrealizowanie żądania.

Cassandra w tym teście była w stanie zrealizować do 4200 operacji na sekundę. Podczas wykonywania operacji odczytu opóźnienie wzrastało od 1 ms przy małym obciążeniu do około 1.5 ms przy granicznym obciążeniu. Aktualizacja rekordów trwała od 0.4 ms do 0.6 ms podczas całego trwania testu.

Gemius BigTable był w stanie obsłużyć istotnie mniejsze obciążenie kończąc test na obciążeniu 350 operacji na sekundę. Istotnie dłużej trwały też operacje odczytu. Przy obciążeniu do 300 operacji na sekundę opóźnienie utrzymywało się na poziomie od 15 do 20 ms. Później natomiast drastycznie wzrosło osiągając pod koniec testu wartość 88 ms. Zdecydowanie szybsze natomiast okazała się aktualizacja danych oscylując podczas całego testu na poziomie od 40 do 60 mikrosekund.

W wyniku powtarzania testu otrzymywany jest niemal identyczny wynik. Bardzo duże opóźnienie na operacjach odczytu pojedynczych rekordów jest znanym problemem w systemie Gemius BigTable. Podczas odczytu GBT otwiera wszystkie pliki znajdujące się w tablecie obsługującym dany rekord. W każdym z tych plików odszukiwany jest rekord przy pomocy B-drzewa (więcej o fizycznym modelu danych można przeczytać w podrozdziale 2.6). Wiąże się to z odczytem kilku bloków danych. Sytuację można poprawić implementując następujące optymalizacje związane z dostępem do danych:

- pamięć podręczna (ang. *cache*): zapamiętywanie bloków indeksu (szczególnie bloków korzenia) istotnie zmniejszyłoby liczbę odwołań do systemu plików;

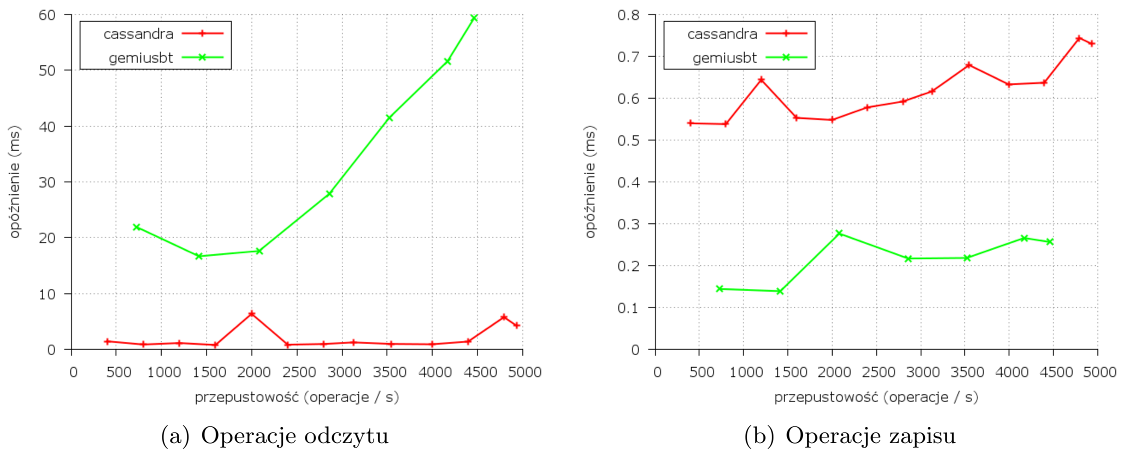
- filtry Blooma: wówczas odczytanie rekordu wymagałoby operacji tylko na pewnym podzbiorze plików w tablecie;
- asynchroniczny odczyt: biblioteka CommonLib podczas tworzenia strumienia danych odwołuje się do wymaganych plików w sposób synchroniczny. Asynchroniczne podejście pozwoliłoby zrównoleglić odszukiwanie rekordów w plikach.

Intensywny zapis

Podczas tego testu wykonywane były operacje odczytu oraz zapisu do bazy danych. Stosunek operacji wynosił odpowiednio 99% i 1%. Jest to test dostarczony przeze mnie, a jego nazwa w źródłach YCSB to `heavyInserts`. Konfiguracja testu jest następująca:

```
1 readallfields=true
2 readproportion=0.01
3 updateproportion=0
4 scanproportion=0
5 insertproportion=0.99
6 requestdistribution=uniform
```

W odróżnieniu od poprzedniego testu zastosowany rozkład jest jednostajny.



(a) Operacje odczytu (b) Operacje zapisu

Rysunek 4.2: Opóźnienie operacji w zależności od przepustowości, scenariusz `heavyInserts`

W teście tym system GBT osiągnął istotnie wyższą maksymalną przepustowość osiągając wynik 4462 operacji na sekundę. Jest to nieco mniej od systemu Cassandra, który potrafił wykonać 4930 operacji na sekundę. Także i w tym teście opóźnienie Cassandry na wykonywaniu operacji odczytu oscylowało w okolicach 1 ms. Tym razem jednak przy granicznych przepustowościach opóźnienie odczytu wzrosło do wartości od 4 do 6 milisekund. Zaskakujący jest gwałtowny wzrost opóźnienia do 6 ms przy obciążeniu 2000 operacji na sekundę. Także i w tym teście opóźnienie odczytu systemu GBT było istotnie wyższe. Podczas całego testu wynosiło od 20 do 60 milisekund. Wartość opóźnienia odczytu dla GBT wraz ze wzrostem przepustowości ma tendencję rosnącą.

Gemius BigTable podczas testu osiągnął opóźnienia zapisu rekordów w przedziale od 0.13 do 0.27 milisekundy. Znacznie dłużej trwały zapisy do systemu Cassandra, które trwały od

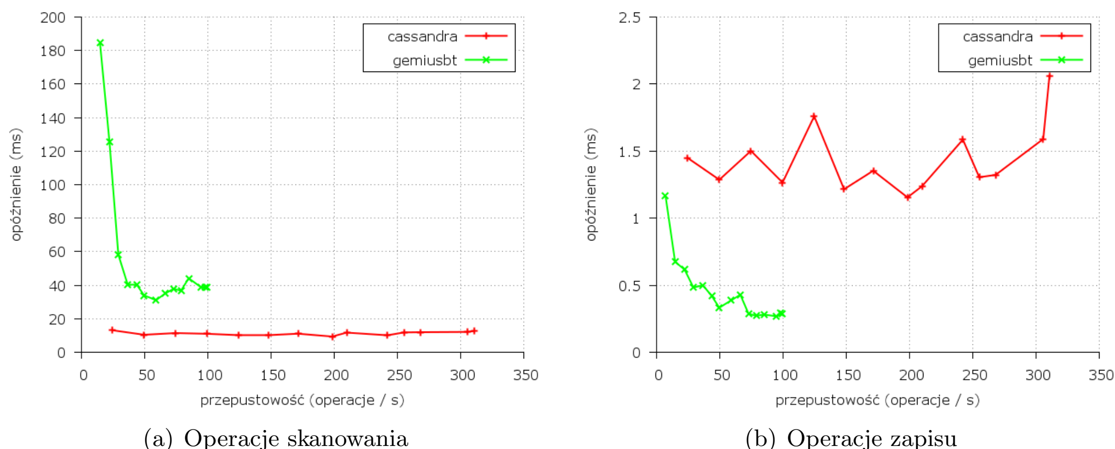
0.53 do 0.74 ms. Opóźnienia zapisu do obydwu systemów wykazują tendencję wzrostową w stosunku do tego jak bardzo obciążony jest system.

Krótkie przedziały

Ten test ma na celu przetestować przedziałowe odczyty z bazy danych. Test ten jest dostarczony przez autorów YCSB i nosi nazwę **workloade**. Odczytywane przedziały są krótkiej długości (do 1000 rekordów). 95% operacji stanowią odczyty przedziałowe, natomiast pozostałe 5% to zapisywanie rekordów. Plik konfiguracyjny testu wygląda następująco:

```
1 readallfields=true
2 readproportion=0
3 updateproportion=0
4 scanproportion=0.95
5 insertproportion=0.05
6 requestdistribution=zipfian
7 maxscanlength=100
8 scanlengthdistribution=uniform
```

Parametry `maxscanlength` oraz `scanlengthdistribution` określają odpowiednio maksymalną długość zapytania przedziałowego oraz rozkład prawdopodobieństwa przy losowaniu długości przedziału do odczytu (tutaj jednostajny).



(a) Operacje skanowania

(b) Operacje zapisu

Rysunek 4.3: Opóźnienie operacji w zależności od przepustowości, scenariusz `workloade`

Podczas testu system Cassandra osiągnął maksymalną przepustowość 310 operacji na sekundę, podczas gdy Gemius BigTable tylko 99. Podczas całego testu czas wykonania operacji odczytu z systemu Cassandra oscylował w granicach 10 milisekund. Natomiast dla GBT wartość ta wahała się od 30 do 40 ms przez większą część trwania testu. Jednakże przy mniejszych obciążeniach (do 30 operacji na sekundę) czas ten wynosił od 60 do 184 ms.

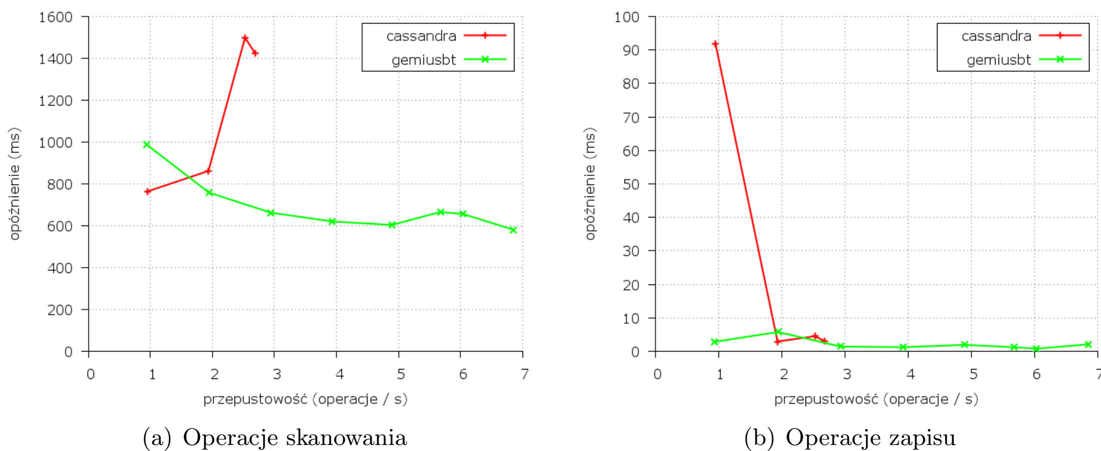
Ponownie podczas operacji modyfikującej (zapisu) Gemius BigTable charakteryzował się niższym opóźnieniem osiągając wartość od 0.26 do 1.16 ms. System Cassandra wykonywał te operacje w czasie od 1.15 do 2.06 ms. GBT charakteryzował się malejącym opóźnieniem wraz ze wzrostem przepustowości.

Dla systemu GBT gwałtowny spadek opóźnienia operacji skanowania na początku testu mógł być spowodowany tym, że obciążenie systemu wzrastało w trakcie trwania testu.

W konsekwencji z upływem czasu bloki dyskowe mogły zostać przechowywane w pamięci podręcznej, co w dalszej części testu skróciło czas dostępu do danych.

Długie przedziały

Test ten ma na celu zbadanie zachowania systemów podczas wykonywania skanowania długich przedziałów danych. Wykonywanych jest 95% operacji skanowania oraz 5% operacji zapisu. Konfiguracja testu jest niemal identyczna jak w przypadku krótkich odczytów przedziałowych. Różnicę stanowi parametr `maxscanlength`, który w tym teście ustalony jest na wartość 10000.



(a) Operacje skanowania

(b) Operacje zapisu

Rysunek 4.4: Opóźnienie operacji w zależności od przepustowości, scenariusz `longScans`

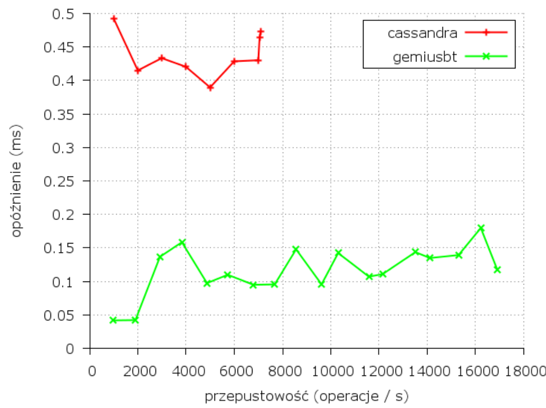
Skanowane przedziały były na tyle długie, że maksymalne przepustowości są istotnie mniejsze niż w poprzednich testach. Gemius BigTable osiągnął maksymalną przepustowość 6.85, zaś Cassandra 2.68 operacji na sekundę. Zachowanie Cassandry było podczas testu bardzo niestabilne. Podczas operacji skanowania opóźnienie wzrosło z 763 do 1500 ms, podczas gdy przy zapisie gwałtownie zmalało z 92 do 2.5 ms.

Gemius BigTable podczas całego testu zapisywał rekordy w czasie od 1 do 5 ms. Skanowanie rekordów dla GBT miało opóźnienie w granicach od 580 do 989 ms. Wraz ze wzrostem obciążenia widoczny był spadek opóźnienia odczytu przedziałowego.

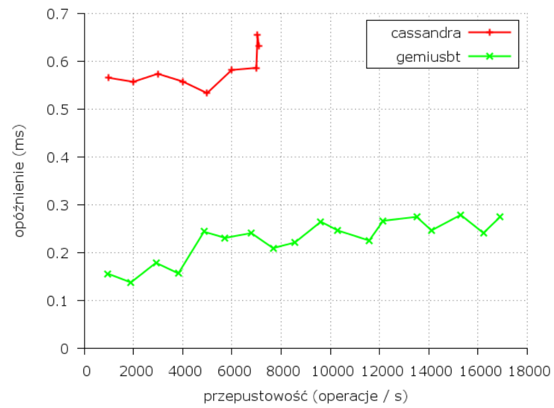
Tylko modyfikujące operacje

W teście tym zbadam zachowanie systemów w sytuacji występowania tylko operacji modyfikujących. Połowę z nich stanowią operacje zapisu, a połowę operacje aktualizacji.

```
1 readallfields=true
2
3 readproportion=0
4 updateproportion=0.5
5 scanproportion=0
6 insertproportion=0.5
7 readmodifywriteproportion=0
8
9 requestdistribution=uniform
```



(a) Operacje aktualizacji



(b) Operacje zapisu

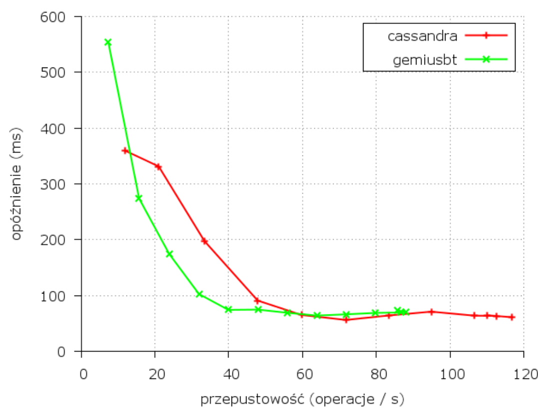
Rysunek 4.5: Opóźnienie operacji w zależności od przepustowości, scenariusz `onlyModify`

Gemius BigTable osiągnął bardzo wysoką maksymalną przepustowość 16905 operacji na sekundę. System ten osiągnął bardzo niskie opóźnienia zarówno podczas operacji zapisu (od 0.13 do 0.27 ms), jak i aktualizacji (od 0.04 do 0.17 ms). W przypadku obu operacji wraz ze wzrostem przepustowości nieznacznie wzrastało opóźnienie.

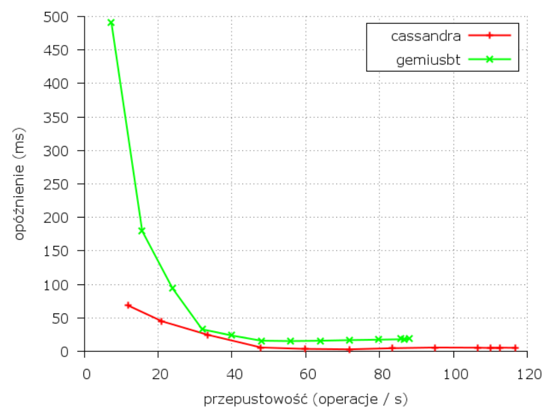
System Cassandra osiągnął maksymalne obciążenie na poziomie 7095 operacji na sekundę. Opóźnienie systemu Cassandra mieściło się w przedziale od 0.38 do 0.49 ms dla aktualizacji, zaś w przypadku zapisu był to przedział od 0.53 do 0.65 ms.

Tylko niemodyfikujące operacje

W teście tym wszystkie badane operacje są niemodyfikujące. Plik konfiguracyjny jest bardzo podobny do tego z poprzedniego testu, jednak rozkład po 50% jest na operacjach odczytu i skanowania.



(a) Operacje skanowania



(b) Operacje odczytu

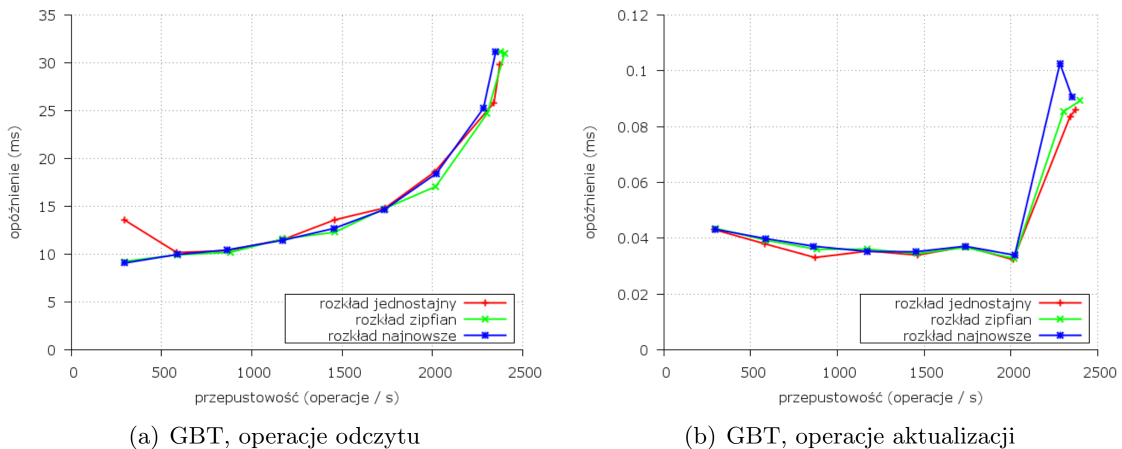
Rysunek 4.6: Opóźnienie operacji w zależności od przepustowości, scenariusz `onlyNonModify`

Maksymalne opóźnienie osiągnięte przez systemy to 88 dla GBT oraz 116 operacji na sekundę dla Cassandry. Opóźnienie operacji skanowania wynosiło od 63 do 553 ms dla GBT natomiast dla Cassandry od 61 do 359 ms. Dla operacji odczytu wartości te wynosiły od 15 do 490 ms w przypadku GBT, zaś dla systemu Cassandra od 3 do 68 ms.

Obydwa systemy wykazały wyraźną tendencję malejącą opóźnienia w stosunku do rosnącej przepustowości. Ze względu na niezmieniający się zbiór danych jest to najprawdopodobniej wynik działania pamięci podręcznej systemu operacyjnego.

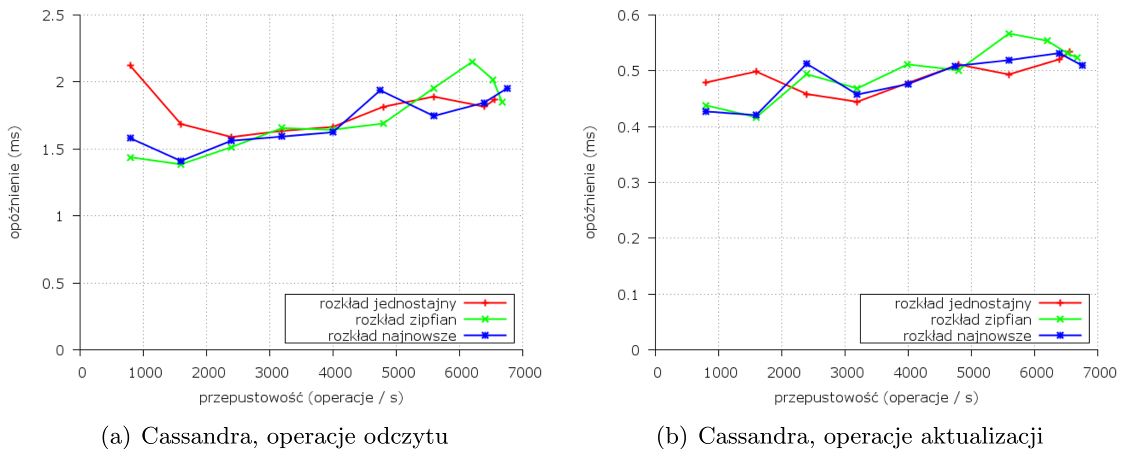
Rozkłady

Test ten porównuje zachowanie systemów przy wykonywaniu tych samych operacji przy różnych rozkładach operacji na danych. Obciążenie było generowane przy pomocy jednego z poprzednich plików konfiguracyjnych: `heavyInserts`, przy czym dla kolejnych testów zmieniane były rozkłady wykonywania operacji na danych.



(a) GBT, operacje odczytu (b) GBT, operacje aktualizacji

Rysunek 4.7: Opóźnienie operacji w zależności od rozkładu, Gemius BigTable



(a) Cassandra, operacje odczytu (b) Cassandra, operacje aktualizacji

Rysunek 4.8: Opóźnienie operacji w zależności od rozkładu, Cassandra

Opóźnienia wykonania operacji w stosunku do przepustowości w systemie Gemius BigTable były niemal identyczne bez względu na zastosowany rozkład wykonywania operacji. Minimalnie większe opóźnienie zostało zaobserwowane w przypadku rozkładu *najnowsze*.

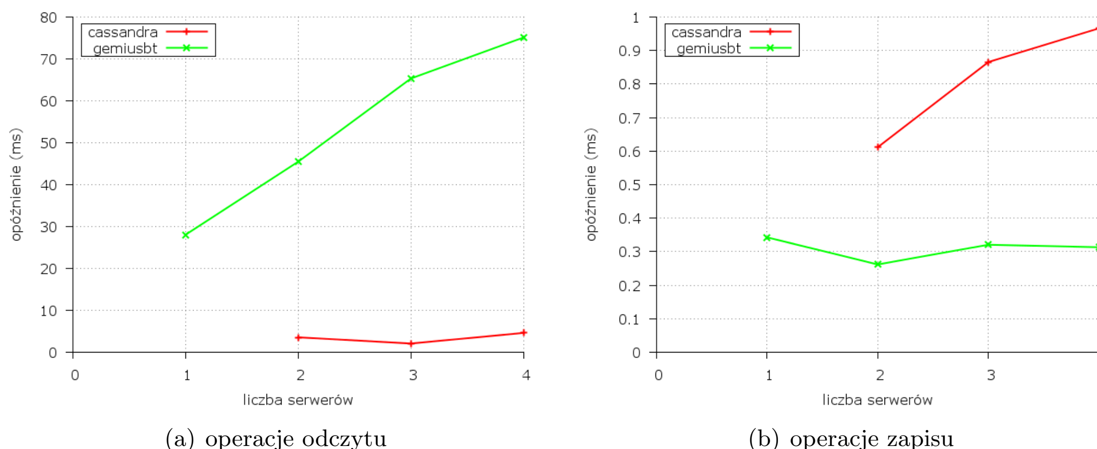
Dla poszczególnych punktów pomiarowych opóźnienia były bardziej zróżnicowane niż w przypadku Gemius BigTable. Nie mniej jednak dla operacji odczytu oraz aktualizacji opóźnienia wykazują tendencję rosnącą bez względu na zastosowany rozkład.

4.1.4. Skalowalność

W tym podrozdziale przedstawię wyniki testów mających na celu zbadanie zdolności systemu do skalowania się. Własności, które będę testować to *scaleup* oraz *speedup*. Opisałem je w podrozdziale 1.5.2.

scaleup

W tym teście zbadam jak testowane systemy reagują na zwiększone zasoby. Dla obu systemów przeprowadziłem ten sam scenariusz testowy (`heavyInserts`) przy różnych liczbach węzłów w systemie. Testowany był scenariusz w systemach z kolejnymi konfiguracjami od jednego do czterech węzłów obsługujących dane. Z kolejnymi, coraz większymi instancjami zwiększane było obciążenie poprzez zwiększanie liczby wątków generujących obciążenie. Liczba wątków klienckich była proporcjonalna do ilości serwerów. W teście, w którym działało k serwerów, obciążenie było generowane przez $2 * k$ wątków.



(a) operacje odczytu

(b) operacje zapisu

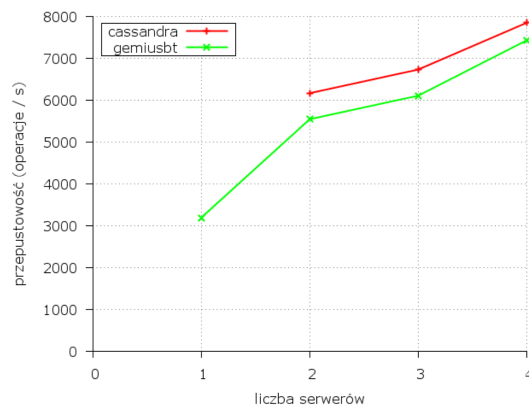
Rysunek 4.9: Wykres opóźnienie w zależności od ilości serwerów

W przypadku operacji odczytu opóźnienie operacji wzrastało monotonicznie z 27 do 75 ms wraz ze zwiększającym się rozmiarem systemu. Z kolei system Cassandra, bez względu na liczbę węzłów w systemie, zachowywał opóźnienie w granicach 4 ms.

Operacje zapisu rekordu Cassandra wykonywała najefektywniej (0.61 ms) w systemie z dwoma węzłami. W systemach z trzema i czterema węzłami danych opóźnienie wzrastało kolejno do 0.86 ms i 0.97 ms. Mimo zwiększającej się instancji systemu i liczby wątków klienckich opóźnienie zapisu dla systemu Gemius BigTable utrzymywało się w granicach od 0.26 do 0.34 ms.

Brak pomiaru systemu Cassandra przy konfiguracji z jednym węzłem jest spowodowany tym, że Cassandra nie przetwarza danych, gdy w systemie obecna jest mniejsza liczba serwerów niż wynosi współczynnik replikacji, który podczas wszystkich testów wynosił 2. Gemius BigTable nie doświadczył tego problemu, ponieważ replikacja jest realizowana przez system plików MooseFS.

Istotną zależnością w tym teście jest to, jak zmienia się przepustowość systemu, gdy test powtarzany jest z coraz większą liczbą serwerów danych.



(a) przepustowość

Rysunek 4.10: Wykres przepustowość w zależności od liczby serwerów

W kolejnych testach GBT osiągał kolejno przepustowości: 3187, 5544, 6106, 7431 operacji na sekundę. Bardzo podobnie zachowała się Cassandra osiągając kolejne przepustowości: 6166, 6731, 7851.

Koniecznym warunkiem do dobrej skalowalności jest zwiększanie przepustowości systemu dla coraz większych instancji. Ponadto, aby mieć cechę dobrego skalowania się, opóźnienie operacji nie powinno rosnąć. W efekcie obydwa testowane systemy wykazują się dość dobrą skalowalnością. Przepustowości Cassandry oraz GBT wzrastały dla testów w coraz to większych środowiskach. Cassandra utrzymała niemal stałe opóźnienie przy operacjach odczytu, jednakże wzrost systemu implikował zauważalny wzrost czasu trwania zapisu rekordu. Dla odmiany GBT zachował mniej więcej stałe opóźnienie, jednak czas opóźnienie odczytu znacząco wzrósł przy większym teście.

Przeprowadzone testy *scaleup* potwierdziły wyniki autorów YCSB z artykułu [Coo10]. Obydwa systemy wykazały się dobrą skalowalnością, jednak zostaje wciąż szerokie pole do ulepszania tej własności.

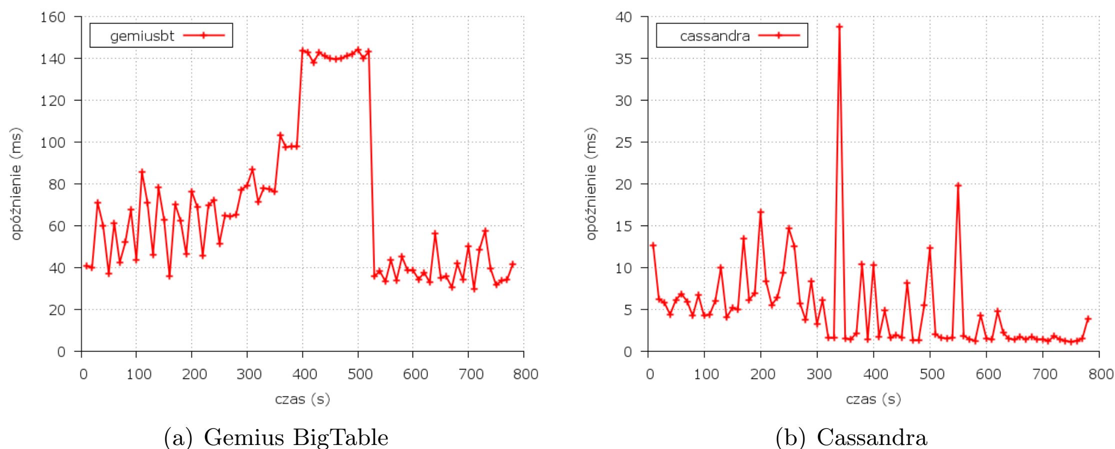
speedup

Test ten ma na celu zbadanie w jaki sposób system dostosowuje się do nowego węzła pojawiającego się w systemie. Obciążeniem zastosowanym w systemie było `heavyInserts`. Do testu użyłem instancji baz danych użytych w podrozdziale 4.1.3. System początkowo pracował z trzema serwerami. Po pięciu minutach został podłączony czwarty serwer, a test trwał do momentu ustabilizowania się systemu.

Po podłączeniu dodatkowego serwera (rysunek 4.11) do instancji bazy danych Gemius BigTable opóźnienie wzrastało w ciągu 100 sekund od 80 do około 140 ms. Na tym poziomie utrzymało się przez 2 minuty, po czym zmalało do 40 ms i na tym poziomie utrzymywało się do końca testu.

Przed podłączeniem dodatkowego serwera do systemu Cassandra opóźnienie operacji odczytu wynosiło od 5 do 17 ms. Tuż po rozszerzeniu systemu opóźnienie na kilka sekund gwałtownie wzrosło do 39 ms. Następnie spadło do 2 ms, jednak przez następne 5 minut opóźnienie chwilowo wzrastało do 10–20 ms. Po tym czasie opóźnienie wykonania odczytu ustabilizowało się na poziomie około 2 ms.

W GBT po pojawienia się nowego serwera powstały dwa nowe tablety w wyniku podziału, co w efekcie doprowadziło do rozłożenia obciążenia. Natomiast chwilowe zwiększenie opóź-



(a) Gemius BigTable (b) Cassandra

Rysunek 4.11: Wykres opóźnienia odczytu przy podłączeniu dodatkowego serwera

nienia mogło być spowodowane nieaktualnymi informacjami klientów o rozkładzie tabletów w systemie oraz nieobecnością danych w pamięci podręcznej nowego serwera.

Ostatecznie, w wyniku dodania kolejnego serwera, opóźnienie operacji odczytu w systemie Gemius BigTable zmalało z około 50 do około 40 ms. W przypadku systemu Cassandra opóźnienie zmalało z około 7 do około 2 ms po dodaniu czwartego serwera. Obydwa systemy wykazały się elastycznością przyspieszenia (*speedup*). Szczególnie dobrze zaprezentowała się pod tym względem Cassandra. Są to odmienne obserwacje od tych, które prezentowane są przez autorów YCSB w artykule [Coo10]. Wówczas Cassandra stabilizowała się wiele godzin. Testowana była tam wersja 0.5.0. Sugeruje to, że własność elastycznego przyspieszenia znacząco się poprawiła od wersji 0.5.0 do wersji 0.7.0.

4.1.5. Wnioski

Środowisko, w którym przeprowadziłem testy, było istotnie mniejszych rozmiarów niż to, które opisali autorzy YCSB w artykule [Coo10] podczas testowania szerszej gamy systemów. W efekcie największe osiągane przepustowości są mniejsze, niż te zaprezentowane we wspomnianym artykule. Nie mniej jednak wiele zawartych tam wniosków na temat systemu Cassandra potwierdziłem w wyniku moich testów, np:

- podczas testu "Intensywna aktualizacja i odczyt" Cassandra zachowuje przez cały czas trwania testu stałe opóźnienie;
- Cassandra dobrze się skaluje zachowując stałe opóźnienie odczytu przy zwiększającym się systemie i obciążeniu;
- odczyt przedziałów danych (szczególnie długich) jest mało efektywny.

Według Cassandry oraz Gemius BigTable są one zoptymalizowane pod kątem wykonywania operacji zapisujących. System GBT wykonuje znacznie efektywniej operacje modyfikujące niż system Cassandra. Niestety jednak główną zaobserwowaną wadą systemu Gemius BigTable jest na tyle wolne wykonywanie operacji losowych odczytów danych, że nie może on być użyty w praktycznych zastosowaniach, które intensywnie pobierają pojedyncze rekordy. Najefektywniejszą metodą dostępu do danych dla GBT jest przedziałowy odczyt danych, szczególnie dla długich przedziałów.

Zachowanie systemów podczas testów z różnymi rozkładami dostępu do danych miało podobne trendy bez względu na zastosowaną sekwencję operacji na danych. Podobnie zarówno GBT, jak i Cassandra wykazały się dobrą skalowalnością. Poprawienie skalowalności odczytu w GBT, zaś zapisu w Cassandra to wyzwania jakie czekają na twórców tych systemów.

Testy przeprowadziłem z największą starannością, jednak pewne aspekty wciąż wymagają poprawy podczas powtórzenia testów. Podczas całego procesu testowania w tle działało kilka mało aktywnych procesów, których nie mogłem wyłączyć. Ponadto infrastruktura testowa była na tyle małych rozmiarów, że klient YCSB działał na tym samym serwerze, co jeden z węzłów testowanego systemu.

Pomimo, że w artykule opisującym YCSB [Coo10] autorzy wskazują na dostępność operacji `delete()`, w wersji wykorzystanej do testów nie była dostępna.

4.2. Testowanie polityki łączenia

W podrozdziale tym przedstawię wyniki testów przeprowadzonych za pomocą symulatora obciążenia opisanego w podrozdziale 3.3. Następnie ten sam scenariusz testowy przeprowadzę na rzeczywistym systemie w środowisku rozproszonym.

4.2.1. Ogólny opis

Jako, że symulator nie pracuje fizycznie na danych, część testu związaną z symulatorem wykonałem na lokalnym komputerze. Test w środowisku rozproszonym został wykonany z trzema serwerami tabletów w tym samym środowisku, które opisałem w podrozdziale 4.1.

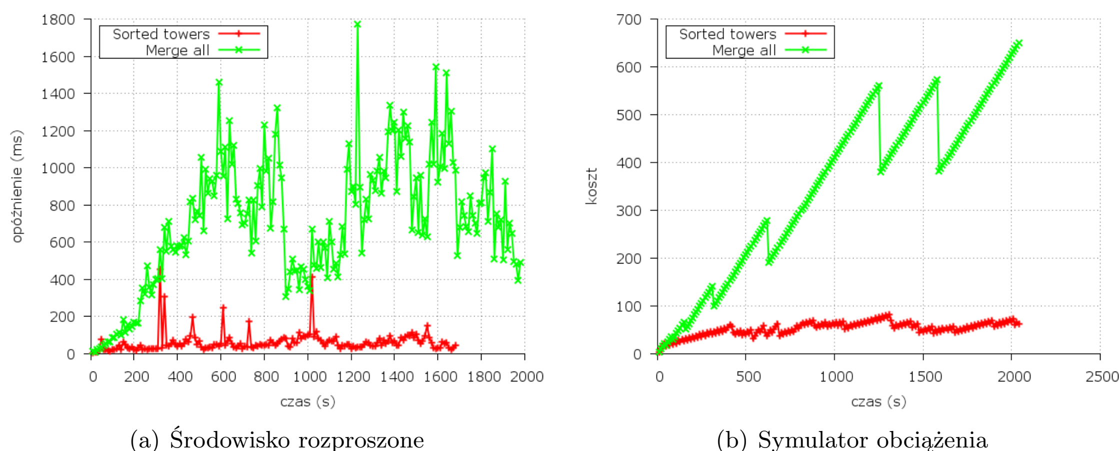
Testy w pierwszej kolejności przeprowadziłem w środowisku rozproszonym. Do testu użyłem scenariusza `mergeSimulator`. Następnie uruchomiłem analogiczny scenariusz na symulatorze. Do symulatora została przekazana liczba operacji na sekundę taka, jaką została uzyskana w teście w środowisku rozproszonym.

```
1  operationcount=20000000
2
3  readallfields=true
4
5  readproportion=0.0001
6  updateproportion=0
7  scanproportion=0
8  insertproportion=0.9999
9
10 requestdistribution=uniform
```

Symulator został uruchomiony z taką samą przepustowością rekordów jaka została uzyskana w środowisku rozproszonym, tj. 9790 operacji na sekundę. Liczba rekordów w jednej transakcji była taka sama jak w transakcji klienckiej. Podobnie jak w rzeczywistym środowisku, symulator modelował przetwarzanie 20 milionów rekordów. Testowane polityki (opisałem je w podrozdziale 3.3.2) były skonfigurowane z parametrami:

- *Merge all*: `MergePolicySortedTowersN 5 100`;
- *Sorted towers*: `MergePolicyAllWhenMoreThanN 5 100`;

4.2.2. Wyniki testów



(a) Środowisko rozproszone

(b) Symulator obciążenia

Rysunek 4.12: Opóźnienie odczytu ze względu na politykę łączenia

Na obu wykresach zamieszczonych na rysunku 4.12 oś pozioma określa czas zmierzony w sekundach od momentu rozpoczęcia testu. Oś pionowa na rysunku 4.12(a), podobnie jak podczas testów z podrozdziału 4.1, określa czas wykonania operacji (mierzony w milisekundach). Natomiast oś pionowa z rysunku 4.12(b) oznacza koszt wykonania operacji. Pojęcie kosztu zdefiniowałem w podrozdziale 3.3.

Podczas testowania polityki *Sorted towers* symulator wskazał rosnący koszt operacji odczytu przez pierwsze 500 sekund testu do wartości około 50. Następnie do końca testu koszt utrzymywał się w przedziale 50–75. Koszt polityki *Merge all* przez pierwsze 1200 sekund systematycznie rósł do wartości 570. Następnie gwałtownie spadał do 380. Proces systematycznego wzrastania kosztu oraz gwałtownego spadania ma charakter powtarzalny.

Kształt wykresów z rysunku 4.12(a) odbiega nieco od tych z sąsiedniego rysunku. Główną różnicą jest to, że punkty pomiarowe na rysunku 4.12(a) mają znacznie większe odchylenie od głównego trendu. Jednak główne tendencje pozostają zachowane. Przy zastosowaniu polityki *Sorted towers* opóźnienie odczytu wzrastało przez pierwsze 500 sekund do wartości 50 ms, a następnie utrzymywało się w przedziale 30–70 ms. Widoczne są sporadyczne, chwilowe wzrosty opóźnienia do wartości nawet 270 ms. Użycie polityki *Merge all* spowodowało przez pierwsze 650 sekund systematyczny wzrost czasu trwania odczytu do wartości około 1 sekundy. Widać gwałtowne spadki opóźnienia odczytu, np. w 680., czy 1700. sekundzie.

4.2.3. Wnioski

Przedstawione wyniki testów wyraźnie obrazują fakt, że efektywność wykonania odczytów w systemie Gemius BigTable jest bardzo silnie powiązana ze strategią wyboru plików do łączenia. Mimo iż polityka *Merge all* w wyniku złączenia powoduje zmaterializowanie wartości rekordów, to jednak polityka *Sorted towers* wykazuje lepsze wyniki.

Mimo iż symulator obciążenia nie jest w stanie idealnie modelować operacji wykonywanych na tablecie, to jednak główne trendy związane z czasem odczytu rekordu w realnym środowisku mają swoje bezpośrednie przełożenie na wyniki generowane przez symulator.

Dla systemu GBT równie ważną cechą jest odpowiedni dobór polityki podziału tabletu (istotę podziału tabletu opisałem w podrozdziale 2.4). Obecny symulator można rozszerzyć o symulowanie wielu tabletów dzielonych przy pomocy testowanej polityki.

Rozdział 5

Podsumowanie

W niniejszej pracy magisterskiej opisałem ogólną charakterystykę bazodanowych systemów składowania danych NoSQL. Ze względu na fakt, że mogą one działać zarówno na domowym komputerze klasy PC, jak i środowisku produkcyjnym wielkich firm, szybko zyskały popularność w informatycznych kręgach. Przyjrzałem się kilku najpopularniejszym rozwiązaniom i przedstawiłem ich ogólną charakterystykę. Opisałem również produkt Yahoo! Cloud Serving Benchmark stworzony w celu dostarczenia środowiska do testowania w jednolity sposób wydajności rozproszonych systemów bazodanowych.

W dokumencie zawarłem również szczegółowy opis systemu bazodanowego Gemius BigTable, który rozwijałem w ramach pracy zawodowej. Dostarczyłem narzędzia wspomagające testowanie wydajności tego systemu. Mimo pewnych niezgodności YCSB w wersji 0.1.4 z dokumentacją, udało mi się w pełni zintegrować Gemius BigTable z narzędziem YCSB. Stworzyłem także symulator obciążenia tabletu, którego ocena kosztu wykonania odczytu danych jest skorelowana z czasem wykonania tej operacji w rzeczywistym systemie.

W ramach pracy zaproponowałem zestaw testów, a następnie użyłem wspomnianych narzędzi do zbadania wydajności tytułowego systemu na tle jednego z popularnych rozwiązań bazodanowych Cassandra. Te z testów, które powtórzyłem po twórcach systemu YCSB, potwierdzają wyniki uzyskane przez pracowników firmy Yahoo!. Pozostałe natomiast sugerują, że optymalizowanie systemu pod kątem jednej operacji ma swój koszt podczas wykonywania innych zadań. Podczas testów Gemius BigTable znacznie efektywniej wykonywał operacje modyfikujące dane. Z drugiej jednak strony odczyt pojedynczych rekordów z systemu jest skrajnie nieefektywny.

W ramach pracy dokonałem porównania polityk łączenia plików. Otrzymane wyniki wskazują, że parametr ten ma istotny wpływ na efektywność dostępu do danych. Niewątpliwie odczyt pojedynczych rekordów wymaga poprawy, aby zbliżyć się do akceptowalnego czasu wykonania tej operacji. Przedstawiona przeze mnie ewaluacja możliwości systemu Gemius BigTable może stanowić punkt wyjścia do dalszych badań.

Dodatek A

Zawartość płyty

Na załączonej płycie znajdują się następujące pliki:

- `1000-MGR-INF-88021314434.pdf` — dokument mojej pracy magisterskiej w wersji elektronicznej,
- `ycsb` — projekt YCSB w wersji 0.1.4 z moimi rozszerzeniami,
- `workloadSimulator` — symulator obciążenia tabletu Gemius BigTable.

Bibliografia

[ApaHB] Apache HBase, `http://hbase.apache.org/`.

[Bar10] P. Barry, *Doing IT the App Engine Way.* Linux Journal, Volume 2010 Issue 197, wrzesień 2010.

[Cha08] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber, *Bigtable: A Distributed Storage System for Structured Data*, ACM Transactions on Computer Systems (TOCS), Volume 26 Issue 2, czerwiec 2008.

[Coo10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears. *Benchmarking Cloud Serving Systems with YCSB*, In ACM Symposium on Cloud Computing, 2010.

[Dum04] C. Dumitrescu, I. Raicu, M. Ripeanu, I. Foster, *DiPerF: An Automated DIstributed PERformance Testing Framework.* GRID '04 Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, 2004.

[Ghe03] S. Ghemawat., H. Gobioff, S. T. Leung, *The Google file system.* In Proc. of the 19th ACM SOSP, 2003.

[Gup11] D. Gupta, K. V. Vishwanath, A. Vahdat, *DieCast: Testing Distributed Systems with an Accurate Scale Model.* ACM Transactions on Computer Systems (TOCS), Volume 29 Issue 2, maj 2011.

[Hup09] K., Huppler, *The Art of Building a Good Benchmark.* Performance Evaluation and Benchmarking, Springer-Verlag Berlin, Heidelberg 2009.

[Lak10] A. Lakshman, P. Malik, *Cassandra: a decentralized structured storage system.* ACM SIGOPS Operating Systems Review, Volume 44 Issue 2, kwiecień 2010.

[Gad10] M. Gądarowski, *MooseFS: Bezpieczny i rozproszony system plików.* Linux Magazine Poland, kwiecień 2010.

[Pok11] J. Pokorny, *NoSQL databases: a step to database scalability in web environment.* iiWAS '11 Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services, 2011.

[Wek10] T. Weksej, *Niezawodność w rozproszonych systemach bazodanowych.* Praca magisterska, Instytut Informatyki Uniwersytetu Warszawskiego, czerwiec 2010.