POLSKO-JAPOŃSKA WYŻSZA SZKOŁA TECHNIK KOMPUTEROWYCH

PRACA MAGISTERSKA

Nr ....

Budowa indeksu dla systemu OfficeObjects Portal z wykorzystaniem struktur SDDS

Studenci	**Michał Ziółkowski, Piotr Wilczyński**
Numer	s1824, s1823
Promotor	prof. dr hab. Kazimierz Subieta
Specjalność	Inżynieria Oprogramowania i Baz Danych
Katedra	Systemów Informacyjnych
Data Zatwierdzenia Tematu	
Data Zatwierdzenia Pracy	

Podpis promotora pracy	**Podpis kierownika katedry**

Streszczenie

Praca dotyczy problemu integracji dwóch niezależnych systemów informatycznych, SDDS-2000 oraz OfficeObjects Portal. Opisana została w niej architektura stworzonych rozwiązań, przedstawione szczegóły implementacji oraz interakcji pomiędzy poszczególnymi komponentami. Wyjaśnia też dlaczego zdecydowano się na określone podejście do problemów.
W pracy przedstawiono systemy podlegające integracji, SDDS-2000 oraz OfficeObjects Portal. Głównym owocem pracy okazał się system SDDS-LH*, którego powstanie zostało zainspirowane niedoskonałością pakietu SDDS-2000, a który nie był początkowo planowany jako jej część. Z tego powodu szeroko opisano algorytmy LH, LH* będące podstawą koncepcji SDDS-LH*.
Praca zwiera spostrzeżenia dotyczące wydajności działania systemu OOP z punktu widzenia użytkownika-autora aplikacji opartej na OfficeObjects Portal. Opisane zostały różnice w czasie dostępu do obiektów z zastosowaniem systemów SDDS oraz bez nich.
Na koniec opisano problemy na jakie natknięto sie podczas realizacji pracy, oraz zasugerowano możliwe dalsze prace nad stworzonym oprogramowaniem.

Spis treści

Rozdział 1.Wstęp....5
1.1.Cel pracy....5
1.2.Rezultaty pracy....5
Rozdział 2.OfficeObjects Portal....6
2.1.Opis....6
2.2.Moduły OfficeObjects Portal....7
2.3.Architektura 8
Rozdział 3.Indeksy 10
Rozdział 4.Mieszanie liniowe....14
4.1.Wstęp....14
4.2.Podstawy mieszania....15
4.3.Definicja funkcji mieszającej....16
4.4.Dokonywanie podziałów....16
4.5.Wyliczanie adresu....17
4.6.Kontrola wypełnienia oraz zmniejszanie przestrzeni....19
4.7.Przykład działania....19
Rozdział 5.SDDS LH*....22
5.1.Wstęp....22
5.2.Rozszerzanie pliku....23
5.3.Adresowanie....24
5.4.Obliczanie adresu po stronie klienta....25
5.5.Obliczanie adresu po stronie serwera....27
5.6.Korekcja obrazu klienta....30
5.7.Dokonywanie podziałów....31
5.7.1.Podziały niekontrolowane....31
5.7.2.Podziały kontrolowane....31
5.8.Kurczenie się plików....33
5.9.Umiejscawianie kubełków....34
5.10.Zapytania równoległe....35
5.10.1.Propagacja zapytań równoległych....35
5.10.2.Korekcja obrazu....36
5.11.Wydajność....37
5.12.Podsumowanie....38
Rozdział 6.Algorytmy LH*m, LH*s, LH*g....39
6.1.Wstęp....39
6.2.Algorytm LH*m....40
6.2.1.Pliki lustrzane strukturalnie jednakowe....41
6.2.2.Pliki lustrzane strukturalnie odmienne....44
6.3.Algorytm LH*s....46
6.3.1.Podstawy LH*s....46
6.3.2.Operacje na pliku....47
6.3.3.Odtwarzanie kubełków....49
6.3.4.Bezpieczeństwo....49
6.3.5.Wydajność....49
6.4.Algorytm LH*g....49
6.4.1.Struktura pliku LH*g....50
6.4.2.Operacje na pliku....51
6.4.3.Odtwarzanie kubełka podstawowego....52
6.4.4.Odtwarzanie kubełka parzystości....52
6.4.5.Odtwarzanie stanu pliku....52

6.4.6.Wydajność....53
6.4.7.Pliki LH*g z n-dostępnością....53
6.5.Porównanie LH*, LH*m, LH*s, LH*g....54
6.5.1.Wyszukiwanie....54
6.5.2.Wstawianie....54
6.5.3.Przestrzeń składu....55
6.5.4.Dostępność....55
6.5.5.Odtwarzanie kubełków....55
6.5.6.Przepustowość....55
6.5.7.Bezpieczeństwo....55
6.6.Podsumowanie....56
Rozdział 7.Implementacja....57
7.1.Rozwiązania przyjęte w pracy....57
7.1.1.Java....57
7.1.2.C++....58
7.2.SDDS-2000....59
7.2.1.Algorytmy z rodziny RP*....60
7.2.2.Architektura systemu....61
7.2.3.Uwagi....62
7.3.SDDS-LH*....63
7.3.1.Wstęp....63
7.3.2.Założenia SDDS-LH*....64
7.3.3.Koordynator....65
7.3.4.Serwer....66
7.3.5.Klient....67
7.3.6.Tworzenie plików....68
7.3.7.Operacja wyszukiwania rekordu....68
7.3.8.Operacja wstawiania rekordu....70
7.3.9.Operacja usuwania rekordu....70
7.3.10.Podział kubełków....70
7.3.11.Złączenia kubełków....71
7.3.12.Kontrola wypełnienia....71
7.3.13.Zapytania równoległe....72
7.3.14.API SDDS-LH*....72
7.3.15.Wielowątkowość....73
7.3.16.Podsumowanie....74
7.4.Wrapper systemu OfficeObjects Portal....75
7.5.API indeksu dla systemu OOP....76
7.5.1.Indeks zbudowany na systemie SDDS-2000....78
7.5.2.Indeks zbudowany na systemie SDDS-LH*....88
7.6.Funkcja mieszająca....92
7.7.Testowanie....94
7.8.Testy wydajnościowe....96
7.8.1.Metoda testowania....96
7.8.2.Wyniki....96
7.8.3.Podsumowanie....98
Rozdział 8.Napotkane trudności....99
8.1.Problemy....99
8.1.1.SDDS-2000....99
8.1.2.OfficeObjects Portal....99
8.2.Plany....99
Rozdział 9.Zakończenie....101

Rozdział 1. Wstęp

Wydajność systemów informatycznych jest dziś jednym z kluczowych czynników decydujących o jakości danego produktu. Pozwala to nie tylko zdobyć przewagę nad konkurencyjnymi rozwiązaniami, ale także (a może przede wszystkim) sprostać wymaganiom użytkowników. Obecnie system musi poradzić sobie z obciążeniem generowanym przez tysiące klientów pracujących jednocześnie, zapewniając przy tym odpowiedni komfort pracy.
Dla systemów, których głównym zadaniem jest zbieranie i przechowywanie danych jednym z najpowszechniej stosowanych mechanizmów pozwalających na przyspieszenie pracy systemu są indeksy. Indeksy pomagają w szybki sposób odnaleźć określony obiekt istniejący w repozytorium informacji.

1.1. Cel pracy

Celem niniejszej pracy magisterskiej było stworzenie API dla programistów pozwalające na wykorzystanie mechanizmu indeksów w systemie OfficeObjects Portal. Ze względu na brak tej funkcjonalności w OOP wykorzystano zewnętrzny system do przechowywania danych o zaindeksowanych obiektach. Jako system zarządzający indeksami zastosowano produkt SDDS-2000, oraz w wersji drugiej własną implementację SDDS (skrót z ang. Scalable Distributed Data Structure – Skalowalna Rozproszona Struktura Danych). Dodatkowo należało sprawdzić czy stosowanie tego rozwiązania przyniesie wymierna korzyści w postaci wzrostu wydajności wyszukiwania.

1.2. Rezultaty pracy

Rezultatem tej pracy magisterskiej miało być API dla programistów OOP korzystających z SDDS-2000 jako repozytorium indeksów. W wyniku prac oraz testów postała dodatkowo własna implementacja systemu SDDS. W efekcie powstały dwie osobne wersje API pozwalające korzystać z obu systemów. Obie biblioteki są funkcjonalne, choć nie w takim samym zakresie. Wersja dla systemu SDDS-2000 pozwala jedynie na tworzenie nowych indeksów oraz wyszukiwanie danych w juz istniejących indeksach. Wersja dla autorskiego projektu pozwala dodatkowo na aktualizację indeksów, wstawianie nowych rekordów oraz usuwanie już niepotrzebnych danych.
W wyniku zastosowania zewnętrznego systemu indeksów zauważono istotne przyspieszenie procesu wyszukiwania.

Rozdział 2. OfficeObjects Portal

Rozdział ten zawiera krótki opis oraz charakterystykę systemu OfficeObjects Portal. System ten jest jednym z dwóch kluczowych elementów na których opiera się ta praca magisterska.

2.1. Opis

OfficeObjects Portal jest platformą narzędziową wykorzystywaną do tworzenia własnych internetowych portali informacyjnych. Na produkt ten składa się wiele bibliotek i modułów, które dostarczają określone usługi. Zadaniem ich jest dostarczenie odpowiedniej informacji szerokiej rzeszy użytkowników, oraz umożliwienie zdalnej pracy rozproszonym użytkownikom systemu opartego na OOP. Typowy portal który jest zbudowany z wykorzystaniem technologii OOP dostarcza następującej funkcjonalności:

- bezpieczne składowanie informacji w centralnym magazynie danych portalu (repozytorium),
- kontrola w udostępnianiu danych poprzez sieć Internet przy pomocy przeglądarki stron WWW,
- umożliwienie wyszukiwania danych w repozytorium (w tym wyszukiwanie strukturalne po atrybutach, którymi opisane zostały obiekty informacyjne oraz wyszukiwanie pełnotekstowe z uwzględnieniem polskiej fleksji),
- grupowanie danych w hierarchiczne struktury w sposób optymalnie wykorzystujący magazyny danych będące pod kontrolą portalu.
- przeglądanie zawartości plików przechowywanych w repozytorium bez potrzeby instalowania dodatkowych aplikacji czytającej dany format pliku.
- za pomocą przeglądarki WWW OfficeObjects Portal pozwala przeglądać zawartość ponad 250 formatów plików,
- wielopoziomowe zabezpieczanie informacji oferowanych przez portal
- łatwe definiowanie nowych struktur danych występujących w repozytorium: obiektów niosących określone informacje, struktur grupujących te obiekty, słowników, zarządzanie prawami dostępu użytkowników,
- raportowanie wytwarzające opracowania oparte na danych składowanych w OOP
- rejestrowanie operacji wykonanych na wybranych obiektach lub przeprowadzone przez określonych użytkowników,
- możliwość wymiany danych z dowolnymi systemami zewnętrznymi wspierającymi uniwersalną technologię wymiany informacji opartą o język XML
- zarządzanie procesami pracy i powiązanymi zasobami w oparciu o standardy koalicji WorkFlow Management Coalition.

Powyższa funkcjonalność możliwa jest do uzyskania dzięki modularnej i warstwowej budowie systemu. System OOP opiera się na serwerze aplikacyjnym (np. IBM WebSphere), oraz obiektowo-relacyjnym systemie zarządzania bazą danych, np. Oracle.
System OOP zbudowany jest zgodnie ze standardem J2EE. Do wymiany informacji stosuje standard XML Schema oraz protokół SOAP. Dzięki zastosowaniu serwera aplikacyjnego zgodnego z J2EE portal oparty na OOP jest:

- skalowalny (poprzez wykorzystanie standardowych funkcji rozkładania obciążeń (load balancing)
- bardziej niezawodny poprzez możliwość jednoczesnej pracę wielu replikom środowiska (J2EE clones)

2.2. Moduły OfficeObjects Portal

System OpenOffice Portal składa się z następujących modułów:
OfficeObjects Repository – odpowiada za przechowywanie obiektów reprezentujących określone informacje. Zajmuje się ich składowaniem, wyszukiwaniem, aktualizacjami, dodawaniem nowych obiektów itp.
OfficeObjects HSM (Hierarchical Storage Manager) - odpowiada za optymalne rozmieszczenie dużych porcji danych na dostępnych nośnikach (z uwzględnieniem czasów dostępu, szybkości itp.). Migracja pomiędzy nośnikami odbywa się na podstawie zdefiniowanych reguł i ścieżek migracji.
OfficeObjects ViewServer – poprzez przeglądarkę WWW udostępnia zawartość plików danych przechowywanych w OOP. Moduł ten pozwala na odczytanie plików w przeszło 250 formatach danych. Rozwiązanie jest oparte na bibliotekach Outside In firmy Stellent.
OfficeObjects VSTS – umożliwia pełnotekstowe wyszukiwanie informacji w portalach opartych na OOP. Wyszukiwanie uwzględnia polskie znaki diakrytyczne jak również polską fleksję. Moduł zbudowano na podstawie znanych i sprawdzonych bibliotek firmy Verity.
Narzędzie projektanta – panel administracyjny dla systemu opartego na OOP. Z tego miejsca projektant/administrator może tworzyć nowe struktury danych, zarządzać uprawnieniami, definiować słowniki pojęć itp. Moduł zrealizowano w postaci bibliotek programistycznych która swoje funkcje udostępnia poprzez interfejs API.

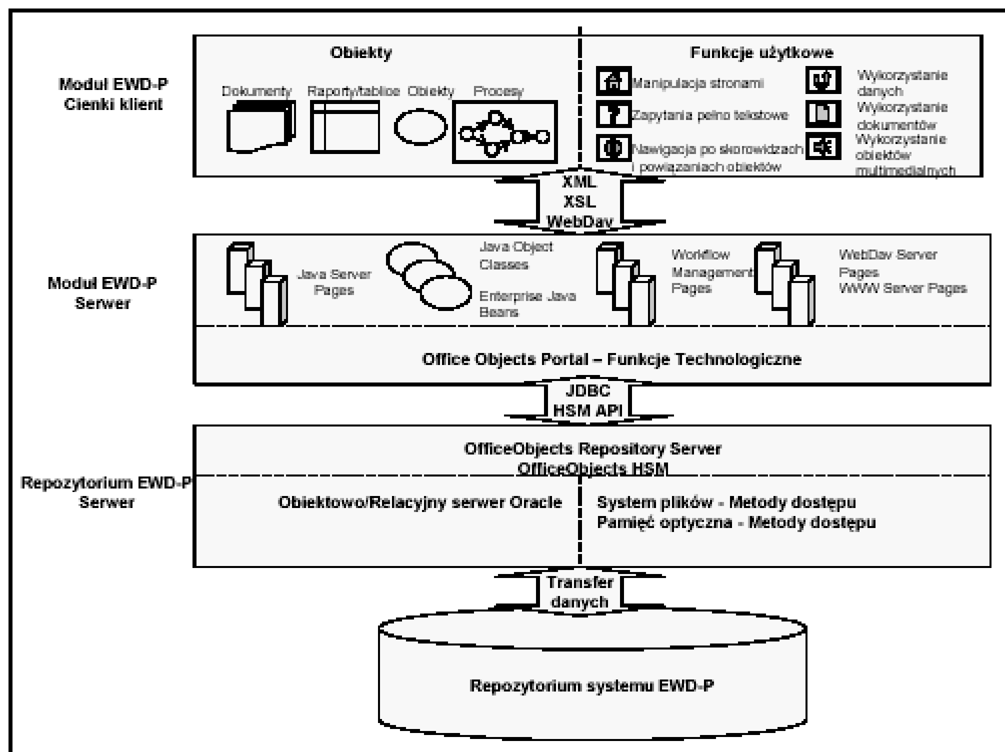
OfficeObjects Workflow – moduł ten dostarcza mechanizmów do zarządzania procesami pracy. W powiązaniu z repozytorium danych moduł ten udostępnia użytkownikowi swoje funkcje za pomocą przeglądarki internetowej. Informacje o procesach pracy zawarte w opisach typów zgodne są z wymaganiami Workflow Managment Coalition. Ważną cechą opisów jest pojęcie Roli, identyfikujące określoną grupę kompetencji i uprawnień. Pozwala to na automatyczny przydział pracy odpowiednim osobom, a także stanowi bardzo istotny aspekt zarządzania procesami pracy.

OfficeObjects e-Forms – pozwala na definiowanie własnych formularzy elektronicznych wizualizujących obiekty informacyjnych przechowywanych w bazie danych, jak również na definiowanie zaawansowanych formularzy stosowanych w systemach sprawozdawczości.

OfficeObjects Reporter – pomaga zarządzać raportami stworzonymi za pomocą różnych narzędzi raportowych, jak np. Crystal Reports.

2.3. Architektura

Kluczową sprawą jest założenie o trójwarstwowej architekturze tworzonych aplikacji. Architektura taka jest charakterystyczna w przypadkach, gdy medium komunikacyjnym jest Internet lub sieci intranetowe. Skalowalność zapewniana jest poprzez albo zwiększenie liczby procesorów albo poprzez przeniesienie aplikacji na wydajniejsze platformy. Możliwe jest to dzięki niezależności od rozwiązań stosowanych w warstwie niższej. Bezpieczeństwo z kolei zapewniane jest poprzez nałożenie mechanizmów zabezpieczających na każdą warstwę przetwarzania. Dodatkowym elementem jest wydzielenie stref ochrony odizolowanych od świata zewnętrznego i siebie za pomocą ścian ogniowych.



Ilustracja 1. Architektura przykładowego systemu opartego o OOP (zaczerpnięto z [OOP03])

Architektura przykładowego systemu opartego na systemie OfficeObjects Portal wygląda następująco:

warstwa zarządzaniem składowaniem danych – system zarządzania bazami danych, odpowiedzialny za składowanie informacji w portalu (np. Oracle, MsSQL Server, Informix)

warstwa narzędziowa portalu – moduły realizujące określone funkcje portalu (zarządzanie obiektami, wyszukiwanie danych)

warstwa uruchamiania aplikacji sieciowych w oparciu o serwer aplikacyjny – w warstwie tej funkcjonują obiekty OfficeObjects realizujące logikę biznesową portalu

warstwa komunikacyjna – oparta o protokół HTTP, realizuje przesyłanie informacji do użytkownika. Transfer danych poufnych odbywa się poprzez protokół HTTPS wykorzystujący mechanizm Secure Socket Layer (SSL)

warstwa interfejsu użytkownika- odpowiedzialna za prezentację danych oraz udostępnianie interfejsu do wykorzystania funkcjonalności portalu. Warstwa ta funkcjonuje na bazie przeglądarki WWW (np. Internet Explorer) lub w oparciu o aplikację stacjonarną.

Na podstawie:

[OOP03]

Rozdział 3. Indeksy

Indeksy są pomocniczą strukturą danych przechowywaną po stronie serwera. Wykorzystywane są one do przyspieszenia wyszukiwania w dużych zbiorach danych. W momencie stwierdzenia konieczności przyspieszenia wyszukiwania administrator może utworzyć (a także usunąć) indeks na odpowiednich danych.

Indeksy są strukturą redundantną w stosunku do oryginalnych danych. W związku z tym wymagane jest przeznaczenie dodatkowego miejsca na jego utworzenie. Jednak rozmiar indeksu najczęściej jest stosunkowo mały w stosunku do indeksowanych danych, a przyspieszenie wyszukiwania osiągane dzięki indeksom znaczne. Dlatego też stosowanie indeksów jest bardzo korzystne i na ogromny wpływ na prędkość działania całego systemu. Trudno sobie wyobrazić dzisiejsze systemy zawierające duże ilości danych bez indeksów. Są one stosowane praktycznie we wszystkich systemach zarządzania bazami danych.

W najprostszej postaci indeks jest to dwuwymiarowa tabela. W pierwszej kolumnie przechowywane są wartości kluczowe, w drugiej niekluczowe, Rys1. Wartości kluczowe są unikalne, służą od odnajdowania odpowiedniej pozycji w tabeli. Stanowią one jeden z atrybutów indeksowanego obiektu (lub są obliczane na jego podstawie), po którym najczęściej odbywa się wyszukiwanie. W kolumnie z wartościami niekluczowymi znajdują się referencje do obiektów, rekordów w tabelach, adresy sektorów na dysku itp. Pozwala to, tak jak w przypadku indeksów stosowanych w książkach, w szybki sposób odnaleźć poszukiwaną wartość lub obiekt.

Klucz	***Wartość***
Kowalski	ref#Kowalski
Nowak	ref#Nowak

Ilustracja 2. Przykładowy indeks

Indeksy powinny odznaczać się dwoma własnościami:

przeźroczystość – programista nie musi uwzględniać istnienia indeksów, tj. odwoływać się do nich bezpośrednio. Dzięki temu administrator może generować indeksy zależnie od rozpoznanych potrzeb oraz typu zapytań. Zmiany mogą być dokonywane w każdej chwili, bez potrzeby modyfikacji kodu samej aplikacji.

automatyczna aktualizacja – występuje wtedy, gdy w wyniku dokonania jakiś zmian bazie danych zmieniane (aktualizowane) są także indeksy. Proces ten powinien występować automatycznie. Jeśli to nie nastąpi, dane mogą utracić spójność.

(ze względu na specyfikę projektu indeks powstały w wyniku tej pracy magisterskiej nie jest ani przeźroczysty ani automatycznie aktualizowany)

Indeksy można sklasyfikować w następujący sposób:

- indeksy główne
- indeksy wtórne
- indeksy gęste
- indeksy zakresowe

Indeks główny zakładany jest na unikalny klucz w danej kolekcji obiektów, kolumnie w tabeli w unikalnymi wartościami itp. Dla określonego klucza wyszukiwania zwraca on tylko jedną wartość. Indeks główny gwarantuje, że nie ma duplikatów w zbiorze wartości kluczowej. Przykładem może być indeks z numerem PESEL lub numerem rejestracyjnym samochodu jako wartością kluczową.

Klucz	***Wartość***
WA-223J	ref#samochod1
WX-878K	ref#samochod2
WR-234C	ref#samochod3

Ilustracja 3. Przykładowy indeks główny

Indeks wtórny dotyczy atrybutu, który nie jest kluczowy dla danej kolekcji obiektów. Dlatego też indeks wtórny może zwrócić więcej niż jedną wartość dla określonego klucza wyszukiwania. Przykładowym atrybutem indeksowania może być nazwisko.

Klucz	***Wartość***
Kowalski	ref#osoba1, ref#osoba2
Nowak	ref#osoba3

Ilustracja 4. Przykładowy indeks wtórny

Indeks gęsty zakładany jest na każdą wartość występującą w atrybucie kolekcji obiektów. Może to być np. nazwisko lub miasto zamieszkania. W wyniku wyszukiwania może zostać zwrócona jedna lub wiele wartości.

Indeks zakresowy oznacza, że dana pozycja z tabeli indeksowej dotyczy nie jednej konkretnej wartości, ale pewnego zakresu (przedziału) wartości. Mogą to być np. grupy zarobkowe 100-200; 200-300, nazwiska rozpoczynające się od konkretnej litery lub klienci generujący średnio rachunki w określonym zakresie. Indeks taki może okazać się szczególnie przydatny, gdy warunkiem zapytania są właśnie określone przedziały.

Klucz	*Wartość*
A	ref#osoba1
B	ref#osoba2, ref#osoba3, ref#osoba4
C	ref#osoba5

Ilustracja 5. Przykładowy indeks zakresowy

Dodatkowo indeksy można podzielić na wewnętrzny i zewnętrzny.

Indeks wewnętrzny występuje w sytuacji, gdy tabela indeksu jest jednocześnie tabelą danych, a więc kolumna danych niekluczowych zawiera konkretne informacje. Indeks wewnętrzny może być *pogrupowany*, tzn rekordy są uporządkowane względem klucza wyszukiwania.

Indeks zewnętrzny jest to z kolei dodatkowa tabela, niezależna od faktycznych danych przechowywanych w systemie.

Indeks wewnętrzny może być tylko jeden, indeksów zewnętrznych może być wiele.

Najczęściej spotykane struktury danych stosowane w indeksach to:

- indeksy z kodowaniem haszującym (hash coding)
- indeksy oparte o B-drzewa
- indeksy bitmapowe

W *indeksach z kodowaniem haszującym* stosuje się funkcję haszującą do wygenerowania wartości kluczowej będącej adresem danego rekordu w tabeli. Funkcja ta będzie stosowana zarówno do tworzenia indeksu, jak tez przy wyszukiwaniu. Funkcja haszująca za argument przyjmuje wartość atrybutu po którym zamierza się indeksować kolekcję a zwraca wartość wygenerowaną dla podanego argumentu. Zaletą indeksów opartych o funkcje haszujące jest bardzo zbliżony czas wyszukiwania dla dowolnego rozmiaru indeksu. Niestety trzeba z góry określić rozmiar tablicy dla indeksu. (Wada została wyeliminowana przez wprowadzenie algorytmów z dynamicznym haszowaniem- dynamic hashing oraz liniowym haszowaniem- linear hashing). Dodatkowo pojawia się konieczność rozstrzygania kolizji powstałych w wyniku zwracania takiej samej wartości dla różnych argumentów. Aby temu zapobiec należy dobrać odpowiednią funkcję haszującą adekwatną do typu i zbioru danych.

Indeksy oparte na *B-drzewach* wykorzystują strukturę B-drzewa (b-tree) do organizacji rekordów danych w indeksie. Takie indeksy idealnie nadają się do zróżnicowanych danych, mających wiele wartości i mało duplikatów. Istnieje wiele wersji B-drzew oraz ich implementacji. Można jednak przyjąć, *że B-drzewo rzędu m to drzewo, które jest puste albo zawiera k-węzły, z k-1 kluczami i k łączami do drzew reprezentujących każdy z k przedziałów ograniczanych przez klucze, i posiada następujące własności strukturalne: dla korzenia k musi*

zawierać się pomiędzy 2 i m, zaś a dla każdego innego węzła pomiędzy 2/m i m; zaś wszystkie łącza do pustych drzew muszą się znajdować w tej samej odległości od korzenia [S99].
Najistotniejszą cechą b-drzew jest to, że są zbalansowane, co oznacza że wszystkie liście (faktyczne wartości przechowywane w drzewie) są na tej samej głębokości w stosunku do korzenia. Dzięki temu operacja wyszukiwania dla każdej wartości wymaga takiej samej liczby operacji odczytu indeksu. Na skutek operacji wstawiania oraz usuwania drzewo może przestać być zbalansowane. Dlatego wymagana jest operacja ponownego zbalansowania drzewa,

Indeksy bitmapowe, najczęściej stosowane w hurtowniach danych. Zakładane są one na dane, wśród których zbiór wartości jest niewielki. Indeksacja polega na przypisaniu wartości 1 lub 0 dla każdej pozycji ze zbioru wartości mogących wystąpić w atrybucie indeksowanej kolekcji.

Nazwisko	***Stanowisko***	***Indeks***		
		D	***K***	***S***
Kowalski	dyrektor	1	0	0
Nowak	księgowy	0	1	0
Adamski	sprzedawca	0	0	1
Koniecki	dyrektor	1	0	0

Ilustracja 6. Przykładowy indeks bitmapowy

Podczas wyszukiwania najpierw dokonuje się sprawdzenia wartości z tabeli indeksowej i przypisuje 1 lub 0 jeśli wiersz spełnia zadany warunek. Następnie w wyniku operacji bitowych na wektorach powstałych po sprawdzeniu wartości dla wszystkich członów warunku otrzymuje się wektor wskazujący wiersze spełniające zadane wymagania, np.:
stanowisko='dyrektor' OR stanowisko='sprzedawca'
[1001] OR [0010] = [1011]

Na podstawie:
[S04]

Rozdział 4. Mieszanie liniowe

Algorytm mieszania liniowego (linear hashing) jest podstawą dla omawianego w następnym rozdziale algorytmu LH*. Trudno byłoby zrozumieć LH* bez wcześniejszego zaznajomienia się z algorytmem LH, dlatego też opisy algorytmów rozpoczną się od tej pozycji.

4.1. Wstęp

Podstawowymi strukturami danych używanymi do budowy indeksów są drzewa oraz tabele z funkcjami mieszającymi. Indeksy oparte na drzewach mają logarytmiczny czas dostępu i na razie nie będą obiektem naszego zainteresowania. W przypadku tabel indeksowanych funkcjami mieszającymi czas dostępu do elementów jest stały, jednakże muszą być spełnione pewne warunki.

Najprostszym rozwiązaniem dla takiego indeksu jest stworzenie tabeli o wielkości uniwersum kluczy rekordów. Rozwiązanie to gwarantuje stały czas dostępu do elementów indeksu, ale jest oczywiście dalece niepraktyczne, gdyż w rzeczywistych zastosowaniach indeksów uniwersum kluczy jest tak duże, że skutkowało by to indeksami przekraczającymi rozmiary pamięci o znikomym poziomie wypełnienia.

Rozwiązaniem kompromisowym jest stworzenie tabeli o rozsądnym rozmiarze oraz zastosowanie funkcji mieszającej, która przekształca wartości z dziedziny kluczy w wartości z dziedziny indeksów tabeli. Dziedzina kluczy jest w takim przypadku zazwyczaj znacznie większa od dziedziny indeksów tablicy, jest zatem oczywiste, że pewne różne od siebie klucze będą musiały być wstawione w to samo miejsce w tabeli. Takie zjawisko nazywamy *kolizją*. Elementy które są ze sobą w kolizji zazwyczaj ustawiane są w listę. Konsekwencją tego jest pogorszenie się czasu dostępu do elementów w przypadku dużej liczby kolizji (zmierza on do liniowego).

Przed wprowadzeniem mieszania liniowego [L80] (i podobnych rozwiązać) trzeba było zdecydować się na jedno z trzech nie najlepszych rozwiązań:

- Stworzenie tabeli o bezpiecznym rozmiarze – marnotrawstwo pamięci.
- Pogodzenie się ze znacznym pogorszeniem wydajności po przekroczeniu progu optymalnego wypełnienia (ok. 70%).
- Kompletna przebudowa indeksu po przekroczeniu progu optymalnego wypełnienia (bardzo kosztowna obliczeniowo przy dużej ilości elementów).

Natomiast mieszanie liniowe, opisane szczegółowo poniżej, rozwiązuje problem dynamicznego rozszerzania tablicy indeksu bez kosztownej operacji przebudowy, zachowując (w przybliżeniu) stały czas dostępu do elementów i, w odróżnieniu od podobnych metod,

przechowując stan indeksu w zaledwie kilku bajtach pamięci.

4.2. Podstawy mieszania

Mieszanie to technika, która przy pomocy identyfikatora zwanego *kluczem głównym* jest w stanie znaleźć adres rekordu powiązanego z danym kluczem. Klucz (nazwiemy go c) jest zazwyczaj dodatnią liczbą całkowitą, rekord jest natomiast dowolnym obiektem w pamięci. *Funkcją mieszającą h* nazywamy prostą funkcję przyporządkowującą kluczowi c komórkę pamięci identyfikowaną przez wartość $h(c)$.
Przykładem prostej funkcji mieszającej jest dzielenie modulo:
$h(c) = c \bmod M \qquad M = 2, 3, \ldots$
Komórki w pamięci nazywamy *kubełkami*. Kubełki mogą zajmować b rekordów. Rekordy wstawiane są do kubełka $h(c)$, nazywanego *głównym* dla c o ile kubełek nie jest jeszcze pełny. Poszukiwania klucza c zaczynają się zawsze od głównego kubełka.

W momencie przepełnienia kubełka (i wystąpienia *kolizji*) stosuje się jakiś algorytm przyporządkowujący kluczowi c adres $m \neq h(c)$. Kubełek m staje się *kubełkiem przepełnienia* dla c.
Jako, że poszukiwania przepełnionych rekordów wymagają *co najmniej* dwóch dostępów to w przypadku rozwiązywania konfliktów przez tworzenie kubełków przepełnienia czas dostępu gwałtownie spada, kiedy główne kubełki zaczynają się wypełniać.
W przypadku wystąpienia przepełnienia przy wstawianiu klucza c, jeśli w kubełku $h(c)$ nie ma żadnych rekordów, które powinny zostać przeniesione do kubełka przepełnień to kolizję można rozwiązać tylko przez wybór nowej funkcji mieszającej h'. Nowa funkcja powinna przyporządkować nowe adresy dla części rekordów umieszczonych w kubełku $h(c)$ przez funkcję h.
Zmiana funkcji może dotyczyć wszystkich kubełków znajdujących się w tablicy. Mamy wtedy do czynienia z całkowitą przebudową indeksu. Nas bardziej interesuje sytuacja w której zmieniamy funkcję mieszającą tylko dla tego pojedynczego kubełka – taką funkcję nazywamy wtedy *dynamiczną funkcją mieszającą*. Modyfikacja funkcji mieszającej nazywamy *podziałem*, a adres *h(c) adresem podziału*.
Podstawowym pomysłem w algorytmach Mieszania Wirtualnego oraz Mieszania Liniowego jest wykorzystanie podziałów do zapobiegnięcia akumulacji przepełnionych rekordów. Podziały są wynikiem zastosowania *funkcji podziału*.

4.3. Definicja funkcji mieszającej

Dla mieszania liniowego funkcja mieszająca zdefiniowana jest w następujący sposób:

Niech C będzie dziedziną kluczy. Niech

$h_0: C \rightarrow \{0, 1, ..., N-1\}$

będzie funkcją początkową. Funkcje h_1, h_2, ... spełniające następujące warunki:

$h_i: C \rightarrow \{0, 1, ..., 2^i N - 1\}$

oraz dla każdego klucza c:

$h_i(c) = h_{i-1}(c)$

lub:

$h_i(c) = h_{i-1}(c) + 2^{i-1}N$

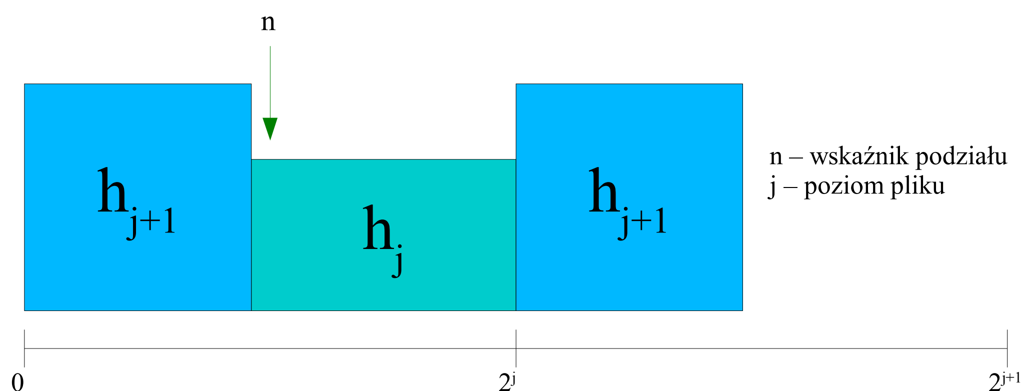
nazywamy funkcjami podziału dla funkcji h_{i-1}.

Przyjmuje się, że funkcje mieszające losowo rozkładają elementy, a prawdopodobieństwo zdarzenia, że klucz c zostanie wrzucony przez h_i do konkretnego kubełka jest bliskie $1/(2^i N)$.

4.4. Dokonywanie podziałów

Podziału pojedynczego kubełka dokonujemy w momencie wystąpienia przepełnienia. Naturalnym pomysłem jest podział kubełka w którym nastąpiło przepełnienie (i takie rozwiązanie było stosowane we wszystkich algorytmach, które poprzedzały mieszanie liniowe). W takiej sytuacji adres podziału jest losowy i wymagana jest pomocnicza struktura danych określająca zastosowane funkcje mieszające dla każdego z kubełków. Liniowe mieszanie radzi sobie z tym problemem dokonując podziałów w z góry określony sposób.

Plik w algorytmie mieszania liniowego scharakteryzowany jest jedynie przez dwa parametry – *poziom* pliku j oraz *wskaźnik podziału* n, nie ma potrzeby przechowywania dodatkowych informacji o zastosowanych funkcjach mieszających. Poziom pliku definiuje bazową funkcje



Ilustracja 7. Struktura pliku w algorytmie LH

mieszająca dla pliku, a wskaźnik podziału wskazuje na kolejny kubełek, który ma zostać podzielony (ilustracja 7).

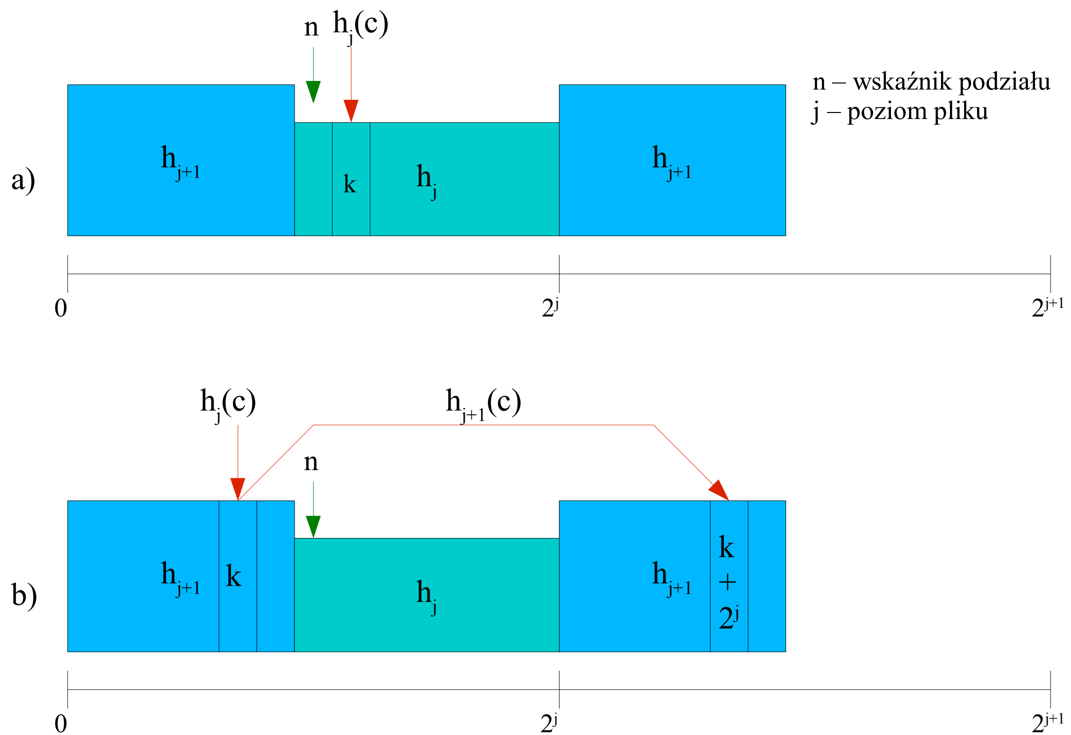
Plik w algorytmie mieszania liniowego jest początkowo na poziomie 0, a wskaźnik podziału wskazuje na pierwszy kubełek (zazwyczaj numer 0). W momencie kolizji adres kubełka do podziału jest wskazywany przez *n*. W trakcie podziału rekordy zostają rozdzielone między kubełki *n* i *n+N* przy pomocy funkcji h_1. Po wykonaniu podziału wskaźnik *n* przesuwany jest na kolejny kubełek, aż do momentu kiedy *n* przekroczy wartość *N-1*. W tym momencie wszystkie *N* kubełków, które początkowo były na poziomie 0 jest już na poziomie 1, a dodatkowo powstało *N* kubełków o numerach *N, ..., 2N-1* także na poziomie 1. W tej sytuacji podnoszony jest poziom pliku, a wskaźnik *n* znowu wskazuje na pierwszy kubełek. Tę procedurę podziałów można powtarzać bez końca, co pozwala na nieskończone rozszerzanie pliku niezależnie od wielkości początkowej.

Przepełniony kubełek nie jest zazwyczaj dzielony w momencie wystąpienia podziału, ze względu na sekwencyjność podziałów i losowość przepełnień. Z faktu sekwencyjnego wykonywania podziałów wynika jednak, że nastąpi jedynie opóźnienie w dokonaniu podziału przepełnionego kubełka. Jeśli elementy są rozkładane w kubełkach w miarę równomiernie przez funkcję mieszającą to przepełnienia kubełków nie powinny być duże.

Opisany mechanizm daje w efekcie przestrzeń adresową, która rozszerza się *liniowo* i ma dowolnie duży rozmiar. Liniowe rozszerzanie (po jednym kubełku na raz) przestrzeni zapewnia niewielki koszt rozszerzania w porównaniu z koniecznością przebudowy całego indeksu.

4.5. Wyliczanie adresu

W momencie wyszukiwania bądź wstawiania klucza *c* najpierw stosowana jest funkcja $h_j(c)$ dająca wartość z przedziału $<0, 2^jN>$. Jeśli wyliczony adres jest większy lub równy od *n* to znaczy, że dany kubełek jest na poziomie *j* i wyliczony adres jest poprawny. Jeśli natomiast wyliczony adres *k* jest mniejszy od *n* to znaczy, że zaadresowany kubełek jest już na poziomie *j+1* i zastosowaną do niego funkcją jest h_{j+1}. W tej sytuacji należy powtórnie wyliczyć adres dla danego klucza ($h_j(c)$), który będzie już na pewno poprawnym adresem. Należy zauważyć, że z własności funkcji mieszającej wynika, że poprawnym adresem będzie albo pierwotnie wyliczony adres *k* albo adres $k+2^j$ (ilustracja 8).



Przy obliczaniu adresu kubełka zawsze zaczyna się obliczanie przy pomocy funkcji h dla bieżącego poziomu pliku (poziom j).

a) jeśli zaadresowany kubełek jest na poziomie j ($h_j(c) \geq n$) to adresowanie jest poprawne

b) jeśli zaadresowany kubełek jest na poziomie j+1 ($h_j(c) < n$) to adresowanie może nie być poprawne i należy obliczyć adres powtórnie funkcją h_{j+1}

Ilustracja 8. Wyliczanie adresu

Algorytm wyliczania adresu *k* jest więc bardzo prosty:

$k := h_j(c)$

if $k < n$ *then*

  $k := h_{j+1}(c)$

4.6. Kontrola wypełnienia oraz zmniejszanie przestrzeni

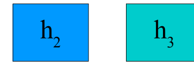
Przedstawiony powyżej sposób uruchamiania podziałów nazywany jest *niekontrolowanymi podziałami*. Możliwe jest alternatywna metoda zwana *kontrolowanymi podziałami*, polega ona na liczeniu bieżącego wypełnienia tabeli i uruchamianiu podziałów po przekroczeniu ustalonego progu. Zastosowanie tego rozwiązania pozwala na uzyskanie wyższego niż w przypadku niekontrolowanych podziałów wskaźnika wypełnienia, choć oczywistą konsekwencją jest zwiększenie liczby przepełnionych kubełków i co za tym idzie pogorszenie czasu dostępu. W przypadku niekontrolowanych podziałów średnie wypełnienie dla kubełków o rozmiarze 1 wynosi ok. 80% i wraz ze zwiększaniem rozmiaru kubełków spada do ok. 60%, w przypadku kontrolowanych podziałów możliwe jest osiągnięcie nawet 90% wypełnienia.
Zastosowanie wskaźnika wypełnienia jest natomiast konieczne w przypadku malejących plików. Złączenia kubełków wykonywane są po przekroczeniu dolnej granicznej wartości wypełnienia pliku. Operacja złączenia jest operacją dokładnie odwrotną do operacji podziału – wskaźnik podziału n jest zmniejszany, a rekordy z kubełka $n+2^iN$ są przenoszone do kubełka n.
Połączenie zastosowania górnych oraz dolnych granic wypełnienia pliku pozwala na uzyskanie w miarę stałego wypełnienia pliku. W praktyce różnica między progiem górnym i dolnym powinna wynosić ok. 10%, w przeciwnym przypadku plik może przechodzić zbyt częste podziały i złączenia.

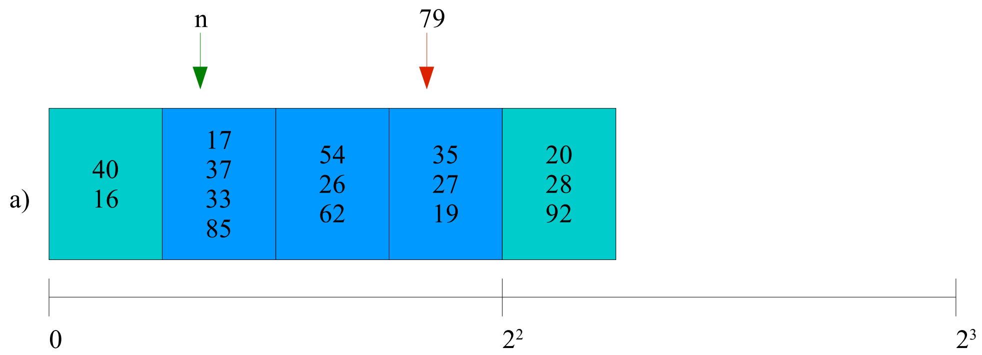
4.7. Przykład działania

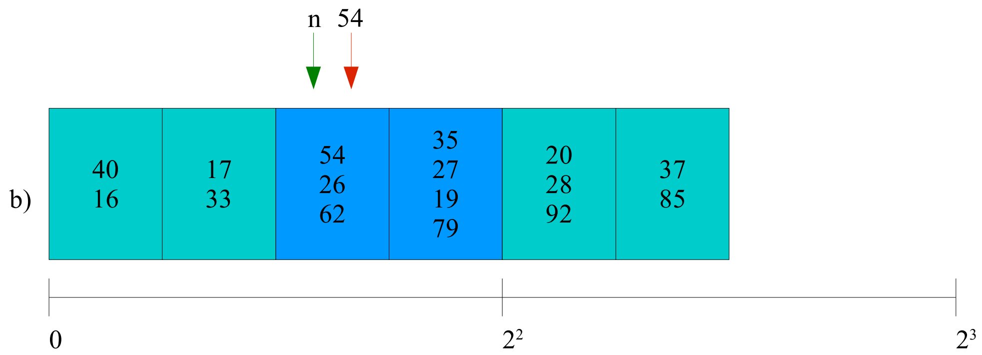
Weźmy przykładowo plik na poziomie 2, ze wskaźnikiem podziału ustawionym na kubełek numer 1 i pojemności kubełka 3 (ilustracja 9a). Funkcje mieszające dla tego pliku mamy zdefiniowane następująco:

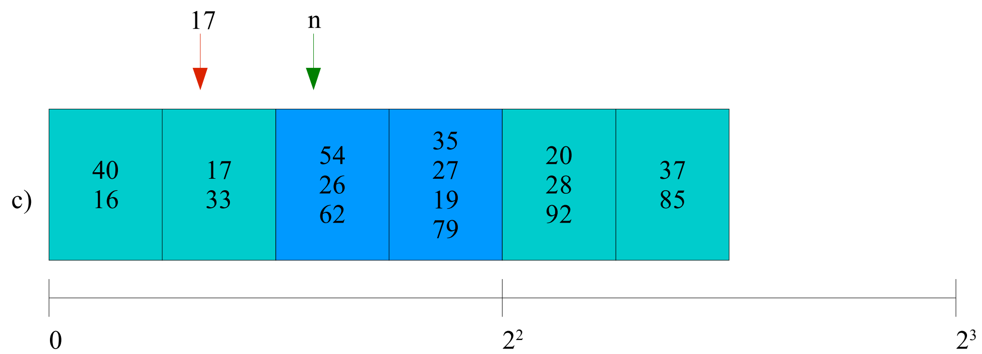
poziom pliku = 2
wskaźnik podziału = 1
pojemność = 3

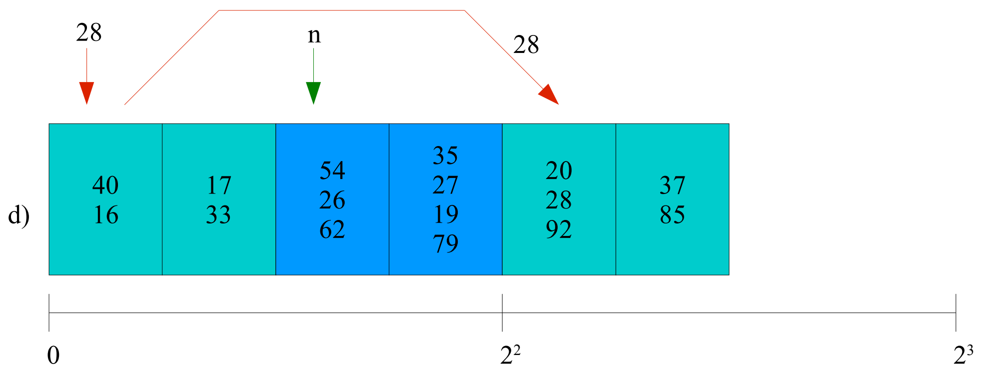
$h_2(c) = c \bmod 4$
$h_3(c) = c \bmod 8$











Ilustracja 9. Operacje wstawiania i wyszukiwania w algorytmie LH

$h_2(c) = c \bmod 4; h_3(c) = c \bmod 8$

Wstawiamy rekord z kluczem 79 do pliku. W pierwszej kolejności korzystamy z funkcji h_2.

$h_2(79) = 79 \bmod 4 = 3$

Wyliczony adres $3 \geq n$, więc jest to poprawny adres. Wstawiamy rekord do kubełka 3. W kubełku 3 są już 3 elementy, więc następuje przepełnienie. Dokonujemy podziału kubełka na który wskazuje n (1), po czym zwiększamy n o 1. Do podziału kubełka wykorzystywana jest funkcja h_3. W wyniku podziału rekordy o kluczach 37 i 85 zostają przeniesione do kubełka 5 ($1 + 2^2$). Wygląd pliku po przeprowadzeniu podziału przedstawiony jest na ilustracji 9b.

Teraz przedstawione zostaną 3 ewentualne rezultaty wyszukiwania. Najpierw wyszukiwany jest rekord o kluczu 54. Jak zwykle na początku stosujemy funkcję mieszającą dla danego poziomu pliku czyli h_2. Jako rezultat dostajemy adres 2 ($\geq n$). Adres 2 jest na poziomie 2 więc jest poprawny i dalsze adresowanie nie jest konieczne (ilustracja 9b).

W drugim przypadku wyszukiwany rekord ma klucz 17. Obliczenie adresu z funkcji h_2 daje adres 1. Kubełek 1 jest na poziomie 3, adres wyliczony przez h_2 niekoniecznie jest poprawny, należy ponowić obliczenie przy pomocy h_3. Także h_3 daje nam adres 1, więc kubełek został poprawnie zaadresowany (ilustracja 9c).

Na koniec wyszukiwany jest rekord o kluczu 28. Obliczenie adresu z funkcji h_2 daje adres 0. Kubełek 0 jest również na poziomie 3, należy ponowić obliczenie przy pomocy h_3. h_3 tym razem daje nam adres 4, więc kubełek nie został poprawnie zaadresowany za pierwszym razem (ilustracja 9d).

Rozdział 5. SDDS LH*

W tym rozdziale zostanie zdefiniowane pojęcie SDDS oraz propozycja struktury SDDS zbudowanej w oparciu o algorytmu LH*.

5.1. Wstęp

W chwili obecnej da się zauważyć trend do zastępowania wielkich wieloprocesorowych systemów o dużej mocy przetwarzania licznymi znacznie uboższymi serwerami połączonymi w określone struktury za pomocą sieci. Zaletą takiego rozwiązania jest przede wszystkim znacznie niższy koszt takiego rozwiązania, a przy tym akceptowalna wydajność.

Rozproszone przetwarzanie należy jednak stosować rozważnie, nie wszystkie problemy da się efektywnie rozwiązać przy zastosowaniu takiej architektury. Ponadto problemem jest ustalenie optymalnej liczby serwerów, które należałoby wykorzystać dla danego zastosowania. Zbyt duże rozproszenie powoduje spadek wydajności (komunikacja przez sieć jest rzędy wielkości wolniejsza od dostępów do pamięci), zbyt małe rozproszenie powoduje przeciążenie poszczególnych węzłów w rozproszonym systemie. Dodatkowym utrudnieniem jest fakt, że dla wielu praktycznych zastosowań trudno przewidzieć obciążenie, które często także zmienia się dynamicznie. Z tych powodów poszukiwany jest system, który jest w stanie dynamicznie się rozszerzać w zależności od aktualnych potrzeb.

Rozważany tutaj problem to system w którym mamy pewną liczbę systemów klienckich (klientów) dzielących plik F. Klienci wstawiają obiekty posługując się kluczami głównymi, szukają oraz usuwają obiekty (także używając wykorzystując klucze). Struktura obiektów jest dowolna i nie jest istotna dla dalszych rozważań. Plik F jest przechowywany na pewnej liczbie serwerów. Zarówno klienci, jak i serwery, są to oddzielne systemy połączone jedynie za pomocą sieci lub też oddzielne procesory systemu wieloprocesorowego posiadające swoją własną pamięć. Każdy serwer dostarcza pewną przestrzeń do składowania pliku *F* zwaną *kubełkiem*. Ilość obiektów wstawianych do pliku jest nieprzewidywalna i może znacznie przekraczać rozmiar kubełka. Aby rozłożyć obiekty w systemie serwery mogą przesyłać je sobie nawzajem. Głównym problemem jest znalezienie efektywnej struktury danych, która będzie spełniać następujące kryteria:

1. Plik rozszerza się na nowe serwery nie powodując długich przestojów, ani dużych obciążeń i tylko wtedy, gdy używane już serwery są już efektywnie zapełnione.
2. Nie ma centralnego serwera przez który musiałyby przechodzić wszystkie obliczenia adresów (np. w celu dostępu do centralnego katalogu).
3. Operacje na plikach (wyszukiwanie, wstawianie, podziały, ...) nie wymagają wykonywania

atomowych aktualizacji u klientów.

Kryterium 2 jest istotne z wielu powodów. Przede wszystkim struktura jest spełniająca jest potencjalnie bardziej efektywna pod względem ilości wiadomości niezbędnych do manipulacji oraz bardziej niezawodna.

Kryterium 3 jest niezbędne w rozproszonym środowisku, gdyż klienci mogą nigdy nie być dostępni wszyscy na raz.

Strukturę, która spełnia wszystkie trzy kryteria, nazywamy *Skalowalną Rozproszoną Strukturą Danych* (Scalable Distributed Data Structure).

Poniżej przedstawiony zostanie algorytm SDDS LH* [LNS96], będący generalizacją algorytmu LH przedstawionego w poprzednim rozdziale i oferujący następujące właściwości:

- plik może osiągać dowolne wielkości, a poziom wypełnienia waha się w granicach 65-95% w zależności od parametrów pliku,
- operacja wstawienia do pliku wymaga zazwyczaj jednej wiadomości, a maksymalnie trzech,
- wyszukiwanie obiektów po kluczu wymaga zazwyczaj dwóch wiadomości, a w najgorszym przypadku czterech,
- równoległe wyszukiwanie na pliku o M kubełkach ma koszt co najwyżej $2M + 1$ wiadomości i między 1 a $O(\log_2 M)$ tur wiadomości.

5.2. Rozszerzanie pliku

Dla uproszczenia opisu algorytmu przyjmuje się, że każdy kubełek umieszczony jest na oddzielnym serwerze (choć w rzeczywistości nie ma takiej konieczności). Każdy kubełek ma zapisany swój poziom w nagłówku. Kubełki mogą być umieszczone w pamięci lub na dysku, a ich organizacja to jedynie kwestia implementacji.

Podziały kubełków uruchamiane są tak samo jak w przypadku algorytmu LH – jeśli w kubełku o pojemności b znajduje się b lub więcej rekordów i nadchodzi żądanie wstawienia rekordu.

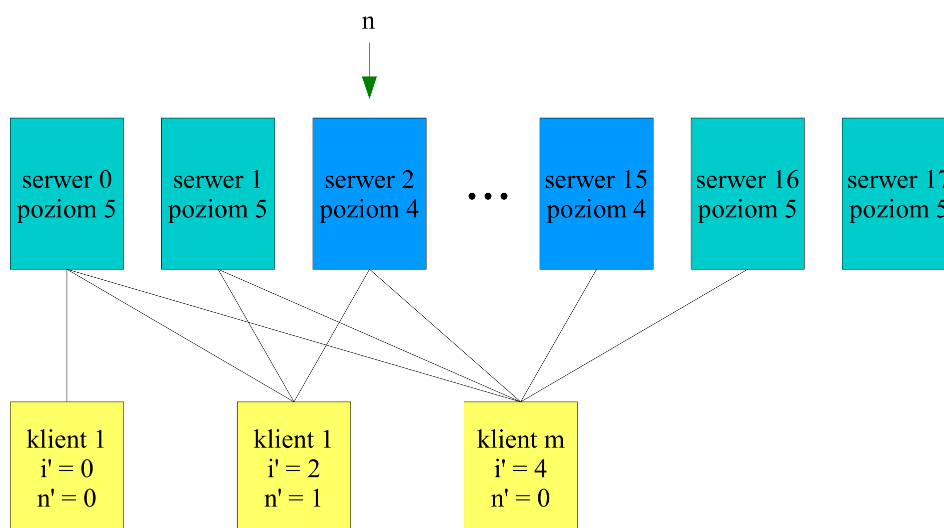
Kubełki ponumerowane są kolejnymi liczbami całkowitymi poczynając od 0. Numery kubełków traktowane będą jak adresy kubełków. W rzeczywistym systemie niezbędna będzie jeszcze translacja z tego logicznego adresu na adres fizyczny (co opisane zostanie dalej).

Sam plik LH* rozszerza się dokładnie tak samo jak plik z algorytmu LH. Na początku istnienia mamy jedynie kubełek 0, wskaźnik podziału $n = 0$ oraz i funkcję mieszającą h_0. W momencie przepełnienia następuje podział kubełka 0, tworzony jest kubełek 1 i używana jest funkcja h_1. Następny podział dzieli znowu kubełek 0 i tworzy kubełek 2, a używaną dla nich funkcją jest h_2, itd. Jeśli zapomnieć o tym, że kubełki w LH* znajdują się na osobnych serwerach to w algorytmie podziałów nie zauważymy żadnej różnicy w stosunku do LH.

5.3. Adresowanie

Operacje na rekordach są wykonywane przez dowolną ilość klientów. Do operacji należą wstawianie, wyszukiwanie, usuwanie i zapytania równoległe (wszystkie przy pomocy klucza głównego). Opisany w poprzednim rozdziale algorytm mieszania liniowego opierał się na tradycyjnym założeniu, że adresowanie zostało wykonane przy pomocy poprawnych wartości poziomu pliku i oraz wskaźnika podziału n. Takie założenie było w pełni uzasadnione dla algorytmu działającego na pojedynczym systemie, biorąc jednak pod uwagę ograniczenia jakie nakłada struktura SDDS, trzeba z niego niestety zrezygnować. Gdyby chcieć je spełnić trzeba by albo wyznaczyć główny serwer, przez który przechodziłyby wszystkie obliczenia adresów, albo wysyłać informacje o bieżącym stanie pliku do wszystkich klientów po każdym podziale lub złączeniu kubełka. Oba rozwiązania są niezgodne z odpowiednio drugim i trzecim kryterium struktury SDDS.

Główną różnicą między algorytmem LH*, a LH jest to, że serwery LH* są przygotowane do błędnie zaadresowanych operacji. Klienci (a także w mniejszym stopniu serwery) mają własne parametry n' oraz i' będące jedynie obrazem wartości n i i – niekoniecznie zgodnym ze stanem faktycznym. Na początku działania parametry te mają zawsze wartości $n' = 0$ i $i' = 0$, ich aktualizacja następuje jedynie po wykonaniu przez klienta operacji na pliku. Można powiedzieć, że każdy z klientów ma swój własny obraz pliku, który może różnić się od prawdziwego pliku oraz od obrazów innych klientów (sytuacja przedstawiona na ilustracji 10).



*Ilustracja 10. Struktura systemu LH**

Klient przy obliczaniu adresu dla operacji może zrobić *błąd adresowania*, to znaczy wysłać zapytanie z kluczem do nieodpowiedniego kubełka. Po otrzymaniu zapytania, każdy z serwerów wykonuje własne obliczenie adresu dla podanego klucza – co jest drugą istotną właściwością

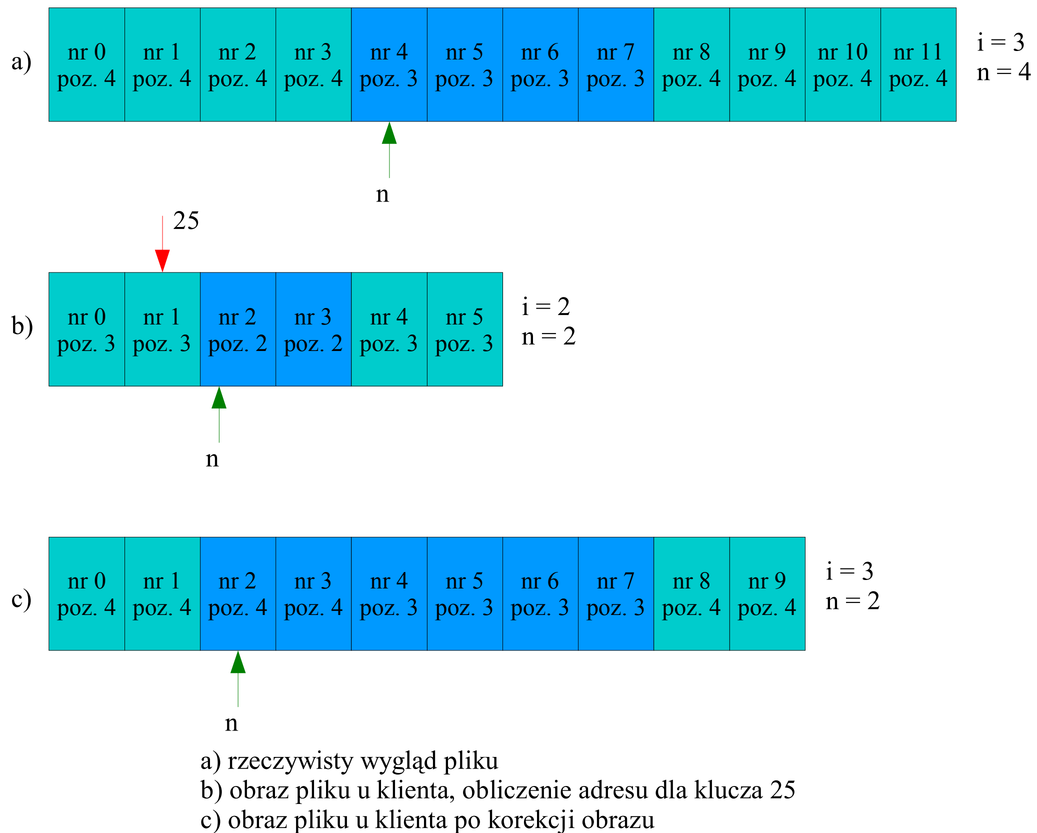
algorytmu LH*. Jeśli obliczony przez serwer adres jest zgodny z zaadresowanym przez klienta kubełkiem operacja jest wykonywana, w przeciwnym przypadku serwer przesyła zapytanie dalej korzystając z własnych obliczeń adresu. Serwer do którego zostało przekazane zapytanie dokonuje kolejnej weryfikacji adresu i w razie potrzeby przekazuje zapytanie do trzeciego serwera. Własności pliku LH* sprawiają, że trzeci serwer na pewno został poprawnie zaadresowany, co zostanie pokazane w dalszej części opisu algorytmu LH*. W przypadku plików LH*, które potrafią zmniejszyć swoją objętość, możliwa jest sytuacja, że zaadresowany kubełek w ogóle nie istnieje. Konieczne jest wtedy skasowanie obrazu pliku do najmniejszego możliwego rozmiaru – to zagadnienie także zostanie omówione w dalszej części opisu.
Ostatnią różnicą między LH, a LH* jest korygowanie obrazu pliku po popełnionym błędzie adresowania. Klient, który popełnił taki błąd dostaje w odpowiedzi *wiadomość korekcji obrazu* (image adjustment message). Taka wiadomość zawiera faktyczny poziom zaadresowanego kubełka. Po jej otrzymaniu klient wykonuje *algorytm korekcyjny klienta* który aktualizuje wartości n' i i', zbliżając tym samym obraz pliku posiadany przez klienta do stanu faktycznego. Zazwyczaj zastosowanie takiego algorytmu powoduje rzadkie występowanie błędów adresowania i niewielką ilość przekazywanych wiadomości między serwerami.

5.4. Obliczanie adresu po stronie klienta

Klient oblicza adres kubełka wykorzystując wzór przedstawiony w algorytmie LH. Podstawione do wzoru wartości wskaźnika podziału n' oraz poziomu pliku i' są wartościami lokalnie przechowywanymi przez klienta. Rezultatem obliczeń jest adres a'.

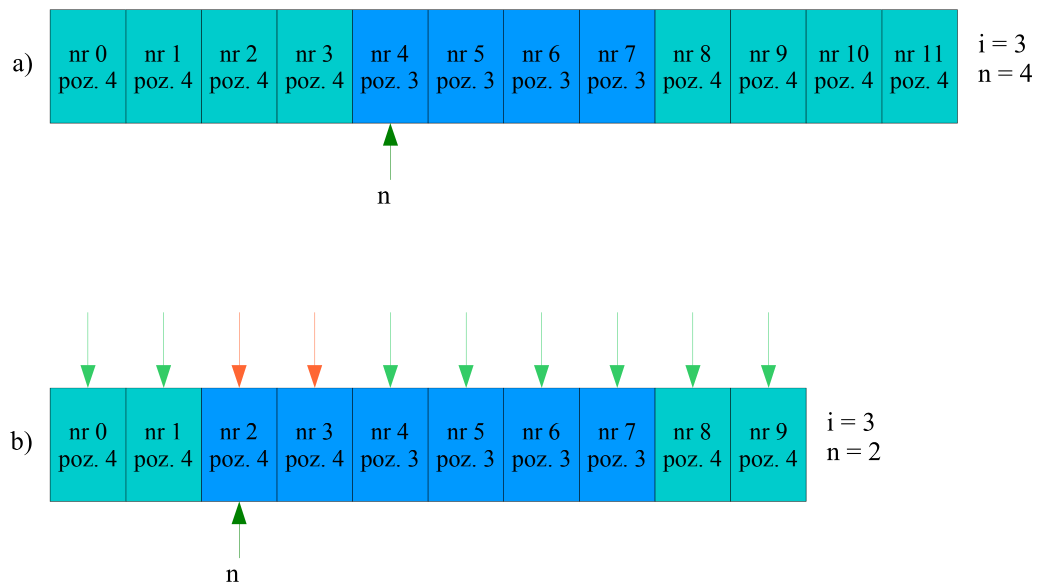
$a' := h_{i'}(C)$
if $a' < n'$
  then $a' := h_{i'+1}(C)$

Jak już wcześniej wspomniano, adres wyliczony z tego wzoru nie musi być prawidłowy, czyli a' może być różne od a wyliczonego przy użyciu bieżących wartości i i n. Istotniejszy jest jednak fakt, że dla wartości i' i n' niewiele odbiegających od faktycznych (w praktyce taka sytuacja ma najczęściej miejsce) większość adresów zostanie obliczona poprawnie! Przykład błędnego adresowania przedstawiono na ilustracji 11. Faktyczny stan wartości n i i to 4 i 3. Jako funkcji mieszającej będziemy używać $h_j(c) = c \bmod 2^j$. Klient próbuje wstawić rekord o kluczu 25 do pliku, według jego obrazu docelowym kubełkiem będzie 1 ($h_2(25) = 25 \bmod 4 = 1$; $h_3(25) = 25 \bmod 8 = 1$). Wysyłając rekord do kubełka 1 klient popełnia błąd adresowania albowiem powinien on trafić do kubełka 9 ($h_3(25) = 25 \bmod 8 = 1$; $h_4(25) = 25 \bmod 16 = 9$), klient dostaje wiadomości korekcyjne zbliżające jego lokalny obraz pliku do rzeczywistego. Po



Ilustracja 11. Obraz pliku u klienta

korekcji obrazu nie jest on wcale zgodny z rzeczywistym, ale pozwala na prawidłowe zaadresowanie kubełka dla klucza 25.



rzeczywisty wygląd pliku
obraz pliku u klienta, poprawnie (kolor zielony) i być może niepoprawnie (kolor pomarańczowy) zaadresowane kubełki

Ilustracja 12. Poprawne i niepoprawne adresowanie kubełków

Na ilustracji 12 widać, że niewielkie różnice między obrazem pliku u klienta, a faktycznym stanem powodują niewiele błędów adresowania. Jeśli wyliczony adres będzie odwoływał się do kubełka oznaczonego strzałką zieloną, adresowanie jest na pewno poprawne. W przypadku strzałki pomarańczowej może wystąpić błąd adresowania, ale wcale nie musi. Jeśli funkcja mieszająca spełnia podane wcześniej założenia i równomiernie rozkłada klucze między kubełki to prawdopodobieństwo wystąpienia błędu dla pomarańczowych kubełków wynosi ½. Takie właściwości plików LH* są bardzo istotne z praktycznego punktu widzenia. Mało aktywni klienci mogą mieć co prawda obraz mocno odbiegający od stanu faktycznego i prawdopodobieństwo popełnienia przez nich błędu adresowania jest stosunkowo duże, ale za to aktywni klienci będą mieli obraz niewiele odbiegający od rzeczywistości i będą popełniać niewiele błędów adresowania.

5.5. Obliczanie adresu po stronie serwera

Dla uproszczenia opisu procedury obliczania adresu przez serwer przyjęte zostanie założenie, że w pliku nie następują złączenia kubełków, a w osobnym podrozdziale umieszczony zostanie opis modyfikacji procedury obliczania adresu dla przypadku kurczących się plików.

Przy powyższych założeniach adres a', obliczony przez klienta, nie może wyjść poza przestrzeń adresową pliku. Oznacza to, że każde wysłane zapytanie trafi do serwera na którym znajduje się kubełek z adresowanego pliku, choć oczywiście może to być kubełek nieprawidłowy.

Do sprawdzenia czy dany kubełek jest faktycznie prawidłowy dla podanego klucza potrzebna jest informacja o poziomie j danego kubełka, przechowywana w nagłówku. Poziom j może być równy poziomowi pliku i lub być już na następnym poziomie.

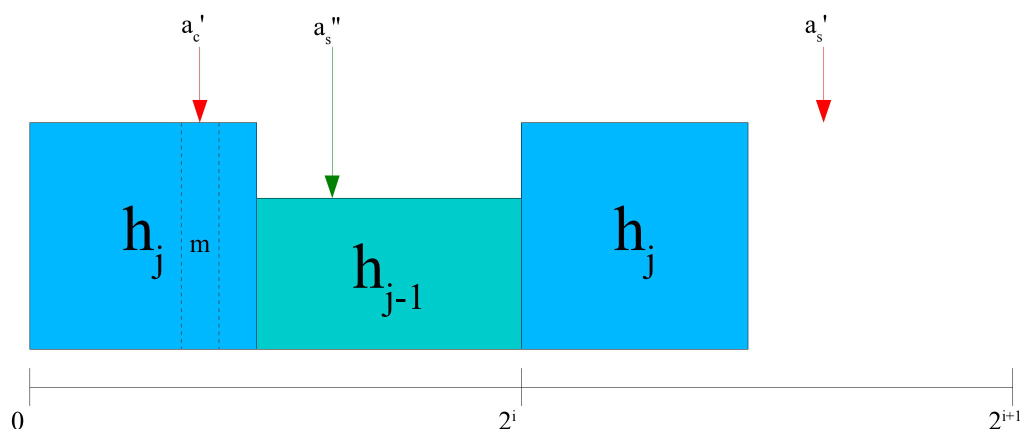
W plikach LH* wartość wskaźnika podziału n nie jest znana serwerom, więc nie mogą skorzystać ze wzoru wykorzystywanego przez klientów, do obliczeń używana jest jego poniższa modyfikacja (c – klucz rekordu, a_c' – adres obliczony przez klienta, a_s' i a_s'' – obliczenia adresu wykonane przez serwer):

$a_s' := h_j(c)$

if $a_c' \neq a_s'$ *then*

    $a_s'' := h_{j-1}(c)$

    if $a_s'' > a_c'$ *and* $a_s'' < a_s'$ *then* $a_s' := a_s''$



Ilustracja 13. Groźba przesłania zapytania za koniec pliku

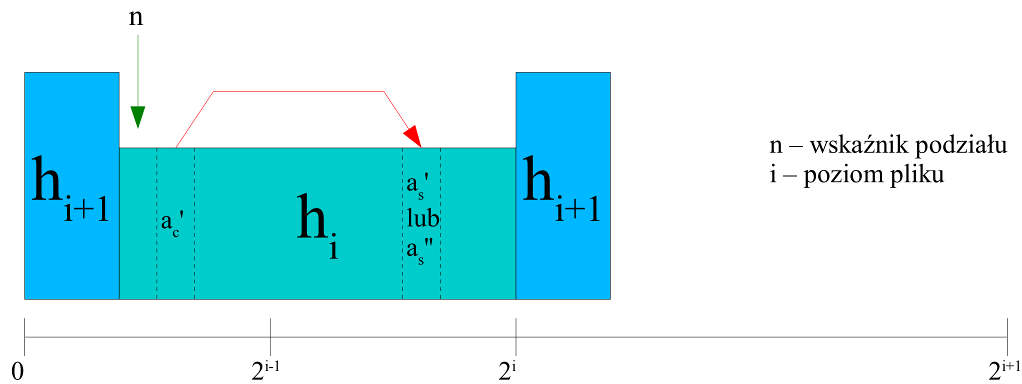
Jeśli a_s' równe jest a_c' oznacza to, że kubełek został prawidłowo zaadresowany i serwer wykonuje zapytanie. W przeciwnym razie zapytanie jest przesyłane dalej do serwera z kubełkiem a_s'. Serwer a_s' dokonuje ponownego obliczenia adresu i, ewentualnie, po raz ostatni przesyła zapytanie do kolejnego serwera. Powyższy wzór wykorzystuje fakt, że elementy z poszczególnych kubełków mogą zostać przeniesione w skutek rozszerzania pliku jedynie do kubełków o wyższych numerach. Należy zauważyć, że pierwsze wyliczenie adresu za pomocą funkcji $h_j(c)$ może dać wynik będący za końcem pliku (ilustracja 13). Aby temu zapobiec adres wyliczany jest powtórnie przy pomocy funkcji $h_{j-1}(c)$, ten wynik na pewno mieści się w pliku, ale może być tak samo nieaktualny jak adres podany przez klienta, stąd porównanie $a_s'' > a_c'$ i ewentualny wybór bezpiecznego rozwiązania.

W odniesieniu do algorytmu LH* udowodniono następującą własność:

Algorytm obliczania adresu znajduje adres dla każdego klucza c, a zapytanie jest przekazywane dalej co najwyżej dwukrotnie.

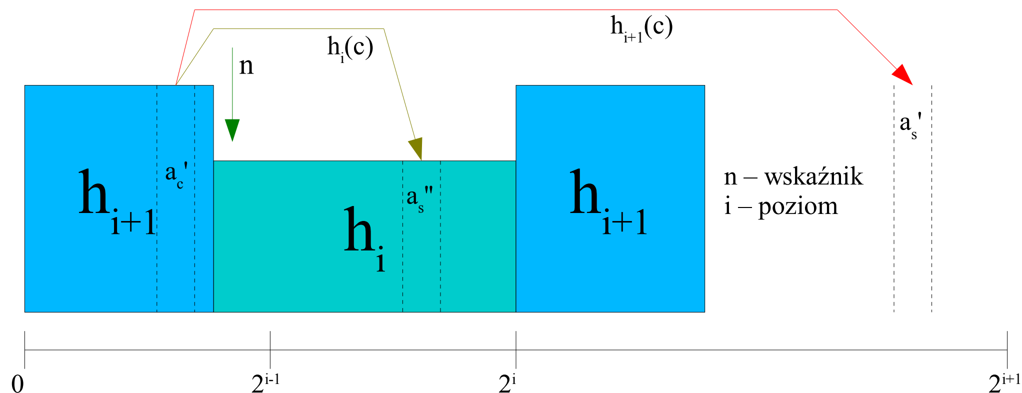
Dowód przeprowadzony jest następująco: Niech i i n będą rzeczywistym poziomem pliku oraz wskaźnikiem podziału, a a_c' niech będzie adresem kubełka na poziomie j otrzymującego zapytanie z kluczem c od klienta. Jeśli $a_c' = a_s' = h_j(c)$ to a_c' jest poprawnym adresem dla c, a algorytm kończy działanie bez dalszego przesyłania zapytania. W przeciwnym razie wyliczamy $a_s'' := h_{j-1}(c)$. W tym momencie może zajść jedna z sytuacji: albo $n \leq a_c' < 2^i$ albo $a_c' < n$ lub $a_c' \geq 2^i$.

W pierwszym przypadku zaadresowany przez klienta kubełek jest na poziomie pliku – $i = j$ (ilustracja), a $a_s' := h_i(c)$, $a_s'' := h_{i-1}(c)$. Jeśli $a_s'' = a_c'$ to a_s' jest poprawnym adresem dla c (bierze się to z własności funkcji podziałów – jeśli na poziomie $j - 1 = i - 1$ klucz c był w kubełka a_c' to na bieżącym poziomie musi być w kubełku $a_s' := h_j(c)$). Jeśli $a_s'' \neq a_c'$ oznacza to, że $a_s'' > a_c'$, a z tego wniosek, że i' w obrazie klienta różni się co najmniej o 2 od j. Zapytanie przekazywane jest do serwera a_s'', poziom tego serwera jest równy poziomowi pierwszego (j),



Ilustracja 14. Przekazywanie zapytania 1

co można wywnioskować z ilustracji. Albo jest to poprawny serwer albo następuje przekazanie zapytania do serwera a_s' (wartości a_s', a_s'' są takie same w wyliczeniach obu serwerów, gdyż znajdują sie one na tym samym poziomie, a że dla drugiego serwera $a_c' = a_s''$ to wybierany jest adres a_s'). Adres a_s' został wyliczony przy pomocy funkcji $h_j(c)$, a $j = i$ jak założono w tym przypadku, więc a_s' jest na pewno właściwym adresem dla klucza c. Wniosek – w pierwszym przypadku mamy co najwyżej dwa przekazania zapytania.

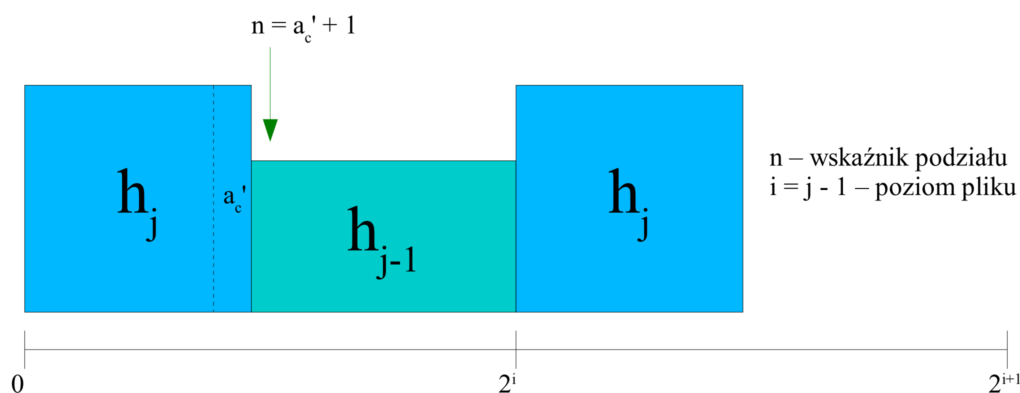


Ilustracja 15. Przekazywanie zapytania 2

W przypadku drugim poziom zaadresowanego przez klienta kubełka wynosi $j = i + 1$ (ilustracja 15), jeśli więc nie ma złączeń kubełków to $a_s'' \geq a_c'$. Jeśli $a_s'' = a_c'$ to zapytanie przekazywane jest do serwera a_s' – poziom tego kubełka także wynosi $j = i + 1$, adresowanie jest więc poprawne gdyż adres został wyliczony funkcją $h_{i+1}(c)$. W tej sytuacji wystąpiło tylko jedno przekazanie zapytanie. Natomiast jeśli $a_s'' > a_c'$ to zapytanie przekazywane jest do serwera a_s'', poziom tego kubełka może być równy $j = i$ lub $j = i + 1$. W drugim przypadku jest to właściwy adres dla klucza i wystąpiło tylko jedno przekazanie zapytania. Jeżeli $j = i$ to albo $a_s'' = a_s'$ i jest to właściwy kubełek albo zapytanie przekazywany jest do serwera a_s', który jest właściwym adresem dla klucza c. W ostatniej sytuacji występują dwa przekazywania zapytania, ale nie więcej. Udowodniono więc, że niezależnie od sytuacji występują co najwyżej dwa przezywania zapytania.

5.6. Korekcja obrazu klienta

W przypadku popełnienia błędu adresowania przez klienta, jeden z serwerów biorących udział w przesyłaniu zapytania wysyła do klienta wiadomość korekcyjną zawierającą poziom j kubełka a_c', do którego klient pierwotnie wysłał zapytanie. Po otrzymaniu takiej wiadomości klient aktualizuje swoje lokalne wartości i' i n'. Celem tego zabiegu jest zbliżenie obrazu pliku klienta do jego faktycznego stanu i co za tym idzie znaczne zmniejszenie prawdopodobieństwa popełnienia kolejnego błędu adresowania, jak to pokazano we wcześniejszej części rozdziału. Algorytm aktualizacji parametrów polega na przyjęciu maksymalnych wartości i' i n', które nie spowodują adresowania poza końcem pliku (zakładając ponownie, że nie występują złączenia kubełków). Wykorzystywana jest tu własność, że poziom j kubełka a_c' można przetłumaczyć na pewien (niezbyt odległy od rzeczywistego) obraz pliku, czyli można wyliczyć maksymalne bezpieczne wartości i' oraz n'. Ze struktury pliku LH wynika, że jeżeli na serwerze umieszczony jest kubełek m o poziomie j to takimi bezpiecznymi wartościami będą $i' = j - 1$ i $n' = a_c' + 1$, gdyż właśnie taki poziom miał plik po utworzeniu kubełka a_c' (ilustracja 16). Kompletny wzór korekcji obrazu ma następującą postać:



Ilustracja 16. Wyliczenie obrazu pliku z poziomu zaadresowanego kubełka

$i' := j - 1, n' := a_c' + 1;$

$if\ (n' \geq 2^{i'})\ then\ n' := 0,\ i' := i' + 1;$

Pierwsza linia wzoru została już wytłumaczona, a druga bierze jedynie pod uwagę fakt, że a_c' mógł być przed podziałem ostatnim kubełkiem na poziomie j – 1. Po dokonaniu podziału wszystkie kubełki w pliku znalazły się na poziomie j, a więc poziom pliku został zwiększony o 1, a wskaźnik podziału wrócił na początek pliku.

Obliczony z tego wzoru obraz może być oczywiście dalej niezgodny z rzeczywistością, ale z każdą wiadomością korekcyjną zbliża się on do faktycznego wyglądu pliku. Jeśli rozważyć

sytuację w której plik przestaje się rozszerzać (nie wykonywane są operacje wstawiania) to klienci z czasem osiągną obrazy w pełni zgodne z rzeczywistym stanem pliku i nie będą popełniać żadnych błędów adresowania.

5.7. Dokonywanie podziałów

Jak już wcześniej powiedziano plik LH* rozszerza się w taki sam sposób jak plik LH poprzez liniowe przesuwanie wskaźnika podziału. Tak samo jak w przypadku algorytmu LH zachodzi potrzeba przechowywania aktualnych wartości i i n w jednym miejscu. Najłatwiej zapewnić taką sytuację wyróżniając jeden z serwerów i przydzielając mu funkcję *koordynatora podziałów* (split coordinator). Tak samo jak w przypadku LH podziału w pliku mogą być kontrolowane lub niekontrolowane.

5.7.1. Podziały niekontrolowane

Dla podziałów niekontrolowanych koordynator podziałów otrzymuje wiadomość od każdego serwera, na którym wystąpiła kolizja. Koordynator wysyła następnie wiadomość uruchamiającą wykonanie podziału do serwera wskazywanego przez parametr n. Po wykonaniu podziału obliczane są nowe wartości i i n:

n := n + 1;
if (n ≥ 2^i) then n := 0, i := i + 1;

Serwer n (z kubełkiem na poziomie j), który otrzymuje wiadomość uruchamiającą podział, wykonuje następujące działania:

a) tworzy kubełek $n + 2^j$ o poziomie $j + 1$ na odpowiednim serwerze,
b) dokonuje podziału kubełka n wykorzystując funkcję h_{j+1} (odfiltrowane obiekty wysyłane są do serwer $n + 2^j$),
c) zwiększa poziom podzielonego kubełka $j := j + 1$,
d) na koniec potwierdza wykonanie podziału do koordynatora.

Ostatni krok (potwierdzenie) pozwala koordynatorowi na dokonywanie podziałów w sposób asynchroniczny. Jest to istotne ze względu na widoczność powstających kubełków i pozwala uniknąć sytuacji w której klient może odwołać sie do jeszcze niepowstałego kubełka.

5.7.2. Podziały kontrolowane

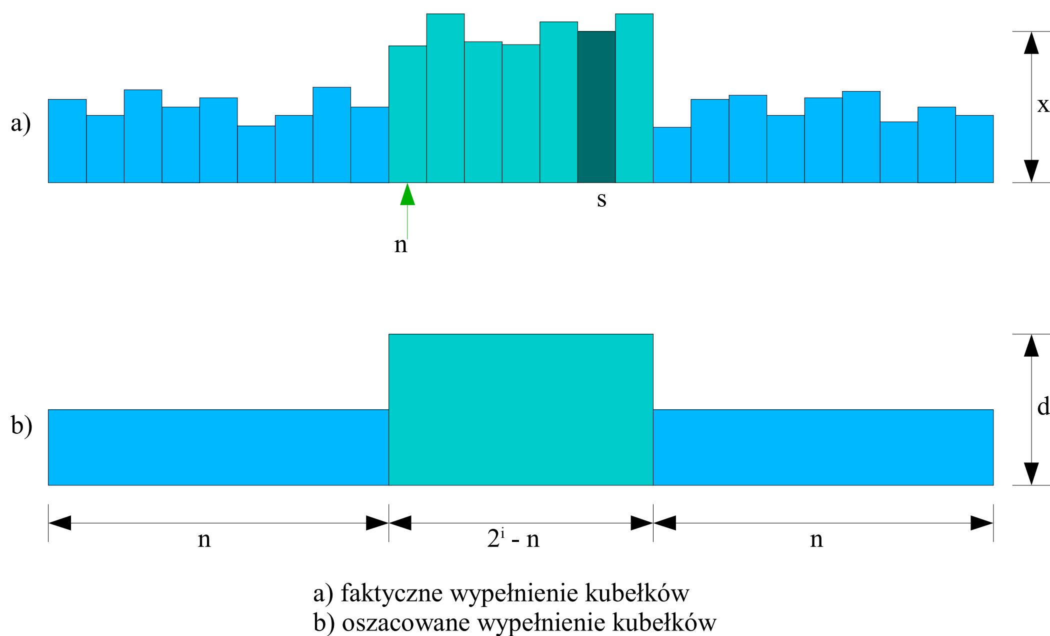
W plikach LH* kontrolowanie podziałów nie jest tak proste jak w plikach LH. Gdyby chcieć

zastosować algorytm przedstawiony w poprzednim rozdziale konieczne byłoby wysyłanie wiadomości do koordynatora z bieżącym wypełnieniem kubełka po każdej operacji wstawienia lub usunięcia rekordu. Spowodowałoby to powstanie wąskiego gardła na serwerze koordynatora, co jest niezgodne z założeniami struktury SDDS. Akceptowalny algorytm kontroli podziałów powinien generować niewielką ilość dodatkowych wiadomości (a najlepiej wcale). Przykrą konsekwencją tego założenia jest niemożność osiągnięcia stabilności wypełnienia pliku LH* porównywalnej z plikami LH. Przykładowy algorytm przedstawiony poniżej pozwala jednak efektywnie kontrolować wypełnienie kubełków nie wymagając przy tym żadnych dodatkowych wiadomości.

Metoda kontroli wypełnienia dla plików LH.*

Niech kubełek s będzie kubełkiem w którym wystąpiła kolizja, b niech będzie pojemnością kubełka (taką samą dla wszystkich kubełków w pliku), x liczbą rekordów w s, a parametr d wyliczamy z wzoru $d = x/b$. Próg wypełnienia przechowywany na serwerze koordynatora oznaczono t. Algorytm ma następujący przebieg:

a) Kubełek s wysyła informację o kolizji do koordynatora, w której zawarte jest bieżące wypełnienie kubełka x,
b) Koordynator wylicza d według wzoru $d = x/b$, następnie jeśli $s < n$ lub $s \geq 2^i$ to d jest podwajane $d := 2*d$,
c) Koordynator wylicza szacunkowe wypełnienie według wzoru: $\alpha' := (2^i * d)/(2^i + n)$,
d) Jeśli $\alpha' > t$ to koordynator daje sygnał do podziału kubełka n, w przeciwny przypadku podział nie następuje.



Ilustracja 17. Wypełnienie kubełków

Algorytm wyliczania szacunkowego wypełnienia opiera się na dwóch założeniach, które postawiliśmy funkcji mieszającej. Po pierwsze funkcja mieszająca rozkłada elementy równomiernie między dostępną przestrzeń kubełków. Po drugie, przy podziale kubełka funkcją mieszającą z następnego poziomu, średnio połowa elementów powinna trafić do nowo utworzonego kubełka. Z tych założeń można wyciągnąć następujące wnioski: wszystkie kubełki z tego samego poziomu są tak samo wypełnione oraz wypełnienie kubełków na poziomie $i + 1$ jest równe połowie wypełnienia kubełków na poziomie i (przedstawione na ilustracji 17). W kroku b obliczane jest wypełnienie kubełków na poziomie i, stąd warunek $s < n$ lub $s \geq 2^i$. Jeśli jest on spełniony to oznacza, że przesłane wypełnienie dotyczy kubełka na poziomie $i + 1$, a więc wypełnienie dla kubełków na poziomie i będzie dwa razy większe. W kroku d) liczone jest średnie wypełnienie kubełka. Podany wyżej wzór jest nieco uproszczony, a bierze się on z następujących obliczeń: całkowita liczba rekordów w pliku to $2n * d/2 = nd$ (dla kubełków na poziomie $i + 1$) plus $(2^i - n) * d = 2^i * d - nd$ (dla kubełków na poziomie i), po zsumowaniu $2^i * d$; podzielona przez całkowitą liczbę kubełków – $2^i + n$ co razem daje średnie wypełnienie kubełka. W kroku d) sprawdzany jest warunek pozwalający na utrzymanie wypełnienia w zadanych granicach.

5.8. Kurczenie się plików

Jak do tej pory zakładane było, że pliki nie zwężają się wskutek wykonywanych na nich operacji. Było to wygodne założenie, gdyż pozwalało przedstawić zwięźle algorytmy adresowania bez brania pod uwagę komplikacji związanych z adresowaniem poza końcem pliku. Przedstawiona w poprzednim podrozdziale metoda kontroli wypełnienia nie jest jednak pełna bez części pozwalającej na złączenie kubełków w momencie kiedy poziom wypełnienia spadnie poniżej określonego progu. Realizacja złączania jest prosta – koordynator korzysta z tych samych wyliczeń oszacowanego wypełnienia co dla podziałów kubełka. Istotne jest tutaj pytanie, w którym momencie serwer powinien wysłać wiadomość ze swoim poziomem wypełnienia, która spowodowałaby oszacowanie wypełnienia przez koordynatora i ewentualną inicjację złączenia kubełków. Algorytm LH* zdawkowo traktuje problem kurczących się plików i w ogóle nie dostrzega tego problemu. Jeśli rozłożenie elementów jest mniej więcej losowe i plik na jakimś etapie swojego istnienia zaczyna się zmniejszać to kolizje w ogóle nie powinny wystąpić, a więc nie wystąpi także wyliczenie bieżącego wypełnienia, a w efekcie poziom wypełnienia indeksu może spadać do zera. Logiczne byłoby zastosowanie w tym momencie dolnego progu wypełnienia dla kubełka, którego przekroczenie powodowałoby podobną reakcje do wystąpienia kolizji. Wysokość progu powinna być wyliczona w momencie tworzenia pliku na podstawie podanej dolnej granicy wypełnienia i przechowywana w nagłówku każdego z kubełków.

Wspomniany wcześniej problem adresowania poza końcem pliku, polega na tym, że klient może posiadać obraz pliku odbiegający od rzeczywistego w drugą stronę. W konsekwencji zapytanie może trafić do serwera na którym, po wykonaniu złączenia, nie ma już żadnego kubełka z adresowanego pliku. Taki serwer, w przeciwieństwie do błędnie zaadresowanego serwera posiadającego kubełki, nie ma najmniejszego pojęcia co do bieżącego stanu pliku – nie ma więc możliwości sensownego przekazania zapytania.
Najprostszym rozwiązaniem tego problemu jest zresetowanie obrazu klienta do wartości $i' = 0$, $n' = 0$. Dalsze adresowanie począwszy od tego momentu przebiega według opisanego wyżej algorytmu i nie mogą pojawić się dalsze komplikacje.
Jeśli obsługiwane pliki mają tendencję do niewielkich zmian rozmiarów to lepsze rezultaty (pod względem ilości przesłań zapytania) może dać zmniejszenie wartości i' o 1 zamiast jej zupełne kasowanie.

5.9. Umiejscawianie kubełków

Adresy kubełków używane we wszystkich rozważaniach do tej pory były równoznaczne z numerami kubełków. Oczywiste jest, że do skutecznego zaadresowania kubełka w rozproszonym środowisku sieciowym niezbędna jest metoda translacji pomiędzy tym logicznym adresem, a adresem fizycznym. Algorytm LH* proponuje dwa rozwiązania, które pozwalają uniknąć zastosowania serwera nazw:

a) lista serwerów jest statycznie określana w momencie utworzenia pliku i nie może się zmienić,
b) lista serwerów jest określona w dynamicznej tabeli aktualizowanej razem z rozwojem pliku.

Najprostszym sposobem implementacji pierwszego rozwiązania jest stworzenie plików konfiguracyjnych dla wszystkich serwerów i klientów określających fizyczne adresy dla numerów kubełków. Plik wielkości kilku kB jest w stanie zaadresować kilkaset komputerów. Rozwiązanie to ma dwie podstawowe wady: trzeba nałożyć górną granicę na rozmiar pliku, przez określenie maksymalnej liczby kubełków oraz pracochłonne tworzenie i modyfikacja plików konfiguracyjnych w przypadku zmian w środowisku.
Druga propozycja jest w swojej naturze znacznie lepiej dostosowana do algorytmu LH*, pozwala nie tylko na dynamiczną alokację kubełków w miarę potrzeb, ale dodatkowo daje możliwości sensowniejszego rozłożenia kubełków (np. kierując się bieżącym obciążeniem serwerów). Przykładowe rozwiązanie oparte na tym pomyśle wygląda następująco:
Serwer na którym zainicjowano podział kubełka tworzy nowy kubełek na wybranym przez siebie serwerze, a jego adres wysyła do koordynatora. W momencie popełnienia błędu

adresowania przez klienta, zwraca się on do koordynatora podając znane sobie wartości i' i n', a koordynator przesyła mu wszystkie nie znane dotychczas adresy kubełków.
Koordynator nie powinien stać się wąskim gardłem w systemie, gdyż liczba popełnianych błędów adresowania nigdy nie jest duża.
Możliwe jest także alternatywne rozwiązanie, w którym między koordynatorem, a klientami nie występuje prawie żadna komunikacja. Klienci zamiast zwracać się z zapytaniami o adresy do koordynatora mogą otrzymywać brakujące adresy od serwerów. Serwer, który wykonuje podział kubełka, przesyła adres nowego kubełka do koordynatora, a koordynator przesyła mu adresy wszystkich kubełków które zostały utworzone od czasu jego utworzenia. Dzielony kubełek przesyła wszystkie znane sobie adresy do nowego kubełka. Dodatkowo przy każdej wiadomości korekcyjnej przesyłana jest cała lista znanych kubełkowi adresów. Takie rozwiązanie niesie ze sobą zwiększone zapotrzebowanie na pamięć oraz dłuższe wiadomości korekcyjne. Można co prawda przesyłać obrazy różnicowe jeśli porównane zostaną wartości i' i n' i wysyłać tylko adresy kubełków utworzonych w międzyczasie. Taka modyfikacja może spowodować posiadanie błędnych adresów przez klientów, jeżeli w pliku nastąpiło złączenie jednego z kubełków, a następnie po wykonaniu podziału kubełek o tym samym numerze powstał na innym serwerze.

5.10. Zapytania równoległe

Zapytania równoległe dla plików LH* polegają na równoczesnym wysłaniu zapytania Q do wybranych lub wszystkich kubełków w pliku, a wykonanie każdego z zapytań jest niezależne od innych. Zapytania równoległe można wykonać przy pomocy wiadomości typu Broadcast jak i Point-to-Point. W pierwszym przypadku zapytanie wysyłane jest do wszystkich serwerów w strukturze SDDS i jest przez nie przetwarzana. W drugim natomiast klient wysyła zapytanie do wszystkich znanych sobie serwerów pliku, a do zapytania dołączane są posiadane przez niego wartości i' i n'. Serwer oprócz wykonania zapytania przesyła je dalej do swoich potomków, którzy znajdują się poza zakresem adresowania klienta.

5.10.1. Propagacja zapytań równoległych

Klient Q chcąc wykonać zapytanie równoległe wysyła wiadomości do wszystkich znanych sobie kubełków. Do każdej wysłanej wiadomości przypisany jest tzw. poziom wiadomości j', będący poziomem adresowanego kubełka w obrazie klienta. Biorąc pod uwagę różnice między j', a faktycznym poziomem zaadresowanego kubełka a, a wysyła zapytanie także do swoich potomków, którzy znajdują się poza obrazem klienta.

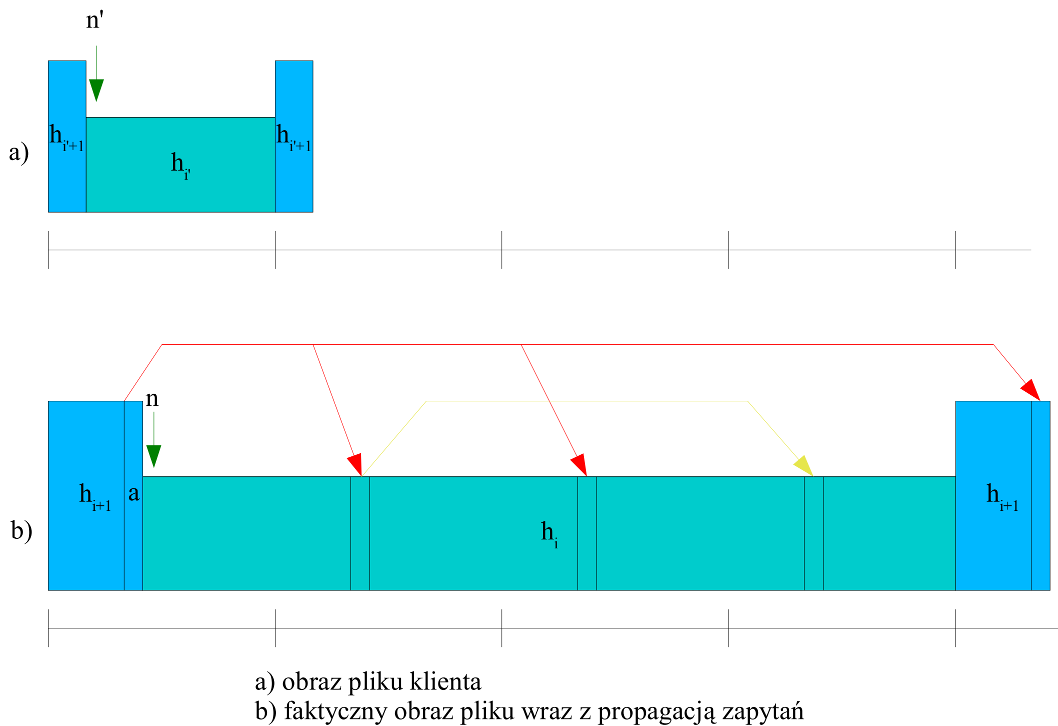
while $j' < j$ *do:*

j' := j' + 1

przekaż (Q, j') do kubełka a + $2^{j'-1}$*;*

end;

W ten sposób zapytanie otrzymują wszystkie kubełki, które powinny otrzymać zapytanie (ilustracja 18).



a) obraz pliku klienta
b) faktyczny obraz pliku wraz z propagacją zapytań

Ilustracja 18. Propagacja zapytań

5.10.2. Korekcja obrazu

Tak samo jak zwykłe zapytanie, zapytanie równoległe może być wykorzystane do korekcji obrazu klienta. Można tego dokonać bez wymiany dodatkowych wiadomości – wystarczy, że w odpowiedziach przesyłanych przez serwery zawarte będą informacje o bieżącym poziomie odpowiadającego kubełka.

Niech j' oznacza najwyższy poziom odpowiadającego kubełka, niech a' będzie najwyższym numerem kubełka zwracającego j', takim że $a' < 2^{j'-1}$, a a''' najwyższym numerem kubełka, takim że $a''' \geq 2^{j'-1}$. Należy wziąć pod uwagę, że zarówno zbiór a' jak i a''' mogą być puste, przy takich założeniach poniższy algorytm przeprowadza najbardziej trafną korekcję obrazu pod warunkiem, że odpowie przynajmniej jeden kubełek.

Algorytm korekcji obrazu wygląda następująco:

i' := j' – 1

if (a' i a'' są niepuste) then n' := max(a' + 1, a''' + 1 – $2^{i'}$*);*

else if (a' jest niepuste) then n' := a' + 1;

else if (a'' jest niepuste) then n' := a''' + 1 – $2^{i'}$*;*

if (n' ≥ $2^{i'}$*) then n' := 0; i' := i' + 1;*

W pierwszej linii algorytmu trzeba przyjąć, że poziom *j'* jest równy *i + 1*, gdyby przyjęto, że *j' = i* to prawdopodobnie spowodowało by to stworzenie obrazu pliku, który byłby większy niż faktyczny. Pierwszy (potrójny warunek if) wykorzystuje fakty znane ze struktury pliku LH – zwiększenie wskaźnika podziału *n* o 1 powoduje pojawienie się dwóch kubełków na poziomie *i + 1*, kubełka *n* oraz kubełka na końcu pliku. Stąd aktualną wartość *n* można odczytać zarówno z ilości kubełków o poziomie *i + 1* na początku jak i na końcu pliku, w algorytmie wybiera się większą (a co za tym idzie bardziej aktualną) wartość *n*. Możliwe jest, że nie przyjdzie żadna odpowiedź od kubełka na poziomie *i + 1*, wtedy *i'* będzie równe *i – 1*, a wyznaczona wartość *n* może być większa niż maksymalna ilość kubełków którą może zawierać plik o poziomie *i'*. Należy wtedy zwiększyć poziom *i'* o 1, a wskaźnik podziału ustawić na początek pliku – ostatni krok algorytmu.

5.11. Wydajność

W pracy przedstawiającej algorytm LH* przedstawiony następujące wyniki symulacji działania dla wstawiania 1 miliona rekordów oraz wyszukiwania 1000 ([LNS96]):

Pojemność kubełka	***Wydajność tworzenia pliku***		***Ilość wiadomości (wyszukiwanie)***	***Ilość utworzonych kubełków***
	Błędy adresowania	***Ilość wiadomości***		
50	1623	2,133	2,001	32791
250	1010	2,033	2,008	8070
500	771	2,017	2,008	4036
1000	558	2,009	2,008	2039
10000	78	2,001	2,006	128

Ilość wiadomości podana przy tworzeniu pliku dotyczy tworzenia z potwierdzeniem, natomiast przy wyszukiwaniu wraz z wiadomością zwracającą wynik. Dla obu przypadków wartością idealną jest 2. Widać więc, że algorytm bardzo wydajnie adresuje kubełki popełniając niewiele błędów, a ilość przesyłanych wiadomości jest bliska ideałowi.

5.12. Podsumowanie

Algorytm LH* można uznać za dosyć ciekawy jeśli chodzi o rozwiązania pozwalające na działanie w środowisku rozproszonym, oferuje także bardzo dobre parametry wydajnościowe dla wstawiania i wyszukiwania. Pozostaje tylko bardzo istotne pytanie, jak taki system ma się do tradycyjnych scentralizowanych baz danych. Najpierw jednak przedstawione zostaną inne warianty struktury SDDS oraz opisy implementacji, a rozważania nad przydatnością tego typu systemów zostaną przeprowadzone w zakończeniu pracy.

Rozdział 6. Algorytmy $LH*_m$, $LH*_s$, $LH*_g$

O ile można uznać algorytm LH* za ciekawy i mający pewien potencjał o tyle nie sposób nie zauważyć jego bardzo poważnych słabości: zerowego poziomu zabezpieczenia danych przed kradzieżą oraz, przede wszystkim, brak jakichkolwiek mechanizmów mogących zapewnić akceptowalny poziom niezawodności w niestabilnym środowisku sieciowym. Algorytmy $LH*_m$, $LH*_s$ oraz $LH*_g$ są modyfikacjami LH* próbującymi zmniejszyć rażące wady swojego poprzednika.

6.1. Wstęp

Pierwszym wymienionym problemem LH* jest bezpieczeństwo danych. O bezpieczeństwie komputerów podłączonych do sieci napisano już tony książek, ale faktem jest, że 100% zabezpieczenie nawet pojedynczego komputera jest niemożliwe, a jeżeli weźmiemy pod uwagę, że struktura SDDS może rozszerzyć się na setki, a nawet tysiące komputerów to zagrożenie utratą lub kradzieżą danych w skutek nieautoryzowanego dostępu staje się bardzo duże. Algorytm LH* rozkłada pomiędzy serwery całe rekordy, a więc spenetrowanie pojedynczego serwera może pozwolić włamywaczowi na łatwe uszkodzenie bądź kradzież jedynie wycinka danych. Nietrudno sobie jednak wyobrazić jakie straty mogłoby spowodować wykradzenie 1000 numerów kart kredytowych spośród 100000, więc nawet taka ewentualność nie jest do zaakceptowania.

O ile bezpieczeństwo nie we wszystkich zastosowaniach ma szczególne znaczenie (np. jeśli indeksuje się publicznie dostępne dane) to problem niezawodności jest o wiele bardziej dotkliwy. Rozważając strukturę LH* rozmieszczoną na 100 serwerach i przyjmując, że dostępność każdego z serwerów wynosi 99% (przez taki procent czasu serwer poprawnie odpowiada na zapytania) to prawdopodobieństwo zdarzenia, że w danej chwili dostępny jest cały plik wynosi zaledwie 37% (dostępność każdego z serwerów to zdarzenie niezależne), jeśli plik przybrał by rozmiary wymagające uczestnictwa 1000 serwerów to prawdopodobieństwo wynosiło by już zaledwie 0.004%. O ile nie we wszystkich zastosowaniach niezbędna jest 100% dostępność indeksu to jednak widać, że niezawodność algorytmu LH* szybko zbiega to daleko nieakceptowalnych wartości.

Poniżej przedstawione będą trzy algorytmy w różnym stopniu rozwiązujące powyższe problemy i charakteryzujące się różnym poziomem pogorszenia wydajności w porównaniu do algorytmu wzorcowego.

6.2. Algorytm LH$_m$*

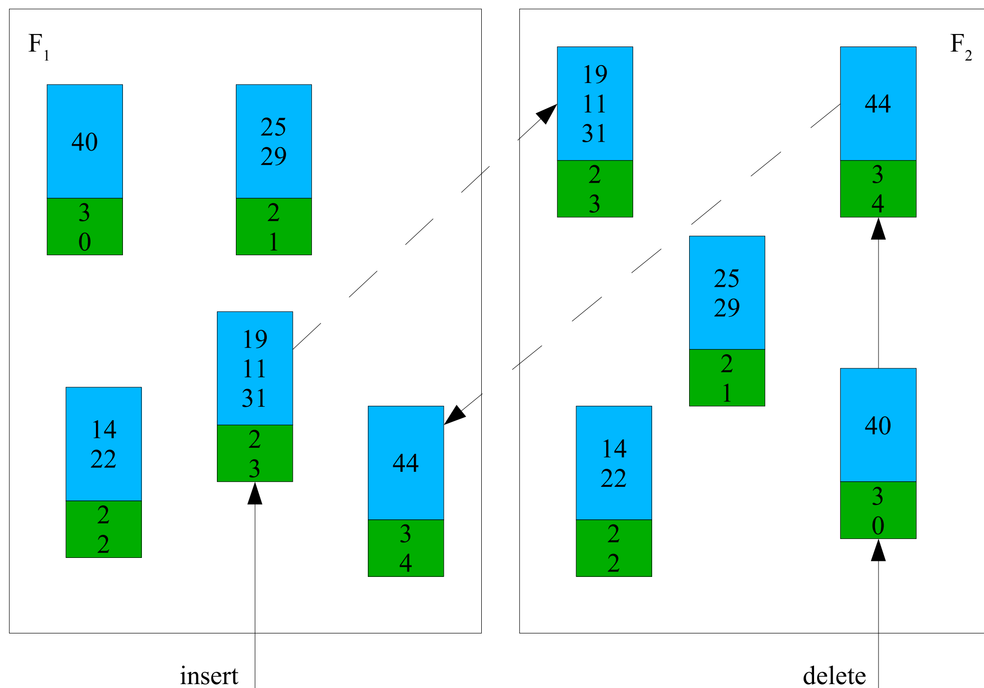
Algorytm LH*$_m$ [LN96] zajmuje się jedynie problemem niezawodności. Jak każde inne rozwiązanie mające zmniejszyć prawdopodobieństwo niedostępności danych LH*$_m$ wprowadza do pliku LH redundancję danych. W przypadku tego algorytmu redundancja polega na utrzymywaniu dwóch bliźniaczych plików F_1 i F_2. Pliki F_1 i F_2 mają rozłączne zbiory podstawowych klientów. Jeśli klient *C* jest klientem podstawowym dla F_1 to jest klientem drugorzędnym dla F_2 i na odwrót. Klient może się kontaktować zarówno z serwerami podstawowego pliku jak i z serwerami pliku drugorzędnego, z tym, że komunikacja z plikiem drugorzędnym odbywa się tylko jeśli w wyniku awarii zajdzie taka potrzeba. Taki podział pozwala dodatkowo wyrównać obciążenie serwerów.

Podstawową rozpatrywaną usterką jest niedostępność kubełka. Kubełek, który przestaje być dostępny uznaje się za stracony i tak szybko jak to możliwe uruchamiana jest procedura odtwarzająca kubełek na innym serwerze.

W algorytmie LH*$_m$ wyróżniono dwa sposoby tworzenia lustrzanych plików:

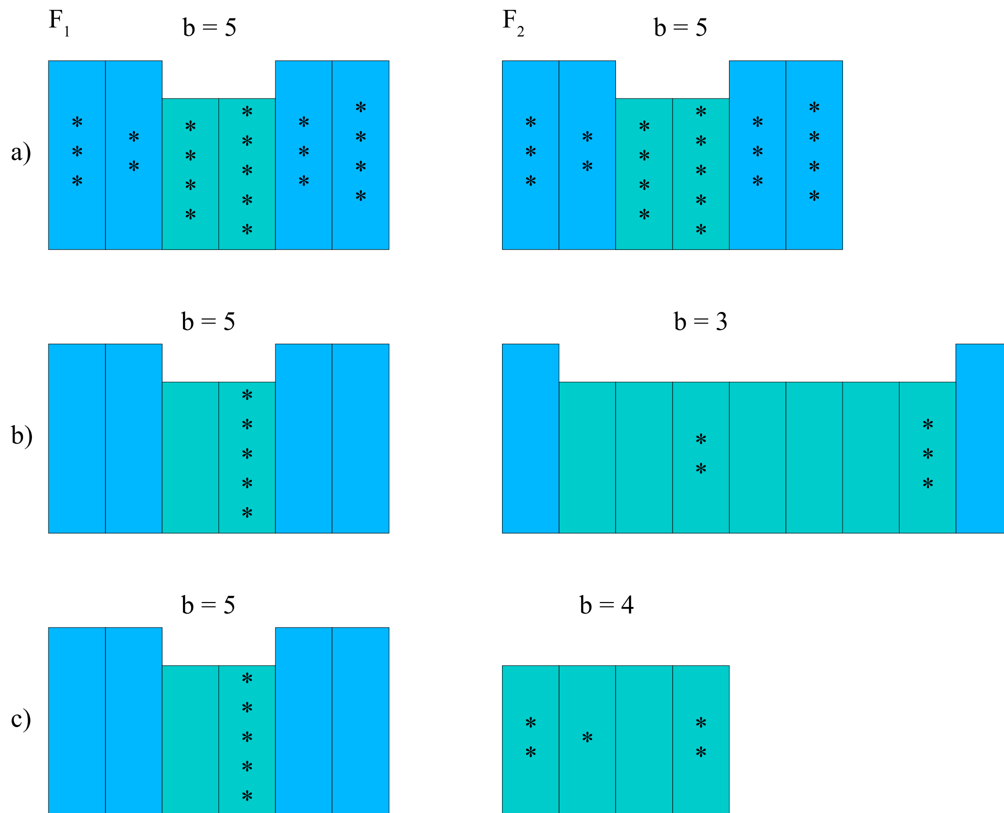
1. Pliki lustrzane są *strukturalnie jednakowe* (ang. structurally-alike) – oba pliki mają takie same parametry (pojemność kubełków, progi kontroli wypełnienia, funkcje mieszające, itp.). Aktualizacje w takich plikach przeprowadzane są tak szybko jak to tylko możliwe.
2. Pliki lustrzane są *strukturalnie odmienne* (ang. structurally-dissimilar) – parametry plików mogą być różne, a aktualizacje mogą być wykonywane w sposób leniwy.

6.2.1. Pliki lustrzane strukturalnie jednakowe



Ilustracja 19. Pliki lustrzane strukturalnie jednakowe

Pliki F_1 i F_2 są strukturalnie jednakowe co oznacza, że z dokładnością do rozmieszczenia są one identyczne (ilustracja 19 oraz 20a). Każdy z kubełków przechowuje w nagłówku (pole zielone) swój poziom oraz numer. Jeśli nie występują błędy spowodowane niedostępnością kubełków operacja wyszukiwania działa tak samo jak dla plików LH*. Operacje zmieniające zawartość pliku (insert, search, update) wymagają natychmiastowej propagacji do pliku lustrzanego (przerywane strzałki). Propagacja w przypadku plików strukturalnie jednakowych jest stosunkowo prosta dzięki symetrii obu plików – kubełek do którego wysłane zostało zapytanie zmieniające stan pliku jest w stanie bezbłędnie zaadresować odpowiadający mu kubełek w pliku lustrzanym. Nieco bardziej skomplikowana jest operacja podziału kubełka – możliwe jest zablokowanie dzielonych kubełków po obu stronach pliku, aż do momentu całkowitego zakończenia operacji dzielenia. Dzięki zastosowaniu blokad nie występują poważne komplikacje, ale znacznie lepsze rezultaty wydajnościowe daje odrębny podział kubełków w plikach lustrzanych. Istnieje wtedy prawdopodobieństwo zaadresowania nieistniejącego kubełka, lecz jest ono stosunkowo małe.



a) pliki lustrzane strukturalnie jednakowe
b) pliki lustrzane strukturalnie odmienne – kopie luźno powiązane
c) pliki lustrzane strukturalnie odmienne – kopie minimalnie powiązane

Ilustracja 20. Typy plików lustrzanych

Przetwarzanie zapytań po stronie serwera

Przetwarzanie zapytań opiera się o założenie, że jeśli klucz *c* znajduje się w kubełku *a* w pliku F_1 to znajduje się także w kubełku *a* w pliku F_2, założenie to jest realizowane przez opisywaną wyżej natychmiastową propagację zmian.

Klient pliku LH* może popełnić błąd adresowania powodując dwukrotne dalsze przekazywanie zapytania. W rozpatrywanym teraz algorytmie LH*$_m$ mogą wystąpić dodatkowe przekazywania zapytania do pliku lustrzanego w przypadku natrafienia na niedostępne kubełki.

Generalnie dla każdego zapytania klienta *C* adresowanego do kubełka a_1 należącego do pliku F_1 kubełek sprawdza czy plik F_1 jest plikiem głównym dla *C*. Jeśli tak to zapytanie jest wykonywane tak jak dla plików LH*. Dodatkowo jeśli zapytanie zmienia stan pliku to właściwy kubełek przekazuje zapytanie do odpowiadającego mu kubełka w pliku F_2. Jeśli F_1 jest plikiem drugorzędnym dla *C* to oznacza, że kubełek w pliku F_2 był niedostępny. Kubełek a_1 powtórnie sprawdza czy kubełek a_2 jest dostępny, jeśli tak to zapytanie jest przekazywane do niego, a klient jest informowany o dostępności a_1. Jeśli sprawdzenie się nie powiedzie a_1 wysyła ostrzeżenie do koordynatora i przystępuje do przetwarzania zapytania. Jeśli zapytanie zostało dobrze zaadresowane i nie zmieniało stanu pliku to zostaje wykonane w a_1.

Jeśli a_1 nie jest właściwym adresem dla zapytania to zapytanie przesyłane jest do g_1 w pliku F_1

lub do g_2 w pliku F_2, jeśli g_1 nie odpowiedziało lub do g_1 i g_2 jeśli zapytanie jest modyfikujące. g_1 sprawdza czy jest adresatem, a jeśli nie to procedura się powtarza.
Kubełek który otrzymuje przekazywane zapytanie zawsze sprawdza czy jest poprawnym kubełkiem dla danego zapytania. Rozbieżności w adresowaniu mogą się pojawić przy asynchronicznym wykonywaniu podziałów kubełków. Jeśli wystąpi błąd w adresowaniu zapytanie jest przekazywane zgodnie ze standardową metodą.
Z tego samego powodu kubełek, jeśli kubełek g_1 nie odpowie, to a_1 może wysłać zapytanie do kubełka g_2, który jeszcze nie został utworzony. W takiej sytuacji a_1 powinien przesłać zapytanie do rodzica g_2, który po zakończeniu podziału prześle zapytanie do g_2.

Przetwarzanie zapytań po stronie klienta

Z powodów synchronicznego podziału kubełków klient także może zaadresować nieistniejący w jednym z plików kubełek (tak samo jak serwer). Klient w takiej sytuacji adresuje rodzica pierwotnego kubełka.
W ogólności klient *C* zaczyna zapytanie od zwykłej kalkulacji adresu zgodnej z algorytmem LH* i używa adresu dla swojego podstawowego pliku. Niech adresatem będzie kubełek a_1. Jeśli kubełek a_1 jest dostępny klient wysyła do niego zapytanie. Jeśli a_1 nie istnieje, to klient wysyła zapytanie do rodzica a_1. Jeśli a_1 jest niedostępny to klient wysyła zapytanie do a_2 w pliku drugorzędnym. Jeśli a_2 jest dostępny to przetwarza zapytanie, a klient odznacza a_1 jako niedostępny i używa zamiast niego a_2, aż do momentu, kiedy a_2 wyśle informacje z nowym fizycznym adresem a_1.

Wykrywanie usterek przez koordynatora

Może się zdarzyć, że niedostępność serwera zostanie wykryta przez koordynatora w momencie inicjacji podziału kubełka. W zaistniałej sytuacji koordynator uruchamia mechanizm tworzenia zapasowego kubełka.

Tworzenie zapasowych kubełków

W momencie wykrycia awarii zachodzi konieczność utworzenia kubełka zapasowego. Jest to kwestia zaalokowania przestrzeni dla nowego kubełka, co w przypadku struktur SDDS nie sprawia specjalnego problemu. Zepsutemu kubełkowi przydzielany jest nowy adres, a z odpowiadającego kubełka w lustrzanym pliku kopiowane są rekordy.

Wydajność

Lustrzane pliki potrzebują oczywiście dwukrotną ilość przestrzeni składowej co zwykły algorytm LH*, wstawianie do pliku wymaga jednej wiadomości więcej, co daje średnio dwie wiadomości, a cztery w najgorszym przypadku. Koszt wyszukiwania się nie zmienia, a podziały kubełków mają koszt dwukrotnie większy od plików LH*.
Awarie kubełków zwiększają koszt zarówno wyszukiwania jak i wstawiania rekordów o dwie

lub maksymalnie kilka wiadomości.
Tworzenie zapasowego kubełka to koszt dwóch wiadomości, jeśli zawartość kubełka można wysłać w obrębie jednej wiadomości.

6.2.2. Pliki lustrzane strukturalnie odmienne

Jak już wcześniej wspomniano, przy takim rozwiązaniu wszelkie parametry dla plików lustrzanych mogą się różnić. Przewagą związaną z różnymi strukturami plików jest lepsze dostosowanie do heterogenicznych środowisk w których mogą się one znaleźć. W odróżnieniu od poprzedniego algorytmu, gdzie kubełki ściśle współpracowały ze swoimi odpowiednikami z drugiego pliku, kubełek w jednym z plików zachowuje się jak klient w stosunku do drugiego pliku. W szczególności oznacza to, że posiada on swój obraz pliku i dostaje wiadomości korekcyjne.
Dodatkowo rozróżniamy dwa typy plików lustrzanych o odmiennych strukturach: takie które używają tej samej funkcji mieszającej nazywane *luźno powiązanymi kopiami* (ilustracja 20b) oraz takie które używają różnych funkcji nazywane *minimalnie powiązanymi kopiami* (ilustracja 20c). Lekko powiązane lustra są interesujące ze względu na potencjalnie prostsze odtwarzanie kubełków.

Luźno powiązane kopie

Przetwarzanie zapytań przez klienta
Podstawową różnicą dla klienta, jest to, że ma on dwa obrazy plików zamiast jednego. Klient normalnie zwraca się do swojego podstawowego pliku, chyba, że zaadresowany kubełek jest niedostępny. Do obliczania adresów używane są wartości odpowiedniego obrazu dla każdego z plików.

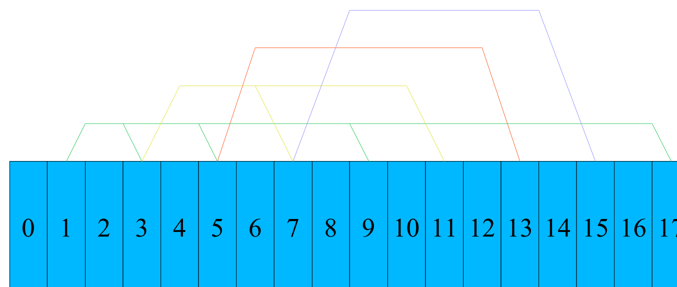
Przetwarzanie zapytań przez serwer
Przy luźno powiązanych kopiach kubełek nie ma odpowiadającego mu kubełka w drugim pliku. Każdy kubełek ma natomiast obraz drugiego pliku i zachowuje się w stosunku do niego jak klient do zwykłego pliku LH*. Oznacza to, że dla zapytania modyfikującego stan pliku kubełek musi wyliczyć odpowiedni adres kubełka z drugiego pliku i następnie tam przekazać zapytanie, może przy tym popełnić błąd i otrzymać wiadomości korekcyjne.
W tym algorytmie występuje zauważalnie więcej wiadomości korekcyjnych niż dla plików lustrzanych strukturalnie jednorodnych. Wykonywane podziały są, ze względu na różnice w strukturze plików, asynchroniczne.

Tworzenie kopii zapasowych
Niech j_1 będzie poziomem niedostępnego kubełka n_1 w pliku F_1. Aby stworzyć nowy kubełek zapasowy należy odnaleźć kubełki zawierające elementy n_1 w pliku F_2. Jak wspomniano

wcześniej użycie takiej samej funkcji mieszającej dla obu plików ułatwia proces wyszukiwania. Może zajść jeden z dwóch przypadków:

Wszystkie rekordy z n_1 są w jednym kubełku w F_2. Taka sytuacja może wystąpić jeśli kubełek n_2 jeszcze nie został utworzony w pliku F_2, a jeśli już istnieje to jest na tym samym lub niższym poziomie co n_1. Jeśli kubełka n_2 jeszcze nie ma w pliku F_2 to należy szukać jego przodka aż do skutku. Zauważmy, że znaleziony tymi sposobami kubełek może zawierać więcej elementów niż n_1 – do wykonania odtworzenia trzeba przefiltrować kubełek funkcją mieszającą na poziomie kubełka n_1.

Może też się zdarzyć, że elementy z n_1 trafiły do wielu kubełków w F_2. W tym przypadku poziom n_2 będzie większy od poziomu n_1. Aby odzyskać wszystkie elementy, które przechowywane były w n_1 należy w sposób zbliżony do propagacji zapytań równoległych w LH* odwiedzić n_2, wszystkich potomków n_2, potomków potomków, itd. (ilustracja 21).



Ilustracja 21. Potomkowie kubełka 1

Wydajność

Koszt wyszukiwania i wstawiania, jeśli nie ma awarii, jest taki sam jak dla plików lustrzanych strukturalnie jednakowych. Jeśli wystąpią awarię to koszty dostępu będą nieco wyższe niż dla jednakowych plików z powodu większej ilości błędów adresowania.

Minimalnie powiązane kopie

Przetwarzanie zapytań jest w tym algorytmie dokładnie takie samo jak w algorytmie dla luźno powiązanych kopii. Jedyna różnica dotyczy procesu odtwarzania zapasowych kubełków. Użycie dwóch różnych funkcji mieszających dla plików lustrzanych powoduje, że każdy z elementów kubełka n_1 z pliku F_1 może znajdować się w dowolnym kubełku pliku F_2. Aby odtworzyć kubełek n_1 trzeba więc wysłać zapytania do wszystkich kubełków F_2. Jako parametr do zapytania musi być podana funkcja mieszająca oraz warunek jaki musi spełniać aby rekord został zwrócony. Jest to kosztowne odtwarzanie pod względem ilości wiadomości, ale obciążenie każdego z kubełków jest takie jak dla pojedynczej operacji wyszukiwania.

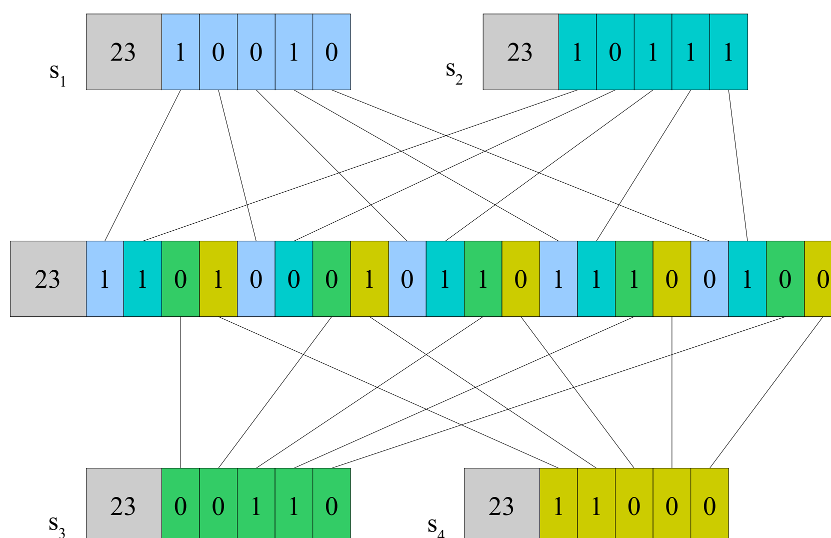
6.3. Algorytm LH*$_s$

Najistotniejszą różnicą między algorytmami LH*$_s$ [LNGNS1], a LH*$_m$ jest inne podejście do uzyskania redundancji. O ile LH*$_m$ tworzy kopię całego pliku, o tyle LH*$_s$ dodaje jedynie stosunkowo niewielką ilość nadmiarowej informacji, a dane przechowywane są w obrębie kilku plików.

6.3.1. Podstawy LH*$_s$

Aby wprowadzić redundancję LH*$_s$ rozważa rekordy nie jako całość, ale jako ciągi bitów i na tym poziomie stosowane są metody podziału rekordu na części oraz wyliczanie informacji nadmiarowej. W przypadku tego algorytmu konieczne jest głębsze rozważenie budowy rekordu – składa się on z klucza oraz ponumerowanego ciągu bitów $B = b_1,...,b_k,...,b_{mk}$ o wielkości mk. Jeśli ciąg bitów jest krótszy stosuje się wypełnienie. Na tym etapie widać już bardzo istotne ograniczenie dla przechowywanych rekordów – zachodzi konieczność ustalenia stałego rozmiaru rekordu. Nieopłacalne lub nawet niemożliwe jest przechowywanie w pliku LH*$_s$ rekordów o zróżnicowanej wielkości.

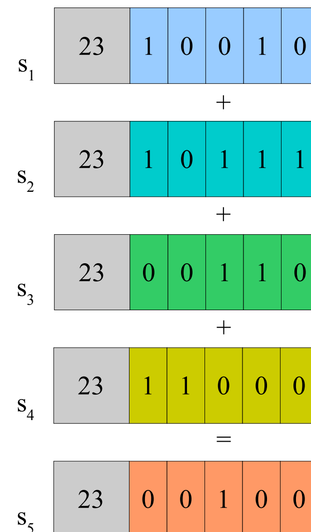
Klient chcący wstawić do pliku rekord R najpierw musi podzielić rekord na k segmentów (k musi być większe od 1, aby algorytm miał sens). Każdy segment s_i zawiera klucz c oraz co k-ty bit rekordu poczynając od bitu i (sama sekwencja bitów bez klucza c oznaczana będzie dalej s'_i) $s'_i = b_i b_{k+i} b_{2k+i}...$(ilustracja 22).



Ilustracja 22. Podział rekordu na segmenty

Następnie wyliczany jest segment parzystości, który także zawiera klucz c oraz ciąg bitów parzystości $s'_{k+1} = b'_1 b'_2 b'_3...b'_m$ gdzie b'_j jest bitem parzystości dla ciągu bitów składających się z j-tego bitu każdego segmentu (ilustracja 23). Jak widać na przykładowej ilustracji zagubienie

jednego z segmentów nie powoduje utraty informacji – dowolny bit można odtworzyć odwracając równanie (uwaga: bit w ostatnim segmencie jest bitem parzystości stąd 1 + 1 + 1 + 0 = 0, równie dobrze może to być równanie nieparzystości i wtedy wynik będzie zgodny ze zwykłym dodawaniem modulo 2).



Ilustracja 23. Tworzenie segmentu parzystości

Plik w algorytmie LH*$_s$ składa się z rodziny $k + 1$ plików LH* - $S_1, ..., S_{k+1}$. Plik S_i przechowuje wszystkie segmenty s_i. Tak samo jak w przypadku algorytmu LH*$_m$ pliki mogą mieć takie same lub odmienne struktury.

Podstawowe zasadą jest to, że każdy z segmentów powinien być umieszczony na innym serwerze (a przynajmniej w innym kubełku). Najprostszym sposobem osiągnięcia tego założenia jest rozmieszenie plików S na rozłącznych zbiorach serwerów, to znaczy jeśli na serwerze znajduje się kubełek z pliku S_i to nie znajduje się tam żaden kubełek z pliku S_j, $i \neq j$.

Plik LH*$_s$ posiada jeden wyznaczony serwer zwany *koordynatorem pliku segmentowego*. Adres koordynatora jest znany wszystkim serwerom i klientom pliku. Koordynator posiada adresy wszystkich serwerów we wszystkich plikach S_i oraz przywraca zagubione kubełki w razie awarii.

6.3.2. Operacje na pliku

Wstawianie

Klient rozpoczyna od podziału rekordu na segmenty. Powstałe $k + 1$ wstawia następnie do kubełków $m_1, ..., m_{k+1}$ w plikach $S_1, ..., S_{k+1}$. Ze względu na dynamiczne odtwarzanie kubełków po awarii może się zdarzyć, że na serwerze jest inny kubełek niż klientowi się wydaje. W tym celu do zapytania jest dołączany numer adresowanego kubełka. Jeśli klient popełni tego rodzaju błąd to zostaje odesłany do koordynatora, który ma aktualne adresy wszystkich kubełków. Taka sama procedura występuje w momencie przesyłania zapytania między serwerami (one także mogą mieć nieaktualne adresy kubełków). Do maksymalnych dwóch odesłań w algorytmie LH* należy więc doliczyć trzy ewentualne odesłania do koordynatora.

Liczba wiadomości korekcyjnych zależy od rodzaju plików S_i, jeśli mają one jednakową strukturę to wystarczy jedna korekcja obrazu, jeśli mają odmienne struktury klient może otrzymać wiadomości korekcyjne od każdego z plików S_i.

Klient jak i serwery mogą napotkać po drodze na niedostępne kubełki. Powiadamiany jest wtedy koordynator. Klient może skutecznie wykonać operację jeśli awarii uległ co najwyżej jeden kubełek, w przeciwnym przypadku musi czekać na odpowiedź od koordynatora. Jeśli klient błędnie zaadresował uszkodzony kubełek to może się zdarzyć, że nawet mimo awarii

więcej niż jednego kubełka można wykonać operację, koordynator w takiej sytuacji jest w stanie podać klientowi prawidłowe adresy kubełków.

Podziały

Podziały są wykonywane dosyć podobnie jak w LH*, brane są dodatkowo pod uwagę możliwe awarie występujące podczas operacji podziału. Jeśli awaria dotyczy nowo utworzonego kubełka, a podział nie został jeszcze zatwierdzony to procedura podziału rozpoczynana jest od początku z nowym adresem kubełka. Jeśli awaria wystąpi w kubełku przechodzącym podział to jest on odtwarzany tak jak inne, a następnie ponownie uruchamiany jest jego podział.

Usuwanie

Fizyczne usunięcie przebiega analogicznie do wyszukiwania, tyle że możliwy jest wariant bez informacji zwrotnej.

Wyszukiwanie

Aby wyszukać rekord o kluczu c klient musi wysłać zapytanie z c do k serwerów $S_1, ..., S_k$. W zależności od budowy plików potrzebne będzie jedno wyliczenie adresu (dla plików strukturalnie jednakowych) lub wyliczenie osobnego adresu dla każdego z plików (dla plików strukturalnie odmiennych). Tak samo jak przy wstawianiu każda wiadomość posiada dodatkowo numer kubełka do którego jest adresowana. Przebieg poszukiwania odpowiedniego kubełka jest identyczny jak dla operacji wstawiania.

Jeśli wszystkie serwery odpowiedziały, klient może bez przeszkód poskładać wiadomość odwracając proces podziału. Gdy jedne z kubełków nie odpowiada, klient zawiadamia koordynatora, a sam wysyła zapytanie do pliku S_k w którym przechowywany jest segment parzystości. Jeśli tylko jeden kubełek nie odpowiedział klientowi wiadomość może zostać odtworzona, w przeciwnym przypadku wyszukiwanie kończy się błędem.

Awarie

Rozróżnić można dwa rodzaje awarii:

1. Klient podczas wyszukiwania na dostępnym serwerze nie znajduje poszukiwanego kubełka. Jeżeli kubełek faktycznie powinien znajdować się na zaadresowanym serwerze to natychmiast uruchamiana jest procedura odtwarzania zapasowego kubełka. Klient nie musi czekać na odtworzenie kubełka, jeśli tylko jeden z kubełków z poszukiwanymi segmentami jest niedostępny – można odtworzyć rekord posługując się segmentem parzystości.
2. Klient nie może skontaktować się z poszukiwanym kubełkiem. Taka sytuacja nie musi oznaczać awarii serwera, a jedynie awarię połączenia sieciowego. Możliwe są dwa rozwiązania takiego problemu – natychmiastowe odtworzenie kubełka i skasowanie starej wersji kubełka jak tylko połączenie zostanie przywrócone albo wykonywanie operacji na pozostałych k kubełkach i aktualizacja niedostępnego kubełka, kiedy tylko połączenie

zacznie funkcjonować. Pierwsze rozwiązanie oferuje oczywiście znacznie większą niezawodność.

6.3.3. Odtwarzanie kubełków

Odtworzenie kubełka wymaga od kontrolera utworzenia nowego kubełka na jednym z dostępnych serwerów, a następnie odszukanie wszystkich segmentów rekordów, które mogły znajdować się w zepsutym kubełku. Procedura wyszukiwania takich rekordów zależy od struktur użytych plików, a różne jej warianty wyglądają tak samo jak te zaprezentowane dla różnych typów plików w algorytmie LH^*_m. Po zebraniu wszystkich segmentów koordynator rekonstruuje rekordy i wstawia je do nowego kubełka.

6.3.4. Bezpieczeństwo

Podział rekordów na segmenty zawierające co *k*-ty bajt znacznie zmniejszają możliwości przechwycenia danych, zarówno przez włamanie do serwerów przechowujących segmenty jak i przez nasłuchiwanie w sieci. Szczególnie jeśli weźmiemy pod uwagę komunikację nie wykorzystującą rozgłoszeń oraz przełączane sieci (w dzisiejszych czasach to już norma). Ceną jaką za to trzeba zapłacić jest znacznie większa ilość wiadomości niezbędnych do zakończenia wszelkich operacji (liniowo zależna od ilości segmentów).

6.3.5. Wydajność

Wstawianie. Podziały oraz przekazywanie nie powinny być częste w plikach LH^*_s, a awarie są jeszcze rzadsze. Wstawienie rekordu do pliku LH* kosztuje średnio jedną wiadomość (dwie z potwierdzeniem). Można więc oczekiwać, że koszt dla LH^*_s będzie $k+1$ razy większy. Istotne zwiększenie czasu dostępu wystąpi w przypadku awarii (średnio trzy wiadomości więcej, a maksymalnie $9*(k + 1)$).

Wyszukiwanie. Koszty dotyczące wyszukiwania są dokładnie takie same jak dla wstawiania z potwierdzeniem.

Usuwanie. Koszt taki sam jak powyżej.

Odtwarzanie kubełków. Koszt odtwarzania może się bardzo różnić w zależności od różnic w strukturach tak jak to opisano w poprzednim rozdziale. W uproszczeniu koszt odtwarzania LH^*_s jest równy *k* krotnemu kosztowi odtwarzania LH^*_m o takich samych różnicach strukturalnych.

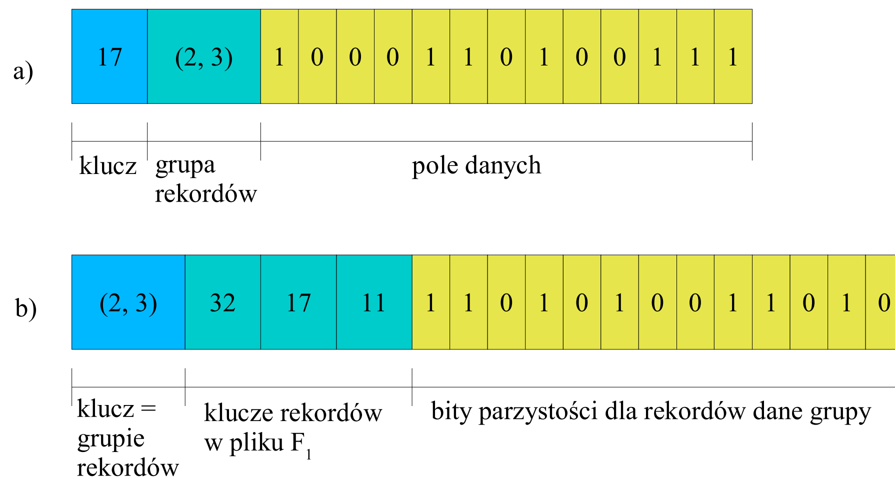
6.4. Algorytm LH^*_g

LH^*_g to najbardziej skomplikowana modyfikacja algorytmu LH* rozpatrywana tutaj. W porównaniu z poprzednim algorytmem LH^*_g oferuje znacznie mniejszy koszt wyszukiwania

(taki sam jak dla LH* jeśli nie wystąpią żadne awarie), a przy tym nie wymaga tyle przestrzeni składowej co LH^*_m.

6.4.1. Struktura pliku LH^*_g

Plik LH^*_g składa się z dwóch plików LH* - głównego pliku F_1 oraz pliku parzystości F_2. Dla całego pliku wyróżniony jest jeden koordynator zarządzający jego stanem. Plik F_1 ma na początku działania *k* kubełków ($k > 1$). Każdy kubełek *m* w pliku F_1 należy do pewnej logicznej grupy kubełków $g = floor(m/k)$ o wielkości *k* kubełków. Rekord w pliku F_1 oprócz klucza i pola niekluczowego (w postaci stałej liczby bitów) zawiera dodatkowo numer swojej grupy rekordów *rg* (ilustracja 24a). Dodatkowo każdy z kubełków w pliku F_1 zawiera *licznik wstawień r* po utworzeniu kubełka ustawiony na 1. Wstawianym rekordom jest przydzielana bieżąca wartość *r*, po czym jest ona powiększana o 1. Rekordy które trafiają do kubełka w wyniku podziału zatrzymują swoje wcześniejsze wartości *r*.
Wstawianemu rekord jest przypisywana grupa rekordów $rg = (g, r)$, która jest niezmienna do końca istnienia. Wartość *g* to grupa kubełka do którego wstawiany jest rekord, a *r* to licznik wstawień tegoż kubełka.



Ilustracja 24. Struktura rekordów w plikach LH^*_g

Dla każdej grupy rekordów *rg* (czyli dla każdego z rekordów w F_1 należącego do grupy rekordów *rg*) istnieje rekord parzystości w pliku F_2. Rekord ten umieszczony jest pod kluczem *rg*. Aby zwiększyć niezawodność zbiory serwerów pliku F_1 i F_2 są rozłączne. Rekord parzystości składa się z klucza, którym w tym przypadku jest grupa rekordów czyli dwójka *(g, r)*, w polu danych zapisane są klucze wszystkich rekordów (jest ich co najwyżej *k*) z danej grupy oraz bity parzystości wyliczone z *i*-tych bitów każdego z rekordów należących do grupy.

Należy tutaj odnotować, że wstawienie nowego rekordu do grupy nie wymaga wyciągnięcia z pliku wszystkich innych rekordów z danej grupy i wyliczenie bitów parzystości od zera. Wystarczy wziąć bity wstawianego rekordu oraz dotychczasowe bity parzystości i zastosować operację *XOR*. Tak wyliczony rekord pozwala odtworzyć każdy pojedynczy zagubiony rekord z grupy *rg*.

Dla tak skonstruowanego pliku prawdziwe są własności:

1. Liczba rekordów w grupie *rg* nigdy nie przekracza *k*. Ilość rekordów w grupie to istotny czynnik warunkujący niezawodność – im więcej rekordów tym większe prawdopodobieństwo awarii dwóch kubełków zawierających rekordy z jednej grupy.
2. Żadne dwa rekordy z jednej grupy rekordów nigdy nie znajdą się w tym samym kubełku. Gdyby takie założenie nie było spełnione to awaria jednego kubełka mogłaby spowodować utratę danych.

6.4.2. Operacje na pliku

Wstawianie

Klient wykonuje operację na pliku w taki sam sposób jak dla algorytmu LH* z tym, że operacja rozszerzona jest o obsługę niedostępnych kubełków. Jeśli jednak nie zostaną napotkane żadne awarie to dla wstawienia rekordu do kubełka *m* w działaniu klienta nie będzie żadnych różnic. Wewnątrz pliku nastąpi natomiast wyliczenie nowego segmentu parzystości – kubełek *m* zachowując się jak klient pliku F_2 zaktualizuje (lub utworzy jeśli takowy nie istnieje) rekord parzystości dla odpowiedniej grupy rekordów. W momencie napotkania awarii kubełka klient przekazuje zapytanie do koordynatora, który najpierw sprawdza czy rekord o zadanym kluczu istnieje, jeśli tak to wysyła komunikat o błędzie, jeśli nie to informuje o sukcesie, a sam asynchronicznie przywraca zagubiony kubełek i wstawia do niego otrzymany rekord. Jeśli błędy wystąpią podczas przekazywania zapytania przez serwery stosowana jest ta sama procedura.

Wyszukiwanie

Wyszukiwanie wykonywane jest analogicznie do wstawiania w LH*, jedynie w momencie napotkania awarii zapytanie przesyłane jest do koordynatora, a ten dokonuje natychmiastowego odtworzenia rekordu z rekordu parzystości oraz asynchronicznie uruchamia procedurę odtwarzania kubełka.

Usuwanie

Procedura działania jak dla wyszukiwania. Usuwanie rekordów może być logiczne lub fizyczne. W przypadku logicznego usunięcia rekordu potrzebny jest jedynie prosty mechanizm oznakowania rekordów jako istniejące i usunięte. Operacja usunięcia polega więc jedynie na

ustawieniu odpowiedniej flagi w kubełku. Rekord parzystości nie musi być aktualizowany, bo fizycznie rekord nadal istnieje i może być wykorzystany w procesie odtwarzania. Fizyczne usunięcie powoduje zmniejszenie licznika wstawień w odpowiednim kubełku oraz aktualizację rekordu parzystości. Należy zauważyć, że funkcja XOR jest przemienna i odwracalna, dzięki temu aby zaktualizować rekord parzystości wystarczy zastosować funkcję XOR do jego bieżącej wartości i pola danych usuwanego rekordu.

6.4.3. Odtwarzanie kubełka podstawowego

Aby odtworzyć kubełek podstawowy w F_1 koordynator najpierw wysyła zapytanie do wszystkich kubełków w pliku parzystości F_2 z bieżącym stanem pliku F_1 (wartościami i oraz n) oraz numerem zepsutego kubełka m. Na podstawie takiego zapytania kubełki F_2 mogą wyselekcjonować wszystkie rekordy, które w danym momencie znajdowały się w kubełku m. Dla każdego znalezionego rekordu zwracana jest jego grupa, rekord parzystości oraz wszystkie inne rekordy, które do tej grupy należą. Koordynator po otrzymaniu takiej informacji może asynchronicznie przetwarzać każdy rekord – pobierać pozostałe rekordy z danej grupy i wyliczać na ich podstawie (wraz z rekordem parzystości) wartość utraconego rekordu. Po złożeniu rekordy wstawiane są do nowego kubełka, a dodatkowo z wartości grup rekordów wyliczana jest wartość licznika wstawień. Aby zbytnio nie obciążać koordynatora można kierować wyniki zapytań od razu do nowo utworzonego kubełka, który sam dokona odtworzenia rekordów.

6.4.4. Odtwarzanie kubełka parzystości

Odtworzenie kubełka parzystości polega na wysłaniu zapytania przez nowy kubełek do wszystkich kubełków F_1, w zapytaniu przekazywany jest stan pliku F_2 w chwili wystąpienia awarii. Na tej podstawie kubełki F_1 wybierają rekordy należące do grup umieszczonych w utraconym kubełku. Po otrzymaniu wszystkich rekordów nowo utworzony kubełek rekonstruuje rekordy parzystości.

6.4.5. Odtwarzanie stanu pliku

Jak wcześniej wspomniano, stan pliku przechowywany jest na jednym wyróżnionym serwerze. Awaria tego serwera nie jest w żaden sposób wykluczona, konieczne jest więc przygotowanie pliku na taką ewentualność. Odzyskiwanie stanów plików F_1 i F_2 wykorzystuje algorytm przedstawiony przy omawianiu korekcji obrazu klienta w zapytaniach równoległych dla plików LH*. Wyznaczony nowy koordynator wysyła zapytanie do wszystkich kubełków w F_1 i F_2, a w odpowiedzi dostaje poziom każdego z odpytanych kubełków. Na tej podstawie wyliczany jest aktualny poziom pliku.

6.4.6. Wydajność

Jeśli chodzi o przestrzeń potrzebną do składowania pliku LH^*_g to jest ona większa o *1/k* od przestrzeni potrzebnej do przechowania analogicznego pliku w algorytmie LH*. Jest to mniej więcej tyle samo co dla LH^*_s i znacznie mniej niż dla LH^*_m.

Wyszukiwanie

Wyszukiwanie dla plików LH^*_g jest równa wydajności dla plików LH*, jeśli po drodze nie wystąpią żadne awarie. Do tego dochodzi koszt odtwarzania rekordów w przypadku niedostępności kubełka oraz, ze względu na możliwość zmian adresów kubełków, dodatkowe błędy adresowania. Awarie jednak nie zdarzają się bardzo często więc nie powinno mieć to dużego wpływu na wydajność.

Wstawianie

Jeśli nie rozpatrywać awarii to w porównaniu do algorytmu LH* wstawianie wymaga wysłania dodatkowej wiadomości – z kubełka wstawiającego rekord do kubełka zawierającego rekord parzystości dla danej grupy. W normalnej sytuacji może to zwiększyć ilość niezbędnych wiadomości o najwyżej trzy.

Podział kubełków

Dzięki temu, że grupy rekordów nie zmieniają się po ich utworzeniu, koszt podziału kubełków w F_1 zostaje taki sam, należy do tego doliczyć koszt podziału dodatkowych kubełków z pliku F_2. Rekordów w pliku F_2 jest około k razy mniej, więc podziały wystąpią k razy rzadziej. Stąd wzrost kosztu podziału kubełków o *1/k* w stosunku do LH*.

Odtwarzanie kubełków

W obu opisanych przypadkach odtwarzanie kubełków sprowadza się do wysłania zapytania do wszystkich kubełków w pliku F_1 lub F_2. Co powoduje, że koszt odtwarzania dla tego rozwiązania jest najwyższy ze wszystkich prezentowanych w tym rozdziale.

6.4.7. Pliki LH^*_g z n-dostępnością

Wszystkie przedstawione w tym rozdziale algorytmy oferują większą dostępność niż LH* i są w stanie odtworzyć dane z dowolnego straconego kubełka. Odzyskanie danych gwarantowane jest jednak tylko pod warunkiem, że awarii uległ jeden kubełek w całym pliku. Awaria dwóch lub więcej kubełków może spowodować (z prawdopodobieństwem zależnym od parametrów pliku) całkowitą utratę części danych.

Algorytm LH^*_g daje możliwość łatwego wprowadzenia własności *n-dostępności*, co oznacza możliwość odzyskania kompletu danych po stracie n kubełków na raz (czyli wszystkie opisane algorytmy oferują 1-dostępność). n-dostępność uzyskuje się przez przypisanie każdego

elementu do *n* grup zamiast do jednej. Aby zapewnić odtwarzalność danych w przecięciu dowolnych dwóch grup może znajdować się co najwyżej jeden rekord.
W jaki sposób taka konstrukcja zapewnia *n*-dostępność? Powiedzmy, że chcemy wyciągnąć rekord *r* i awaria nastąpiła w *n* kubełkach. Z założenia mamy, że rekord *r* należy do *n* grup rekordów rg_1, ..., rg_n. W każdej z tych grup jest przynajmniej jeden niedostępny rekord – poszukiwany *r* – jest on w jednym z uszkodzonych kubełków. Żaden z dwóch rekordów w tej samej grupie nie jest w tym samym kubełku oraz żaden rekord poza *r* nie należy do więcej niż jednej z wymienionych grup – z tego wynika, że poza *r* w grupach rg_1, ..., rg_n jest co najwyżej $n - 1$ uszkodzonych rekordów poza *r*, a więc przynajmniej w jednej grupie jedynym uszkodzonym rekordem jest *r* i można go odzyskać dzięki innym rekordom z grupy oraz rekordowi parzystości.

6.5. Porównanie LH*, LH*$_m$, LH*$_s$, LH*$_g$

Algorytmy z rodziny LH* oferują różne własności dla różnych operacji, na miejscu jest więc porównanie ich ze sobą w różnych kategoriach. Dane przedstawione w poniższej tabeli pochodzą z pracy [LR1]. Numery w polach oznaczają miejsca w rankingu dla danej operacji.

	Wyszuki-wanie	*Wstawianie*	*Przestrzeń składu*	*Dostępność*	*Odtwarzanie*	*Przepusto-wość*	*Bezpie-czeństwo*
LH*	1	1	1	-	-	2	2
LH*$_m$	1	2	4	1	1	1	4
LH*$_s$	2	4	3	2	2	4	1
LH*$_g$	1	3	2	1	3	3	3

6.5.1. Wyszukiwanie

Algorytmy LH*, LH*$_m$, LH*$_g$ prezentują dla tej operacji taką samą wydajność. Wszystkie wymagają średnio jednej wiadomości a co najwyżej trzech w sytuacji bezawaryjnej. Algorytm LH*$_s$ wymaga do wykonania udanego wyszukiwania złożenia *k* segmentów rekordu rozmieszczonych w *k* różnych kubełkach. W tym celu należy przeprowadzić *k* wyszukiwań w różnych plikach (co może zwiększyć koszt w porównaniu z poprzednimi algorytmami *k* razy).

6.5.2. Wstawianie

Koszt wstawiania jest różny dla każdego algorytmu. Klasyczny LH* wymaga góra trzech wiadomości, LH*$_m$ wymaga trzech wiadomości do zaadresowania kubełka przez klienta w jednym z plików i jednej do propagacji zmiany do drugiego pliku (dla plików strukturalnie

jednakowych) bądź trzech dla strukturalnie odmiennych czyli maksymalnie czterech lub sześciu, LH^*_g wymaga w najgorszym przypadku trzech wiadomości do wstawienia rekordu do głównego kubełka i tyle samo do zaktualizowania rekordu parzystości. Najgorszy jest w tej operacji LH^*_s, który musi tym razem wykonać $k + 1$ operacji wstawiania, a każda z nich może wymagać trzech wiadomości.

6.5.3. Przestrzeń składu

Najlepszy jest tu oczywiście algorytm LH*, gdyż nie przechowuje żadnych informacji redundantnych. LH^*_s i LH^*_g przechowują dodatkowo $1/k$ informacji nadmiarowych, ale że w przypadku LH^*_s od k zależy liniowo koszt operacji wstawiania i wyszukiwania to LH^*_g wygrywa w tej kategorii. LH^*_m przechowuje co najmniej dwa razy tyle informacji co LH* dla takiego samego pliku.

6.5.4. Dostępność

LH* nie posiada żadnych mechanizmów pozwalających poradzić sobie z awariami kubełków, więc nie jest brany w tej kategorii pod uwagę. Najlepszą dostępność oferują algorytmy LH^*_m, LH^*_g gdyż oba pozwalają na zastosowanie n-dostępności.

6.5.5. Odtwarzanie kubełków

W większości rozwiązań odtworzenie kubełka w LH^*_m wymaga jedynie skopiowania zawartości jednego kubełka do drugiego lub przeszukanie ograniczonej ilości kubełków. LH^*_s jest istotnie gorszy, gdyż wymaga odpytania podobnej liczby kubełków co LH^*_m pomnożonej przez liczbę segmentów + 1. LH^*_g prezentuje sie najgorzej – wymaga odpytania wszystkich kubełków w jednym z plików.

6.5.6. Przepustowość

Największą przepustowość osiąga LH^*_m dzięki podziałowi zbiorów klientów między swoje pliki lustrzane, a dzięki temu rozłożeniu obciążenia. Klasyczny LH* ma taki sam koszt liczony w wiadomościach jak LH^*_m, ale wszyscy klienci korzystają z tych samych serwerów – stąd drugie miejsce. Także w LH^*_g wszyscy klienci korzystają z tej samej grupy serwerów, a dodatkowo wymagane są wiadomości aktualizujące rekordy parzystości. LH^*_s ze względu na bardzo duży koszt operacji wstawiania i wyszukiwania jest najgorszy.

6.5.7. Bezpieczeństwo

Najbezpieczniejszy jest specjalnie w tym celu zaprojektowany algorytm LH^*_s. Drugi jest klasyczny LH*, głównie dzięki temu, że unika redundancji. Niewiele gorszy jest LH^*_g, który

wprowadza niewielką redundancję (co ułatwia nieuprawniony dostęp do danych), ale ze względu na rodzaj redundancji, jest ona trudna do wykorzystania, nieco gorzej ma się sprawa z większą liczbą przesyłanych w sieci pełnych rekordów. Najgorzej sytuacja wygląda w LH^*_m, gdzie przechowywane są (przynajmniej) dwie pełne kopie rekordów, a co za tym idzie istotnie zwiększa się prawdopodobieństwo włamania i kradzieży danych.

6.6. Podsumowanie

W swojej klasycznej postaci algorytm LH* ma bardzo ograniczone zastosowanie, głównie ze względu na to, że nie radzi sobie z awariami. Przedstawione modyfikacje podnoszą niezawodność działania systemu, czyniąc pliki znacznie bardziej przydatnymi do rzeczywistych zadań. Koszty poszczególnych modyfikacji są różne dla różnych operacji. Jak widać w porównaniu nie ma wśród nich algorytmu, który oferowałby dobrą wydajność dla wszystkich rozpatrywanych operacji. Wybór algorytmu dla praktycznego zastosowania powinien być podyktowany specyfiką umieszczanych w systemie plików (ilość wstawień, częstotliwość awarii, itp.).

Rozdział 7. Implementacja

7.1. *Rozwiązania przyjęte w pracy*

7.1.1. Java

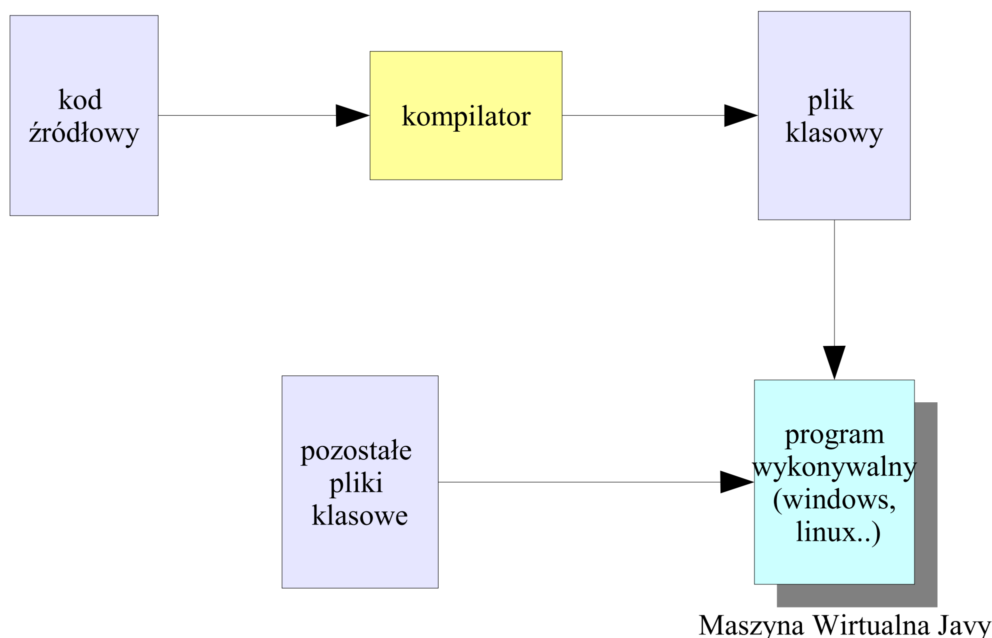
Java jest nowoczesnym językiem programowania. Zrealizowano w nim paradygmat programowania zorientowanego obiektowo, z wbudowanymi mechanizmami współbieżności, obsługi sytuacji wyjątkowych i kontrolą przydziału pamięci. Jest on niezależny od platformy systemowej dzięki temu, że kod programów jest najpierw kompilowany do kodu pośredniego (byte code), a następnie interpretowany przez maszynę wirtualną odpowiednią dla konkretnego systemu operacyjnego (JVM).

Podczas pisania aplikacji nie trzeba wywoływać funkcji systemu operacyjnego. Wszystko znajduje się we własnych bibliotekach zwanych pakietami.

Java posiada też takie cechy jak:

- prosta: konstrukcje składniowe są bardzo podobne do C++, zrezygnowano jednak z wielu niejasnych i skomplikowanych elementów tego języka; zastosowanie odśmiecacza (garbage collector) pozwala uniknąć wielu prostych błędów
- zorientowana obiektowo: konstrukcje są jasne i przejrzyste, łatwiejsze projektowane i zrozumienie problemu, łatwo stosować ponowne użycie stworzonego kodu (reusable)
- wspierająca środowisko sieciowe: java zawiera biblioteki do łatwej wymiany informacji przez sieć za pośrednictwem protokołów HTTP, ftp itp.
- solidna: Java zawiera mechanizmy ułatwiające pisanie poprawnych programów
- bezpieczna: z uwagi na istotność działania w środowisku sieciowym, java dostarcza mechanizmów uwierzytelniania, zabezpieczenia przed wirusami itp.
- neutralna: kompilator generuje kod pośredni, który może być później uruchamiany na dowolnym systemie dla którego istnieje implementacja JVM
- przenośna: niezależnie od platformy typy proste zawsze mają taką samą konstrukcję; także dostęp do bibliotek systemowych ukryty jest pod postacią zunifikowanych interfejsów
- interpretowana: bytecode javy tłumaczony jest w locie do kodu właściwego dla danego systemu; natywny kod nie jest nigdzie przechowywany
- wydajna:
- wielowątkowa: tworzenie wątków zostało wprowadzone do definicji oraz składni języka; dzięki temu wielowątkowa aplikacja będzie działała na każdym systemie operacyjnym
- dynamiczna: poszczególne komponenty (pakiety) nie zależą od siebie, (np. w C++ po zmianie jednej biblioteki trzeba przekompilować cały produkt); istnieje mechanizm interfejsów specyfikujący metody dostępu, a pomijający implementację

Z punktu widzenia tej pracy magisterskiej istotne jest, że java uwzględnia architekturę klient-serwer w swojej konstrukcji. Standardowe biblioteki javy zostały rozszerzone o możliwość komunikacji sieciowej. Dzięki temu, po połączeniu np. z inną aplikacją internetową i czytać i zapisywać dane w taki sam sposób, jak np. program napisany w C uzyskuje dane z lokalnego terminala.



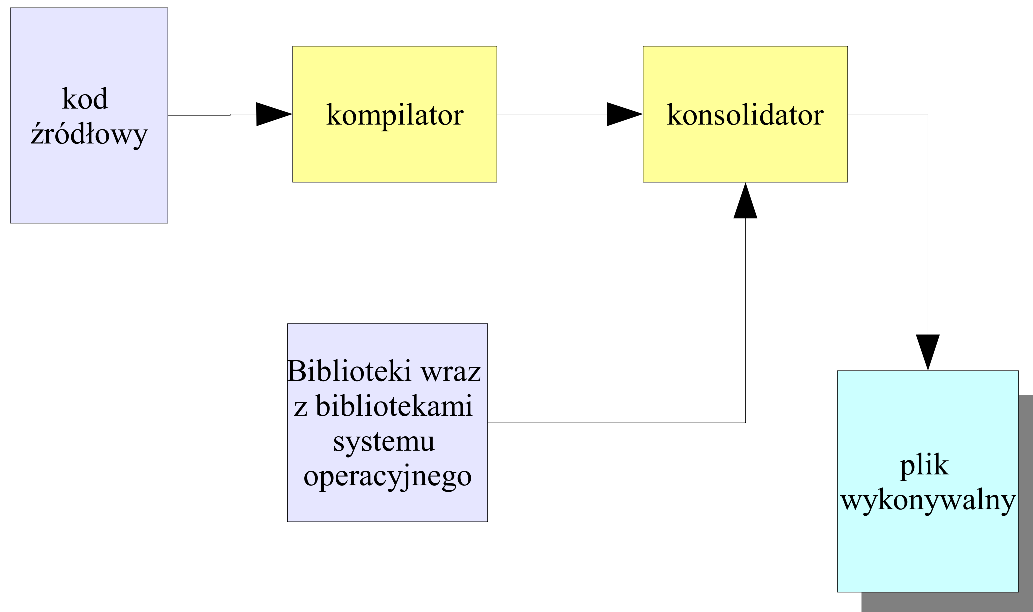
Ilustracja 25. Schemat działania kompilatora Javy i interpretowania kodu przez JVM

Na podstawie:
[J1]
[M01]
[J2]

7.1.2. C++

Język C++ powstał na bazie języka C. W stosunku do swojego pierwowzoru różni się przede wszystkim tym że jest zorientowany obiektowo (posiada więc takie elementy jak klasy, dziedziczenie, konstruktory, destruktory itp.).
Język C++ charakteryzuje się dużą szybkością działania programu wynikowego. Jednak programy napisane w tym języku są nieprzenośne pomiędzy różnymi platformami

systemowymi i niezbędna jest ponowna kompilacja kodu (często wymagane są też zmiany w kodzie). Język C++ (C) daje jednak ogromne możliwość programiście. Łączy on zalety języków wysokiego poziomu z szybkością oraz wydajnością zbliżoną do assemblera.



Ilustracja 26. schemat kompilacji kody źródłowego do programu w C++, wynik jest zależny od platformy docelowej

7.2. SDDS-2000

SDDS-2000 (Scalable Distributed Data Structures) jest prototypowym systemem stworzonym do zarządzania danymi rozproszonymi w sieci, przechowywanymi w pamięci RAM.

SDDS pracuje w lokalnej sieci na maszynach z systemem Windows 2000.

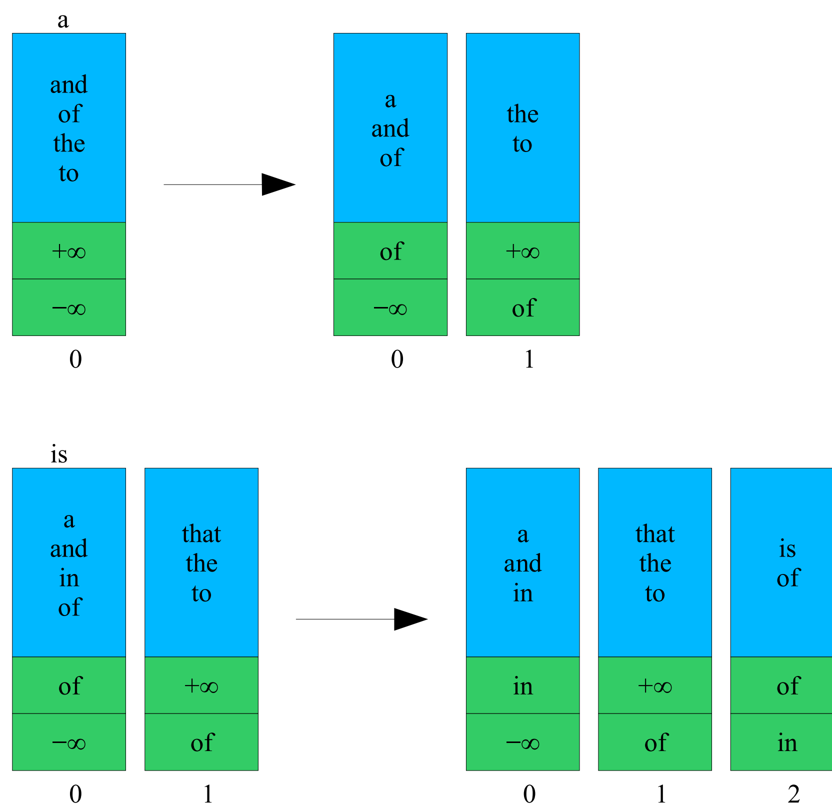
System SDDS-2000 składa się z trzech podstawowych elementów:

- SDDS Server – zarządza strukturami danych przechowywanych w ramach systemu SDDS
- SDDS Name Server- zarządza wszystkimi plikami (nazwami plików) danych utworzonymi w ramach jednego systemu serwerów SDDS
- SDDS Client Service – uruchamia się jako serwis na maszynie klienckiej, za jego pośrednictwem możliwe jest wysyłanie żądań tworzenia pliku lub wyszukiwania do serwera SDDS

dodatkowo wraz z pakietem dostarczana jest aplikacja testowa operująca na danych za pośrednictwem Client Service oraz interfejs programistyczny, pozwalający z poziomu własnych aplikacji napisanych w C/C++ wywoływać funkcje SDDS Client Servcie.

7.2.1. Algorytmy z rodziny RP*

W systemie SDDS-2000 zastosowano nieco inny algorytm niż opisywany wcześniej LH*. LH* opiera się na funkcjach mieszających, natomiast rodzina algorytmów RP* [LNS94] jest generalizacją struktury B^+-drzew, pozwalającą na stosowanie jej w środowisku rozproszonym. Główną zaletą takiego rozwiązania jest przechowywanie danych w sposób uporządkowany. U podstaw algorytmów RP* nie leżą już funkcje mieszające, ale zakresy kluczy przypisane poszczególnym kubełkom. Podział kubełka nie polega więc na odfiltrowaniu rekordów nową funkcją, ale na wybraniu elementu środkowego w kubełku i poczynając od niego przeniesienie wszystkich elementów do nowego kubełka (ilustracja 27).



*Ilustracja 27. Ewolucja pliku RP**

Takie rozwiązanie można uznać za bardzo ciekawe, głównie ze względu na możliwość wyszukiwania przedziałami. Tego typu zapytania często stosuje się w praktyce i jest to istotne przewaga RP* nad LH*. Niestety w SDDS-2000 ograniczono się do kluczy całkowitych i w dodatku zaledwie od 0 do 1 miliona, a przecież nietrudno zobaczyć, że algorytm RP* aby w pełni wykorzystać swoje możliwości potrzebuje zbioru pozwalającego na dowolne

zagęszczanie wartości. Takie możliwości dają ciągi znaków o nieograniczonej (a nawet ograniczonej, ale dostatecznie długie), ale na pewno nie milion liczb całkowitych. Można więc powiedzieć, że ciekawe własności algorytmów z rodziny RP* zostały w SDDS-2000 zwyczajnie zmarnowane.

7.2.2. Architektura systemu

System SDDS-2000 uruchamia się w sieci lokalnej. Do poprawnego działania aplikacji niezbędne jest uruchomienie modułu Name Service oraz przynajmniej jednego Serwera . Na maszynie klienta należy uruchomić Client Service.

Rekord danych składa się z klucza oraz wartości niekluczowej, mogącej przechowywać dowolne dane. Jednak wielkość tego pola jest ograniczona do 100 bajtów.

Klucz	*Wartość niekluczowa*
1	10010100010
2	11000010101
3	100101011

Ilustracja 28. Rekord danych SDDS-2000

Wszystkie dane przechowywane są w pamięci RAM. Dzięki temu dostęp do nich jest niezwykle szybki. Różnice w czasie dostępu w porównaniu do lokalnych dysków twardych są ogromne (1μs w porównaniu do 10 ms). Problem związany z ewentualnym kosztami budowania systemu opartego w całości na pamięci RAM również traci na znaczeniu, z uwagi na coraz niższe ceny kości pamięci.

Dane rozmieszczane są na dostępnych serwerach uruchomionych w sieci. Jeśli na którymś serwerze zachodzi taka konieczność dane przesyłane są dalej do innego serwera. Z tego też względu pytanie o konkretne dane może być zaadresowane do niewłaściwego odbiorcy. Zapytania także są przesyłane pomiędzy serwerami, aż znaleziony zostanie właściwy serwer.

Funkcje klienta

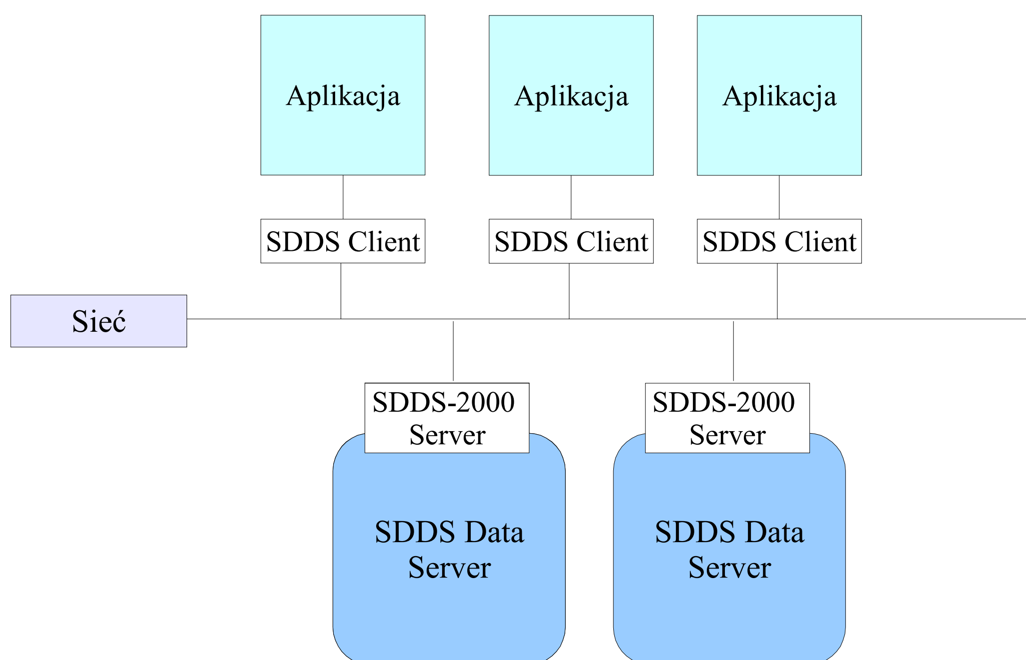
- podział danych pomiędzy serwerami dzieje się bez wiedzy i udziału klienta
- klient nie łączy się z żadnym centralnym katalogiem danych
- zawiera mniej lub bardziej aktualną informację o aktualnym wyglądzie plików danych
- może wysyłać informacje o wyszukiwaniu lub aktualizacji do niewłaściwego serwera

Funkcje serwera

- może przesyłać zapytania o konkretne dane do innych serwerów
- może wysyłać do klienta informację aktualizującą lokalny plik zawierający informacje o

układzie danych w systemie

- serwery stosują skanowanie równoległe w poszukiwaniu danych



Ilustracja 29. Architektura systemu opartego na SDDS-2000

7.2.3. Uwagi

System SDDS-2000 jest ciągle rozwijany. Z tego względu jest bardzo niestabilny oraz zawiera sporo błędów. Bez przerwy jednak prowadzone są prace nad nowymi wersjami (obecnie SDDS-2004). Dodawana jest nowa funkcjonalność, czynione są starania aby poprawić występujące i odkryte błędy. W związku z tym jest nadzieja, iż niedługo powstanie wersja systemu pozwalająca przetestować faktycznie działanie systemu i sprawdzić jego przydatność w praktyce.

Na postawie:
[L1]

*7.3. SDDS-LH**

W tym podrozdziale opisana zostanie implementacja struktury SDDS LH* wykonana przez autorów oraz przyczyny jej powstania.

7.3.1. Wstęp

Zależność od zewnętrznych technologii stwarza istotne zagrożenie dla każdego projektu, zwłaszcza jeśli technologia jest nowa i nie do końca przetestowana w zastosowaniach praktycznych. W przypadku produktów akademickich takie ryzyko należy określić jako bardzo duże. Dlatego już na etapie konspektu, usterki w systemach zewnętrznych zostały wymienione jako główne zagrożenie dla powstania pracy i jak się niestety później okazało, obawy były w pełni uzasadnione.

W chwili rozpoczęcia prac nad indeksem dla systemu OOP, istniały już dwie wersje systemu – SDDS-2000 Alego Wan Diene oraz SDDS-2003, będący udoskonaleniem 2000 wykonanym przez doktoranta Riada Mokameda. Wersja 2000 posiadała jedynie operacje wstawiania oraz wyszukiwania rekordów, więc została skreślona po otrzymaniu od profesora Witolda Litwina (opiekuna obu autorów) wersji 2003. Ulepszona wersja miała zawierać dodatkowo usuwanie i aktualizację rekordów.

Liczne testy przeprowadzone na SDDS-2003 zakończyły się jednak fiaskiem. W przeciwieństwie do wersji 2000, SDDS-2003 po wyszukaniu rekordu nie zwracał pola danych rekordu. Problem został zgłoszony do autora, ale jego potwierdzenie i usunięcie zajęło około roku – z tego powodu zdecydowano, że prace będą prowadzone na wersji SDDS-2000. Jak wcześniej wspomniano, ta wersja nie zawierała obsługi usunięć oraz modyfikacji rekordu, a operacje te były bardzo istotne w projekcie indeksu dla OOP. Dodatkowo system nakładał liczne niewygodne ograniczenia – maksymalna długość pola niekluczowego to 100 bajtów, maksymalna wartość klucza 1000000 (zaledwie jeden milion). Przyczyny pierwszego ograniczenia można tłumaczyć wadami wykorzystanego protokołu UDP, drugie ma swoje źródło prawdopodobnie w niezbyt oszczędnej implementacji. Kolejną słabością jest zastosowanie w niektórych miejscach blokad pętlowych (ang. spin lock) jako metod synchronizacji. Dla uściślenia blokada pętlowa wygląda następująco:

while (warunek)
do nothing;
end while;

Jest to najprostszy istniejący zamek, przydatny w sytuacjach, kiedy wątki zatrzymywane są na takiej blokadzie na bardzo krótkie okresy czasu (pozwala to uniknąć narzutu spowodowanego zmianą kontekstu). Jeśli jednak wątek oczekuje długo to zużywa on cały przydzielany mu czas

procesora (nie przekazuje dobrowolnie sterowania). W przypadku SDDS-2000 i 2003 mamy do czynienia z drugim przypadkiem – niektóre wątki przez cały czas działania programu stoją na blokadach pętlowych, efektywnie zabierając większość czasu procesora. Spowalnia to działanie innych aplikacji, ale co istotniejsze, także pracujących wątków SDDS. Zauważalnie wpływa to na (słabe) wyniki testów wydajności. Usterka nie jest ciężka do usunięcia i została przez autorów wyeliminowana w wersji SDDS-2003. Do SDDS-2000 modyfikacji wprowadzić się nie udało, gdyż oryginalny kod źródłowy nie został zachowany.

Pomimo napotkanych przeszkód prace nad systemem opartym o SDDS-2000 były kontynuowane. Wspomniane braki zmusiły jednak autorów do stworzenia pewnych mechanizmów uzupełniających SDDS-2000 o wymaganą w tej pracy funkcjonalność (co opisano już w części poświęconej indeksowi opartemu na SDDS-2000). Powstały system (pod względem wydajności jak i architektury) nie zadowolił jednak autorów. W efekcie podjęta została decyzja o implementacji własnej struktury SDDS (nazwanej SDDS-LH*).

7.3.2. Założenia SDDS-LH*

W odróżnieniu od SDDS-2000 nowa implementacja wykorzystuje jako podstawę algorytm LH*, opisany szeroko w jednym z wcześniejszych rozdziałów. Do implementacji wybrano język Java, głównie ze względu na prostotę implementacji i zgodność z API systemu OfficeObjects Portal, co bardzo ułatwiło integrację systemów.

Po podjęciu tych dwóch najbardziej strategicznych decyzji przyszedł czas na dobranie odpowiednich środków do implementacji koncepcji zawartej w algorytmie LH*.

*Architektura SDDS-LH**

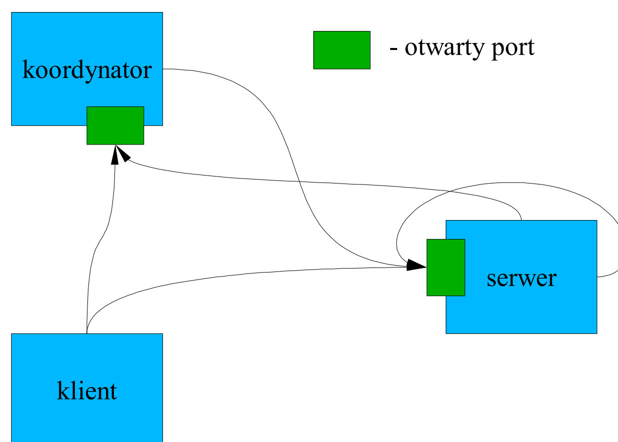
Ogólna architektura jest zbliżona do tej zastosowanej w algorytmie SDDS-2000. Wyróżnione są trzy oddzielne aplikacje:

1. Serwer – to aplikacja fizycznie przechowująca kubełki z danymi, w sieci może być ich dowolnie wiele.
2. Koordynator podziałów – pełni rolę opisanego w algorytmie LH* koordynatora podziałów, posiada aktualny obraz pliku oraz tablice z adresami wszystkich kubełków w pliku.
3. Klient – jest to właściwie tylko moduł dołączany z wygodnym API dla użytkownika końcowego, wywołania API tłumaczone są przez klienta na komunikację z serwerami i koordynatorem.

Komunikacja prowadzona między wszystkimi aplikacjami prowadzona jest poprzez protokół TCP/IP. Co prawda większość rozważań przedstawionych w pracach poświęconych algorytmowi LH* dotyczy wiadomości rozgłoszeniowych. Wiadomości rozgłoszeniowe są w tej chwili powszechnie obsługiwane jedynie przez protokół UDP, a ten akurat do tego typu zastosowań nie wydaje się właściwy. Teoretycznie protokół UDP ma mniejszy narzut, a co za

tym idzie może osiągać większe przepustowości niż TCP korzystając z takiego samego łącza danych, i jak wspomniano, posiada szczególnie wygodne z punktu widzenia algorytmu LH* wiadomości rozgłoszeniowe. UDP jest jednak protokołem zawodnym, pozbawionym kontroli. A co za tym idzie, każdy z pakietów UDP może podczas transmisji między komputerami zostać zgubiony. W przypadku SDDS, może to oznaczać dużą liczbę nieuzasadnionych błędów i przestojów, a dodatkowo ogranicza pracę systemu do jednego segmentu sieci. Co prawda nie trudno sobie wyobrazić system potwierdzeń i powtórzeń wiadomości który sprawi, że wysłane wiadomości, kiedyś dotrą do celu, ale należy jeszcze wziąć pod uwagę, że dłuższe wiadomości są szatkowane na kilka mniejszych mieszczących się w jednostce transportowej protokołu (dla protokołu IP to niecałe 1500 bajtów). Każda z części wiadomości może w tym wypadku zostać zagubiona, a kolejność części pozamieniana, więc potrzebne by były kolejne systemy kontroli. Widać z tego, że budowa niezawodnego protokołu komunikacyjnego na bazie UDP to zadanie pracochłonnością znacznie przekraczające ewentualne korzyści, stąd wybór TCP.

Zarówno serwer jak i koordynator mogą przyjmować połączenia przychodzące (mają otwarte porty). Schemat możliwych połączeń pomiędzy trzema rodzajami aplikacji przedstawiono na ilustracji 30.



*Ilustracja 30. Schemat nawiązywanych połączeń w SDDS-LH**

7.3.3. Koordynator

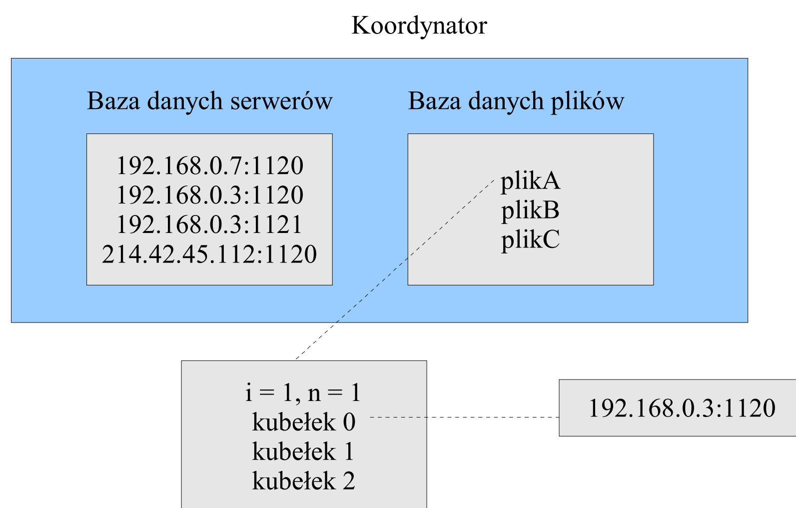
Rolą koordynatora w systemie SDDS-LH* jest przechowywanie bieżącego stanu pliku, kontrola podziałów kubełków, utrzymywanie bazy dostępnych serwerów oraz translacja logicznych adresów kubełków na adresy fizyczne.

Pod pojęciem fizycznego adresu w SDDS-LH* rozumiana jest klasa Javy w której skład

wchodzą: klasa adresu internetowego oraz numer portu. Klasa adresu jest klasą z JRE i obsługuje adresy w postaci IPv4, jak i IPv6. Taka definicja adresu pozwala na umieszczenie dowolnej ilości serwerów na jednym fizycznym komputerze (istotne dla testów).

Koordynator posiada dwie wewnętrzne bazy danych. Bazę danych dostępnych serwerów oraz bazę danych plików. Baza danych serwerów składa się jedynie z adresów i wykorzystywana jest do wyboru serwera pod nowy kubełek.

Baza danych plików posiada nieco bardziej skomplikowaną strukturę. Dla każdego pliku przechowywany jest jego aktualny stan (parametry *i* i *n*) oraz adresy wszystkich kubełków (ilustracja 31).



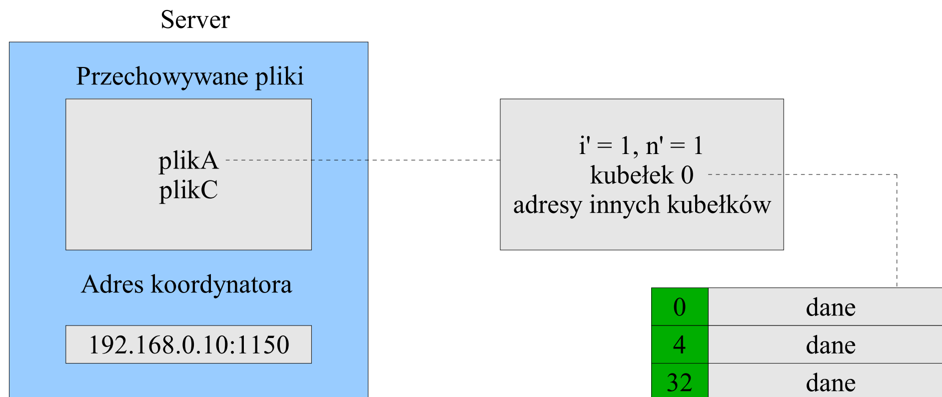
Ilustracja 31. Bazy danych koordynatora

7.3.4. Serwer

Serwer jest jednostką zdolną do przechowywania fizycznych kubełków pliku. Każdy serwer może przechowywać kubełki z dowolnej ilości plików oraz dowolną ilość kubełków z jednego pliku (czyli przestrzeń dostępna dla pliku nie skończy się wraz z zapełnieniem wszystkich kubełków na wszystkich dostępnych serwerach, jak ma to miejsce w SDDS-2000).

Oprócz fizycznych kubełków dla danego pliku, serwer przechowuje także swój obraz pliku, oraz adresy innych serwerów. W klasycznym algorytmie LH*, serwery nie przechowują obrazu, a jedynie poziom kubełka. Takie podejście pozwala dodatkowo zmniejszyć ilość błędów adresowania nie podnosząc przy tym specjalnie kosztów. Kubełki do przechowywania rekordów stosują mapy indeksowane kluczem zaimplementowane na drzewach czerwono-

czarnych (TreeMap).

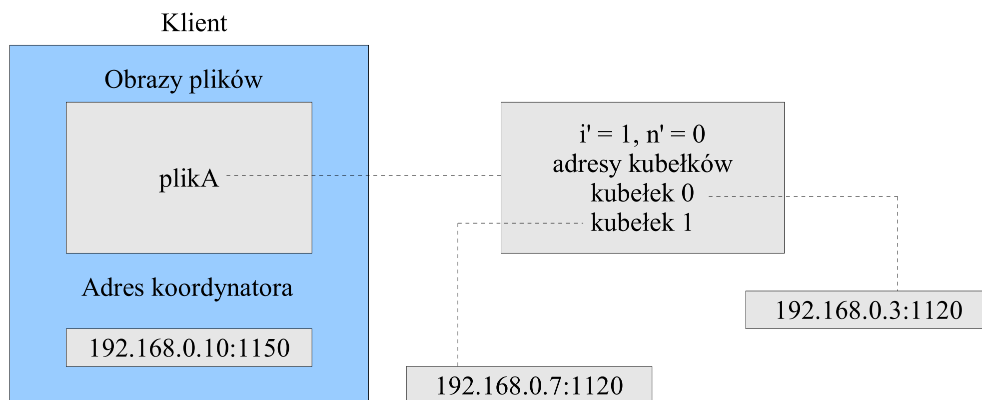


Ilustracja 32. Dane przechowywane przez serwery

Serwer aby rozpocząć działanie musi się zarejestrować u koordynatora. W tym celu musi znać jego adres. Adres koordynatora można określić ręcznie albo serwer spróbuje odszukać dostępnego koordynatora w sieci lokalnej przy użyciu wiadomości rozgłoszeniowych.

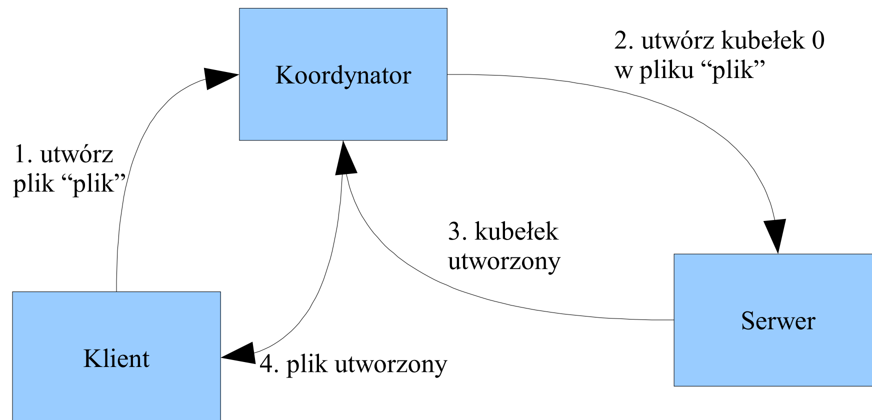
7.3.5. Klient

Do informacji przechowywanych przez klienta należą: obraz pliku, pozyskane adresy kubełków oraz adres koordynatora. Sposób pozyskiwania adresu koordynatora jest taki sam jak dla serwera.



Ilustracja 33. Dane przechowywane przez klienta

7.3.6. Tworzenie plików



Ilustracja 34. Operacja tworzenia pliku

Tworzenie nowych plików odbywa się z inicjatywy klienta. Klient kontaktuje się z koordynatorem zlecając operację utworzenia pliku, wraz z niezbędnymi parametrami – nazwą pliku i wielkością kubełków, dodatkowo możliwe jest podanie progów dla algorytmu kontroli wypełnienia. Jeśli plik o podanej nazwie jeszcze nie istnieje, to serwer tworzy odpowiedni wpis w bazie danych plików, a następnie tworzy kubełek 0 na jednym z zarejestrowanych serwerów. Po wykonaniu operacji serwer informuje klienta o rezultacie – ilustracja 34.

7.3.7. Operacja wyszukiwania rekordu

Operacja wyszukiwania rekordu inicjowana jest przez klienta, jako parametry przekazywane są nazwa pliku oraz klucz rekordu. Jeśli klient nie posiada danych dla wybranego pliku, to klient zwraca się do koordynatora z prośbą o adres kubełka 0. Tak samo postępuje jeśli jego kalkulacje adresu wskażą na kubełek dla którego fizycznego adresu nie ma w podręcznej strukturze. Jeśli jakieś informacje o pliku są już przechowywane u klienta to na pewno wśród nich są parametry i' i n', klient rozpoczyna więc wyszukiwanie od własnej kalkulacji adresu, po czym nawiązuje połączenie z wskazanym przez obliczenia kubełkiem. Pomijając ewentualność awarii na którą SDDS-LH* nie jest przygotowany, mogą zajść następujące przypadki:

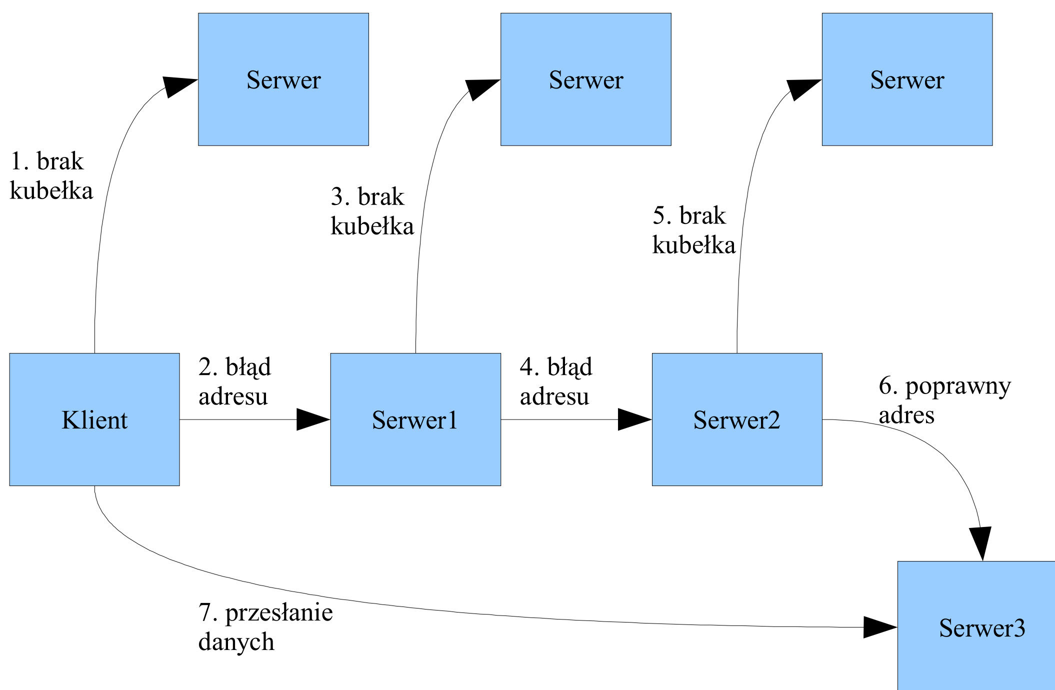
a) zaadresowany został właściwy kubełek, rekord jest wyszukiwany i jeśli istnieje wysyłany do klienta
b) w skutek rozwoju pliku, wyszukiwany element został przeniesiony do innego kubełka
c) w efekcie wykonanych złączeń, zaadresowany kubełek nie istnieje już na tym serwerze.

Jeśli wystąpi a) to jest to koniec procesu wyszukiwania. W przypadku c) serwer prawdopodobnie nie wie nic o bieżącym stanie pliku, więc najlogiczniejszym rozwiązaniem jest

zresetowanie obrazu klienta do wartości $i' = 0$, $n' = 0$. Przyjęcie takich wartości nie spowoduje więcej tego typu błędów adresowania, gdyż serwer z kubełkiem 0 zawsze pozostaje ten sam.

W przypadku b) serwery analogicznie do operacji klienta adresują kubełki zgodne z ich obrazem pliku. Operacja ta przebiega rekurencyjnie aż do odnalezienia właściwego kubełka. Dzięki temu serwery aktualizują swój obraz i popełniają jeszcze mniejszą ilość przekazywania zapytania niż w klasycznym algorytmie LH*. Po odnalezieniu kubełka pierwszy skontaktowany serwer zwraca klientowi zaktualizowany stan pliku, numer kubełka oraz jego fizyczny adres. Klient sam kontaktuje się ze wskazanym serwerem i odbiera od niego dane rekordu. Takie zachowanie ma na celu uniknięcie przekazywania dużej ilości danych między serwerami (albowiem wielkość pola danych rekordu w SDDS-LH* nie jest ograniczona przez implementację, dla porównania SDDS-2000 obsługuje maksymalnie 100 bajtowe rekordy). Serwery tak samo jak klienci mogą zwrócić się z zapytaniem do serwera na którym danego kubełka już nie ma. Oczywiście nie kasują one swojego obrazu pliku do wartości $i' = 0$, $n' = 0$, a jedynie do bezpiecznych wartości wynikających z numeru przechowywanego kubełka. Jak wiadomo z rozdziału o algorytmie LH* wystąpią co najwyżej dwa przekazania dalej zapytania, do tego dochodzą trzy możliwe zapytania wysłane do serwerów nie posiadających adresowanego kubełka. Na koniec należy doliczyć jeszcze połączenie z przekazywaniem danych. Cała operacja przedstawiona jest na ilustracji 33 (strzałki wskazują kierunek nawiązywanych połączeń, komunikacja jest dwustronna).

W SDDS-2000 pole niekluczowe w rekordzie miało postać 100 bajtowej tablicy bajtów. SDDS-LH* w celach zachowania kompatybilności także pozwala wstawiać i odbierać rekord jako tablicę bajtów (ale nie ograniczając jej rozmiaru), a dodatkowo pozwala wstawiać i odbierać jakikolwiek serializowalny obiekt Javy. Zwracanym typem jest wtedy typ Object, który następnie należy zrzutować na pożądany typ (podobnie jak w obsłudze kolekcji).



Ilustracja 35. Procedura wyszukiwania rekordu

7.3.8. Operacja wstawiania rekordu

Operacja wstawiania jest bardzo zbliżona do operacji wyszukiwania – występuje taka sama procedura poszukiwania właściwego kubełka, jedyna różnica dotyczy kierunku przekazywania danych rekordu między klientem, a właściwym serwerem. Po wykonaniu wstawienia serwer sprawdza czy kubełek nie przekroczył pojemności i ewentualnie informuje koordynatora o przepełnieniu.

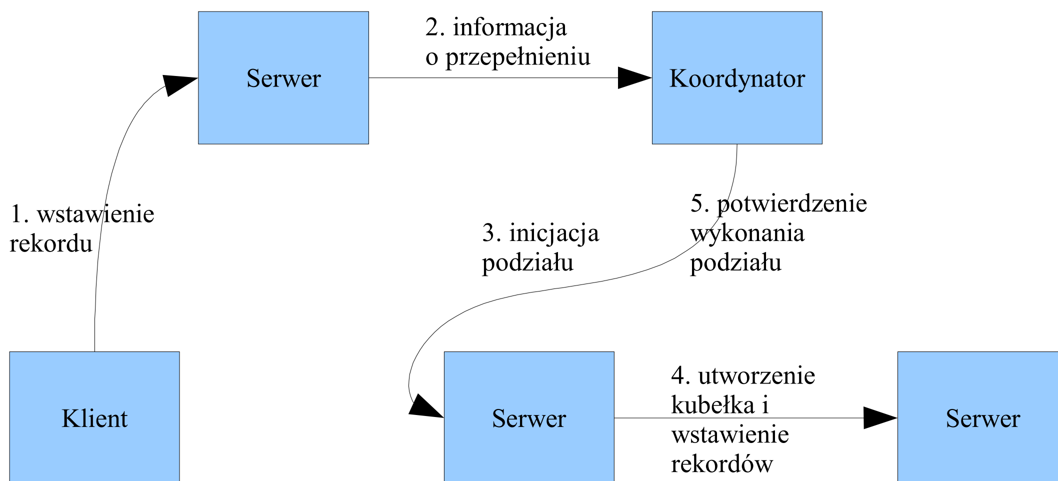
7.3.9. Operacja usuwania rekordu

Operacja wykonywana jest analogicznie do operacji wyszukiwania i wstawiania. Jedynym nowym elementem jest zastosowanie dolnego progu wypełnienia – jeśli wypełnienie zaadresowanego kubełka spadnie poniżej progu to informuje on koordynatora o tym fakcie, a ten może podjąć dalsze kroki (dokonać złączenia kubełków).

7.3.10. Podział kubełków

W momencie wystąpienia przepełnienia w kubełku, serwer na którym jest on przechowywany informuje koordynatora o tym fakcie. Koordynator posługując się wartościami i i n wskazuje kubełek, który powinien wykonać podział. Koordynator wybiera także (posługując się bieżącym obciążeniem serwerów) wybiera spośród zarejestrowanych serwerów ten na którym ma zostać utworzony nowy kubełek. Koordynator wysyła wiadomość do serwera z dzielonym

kubełkiem przekazując mu adres serwera pod nowy kubełek. Dzielący serwer wysyła do drugiego serwera wiadomość o utworzeniu kubełka pod podział, a następnie przesyła do niego rekordy odfiltrowane funkcją mieszającą na odpowiednim poziomie. Po zakończeniu podziału serwer dzielący wysyła potwierdzenie do koordynatora. Do czasu otrzymania potwierdzenia koordynator wstrzymuje kolejne podziały w danym pliku, aby zapewnić ich synchroniczność (ilustracja 32).



Ilustracja 36. Procedura dokonywania podziałów

7.3.11. Złączenia kubełków

Złączenie kubełków jest operacją odwrotną do podziału i wygląda podobnie. Koordynator wysyła do odpowiedniego kubełka wiadomość inicjującą złączenie (adres wyliczany jest z wartości i i n). Usuwany kubełek jest w stanie określić adres swojego rodzica, więc dalsze informacje od koordynatora nie są konieczne, po skontaktowaniu z rodzicem wszystkie rekordy są do niego przesyłane, a kubełek usuwany. Po zakończeniu operacji do koordynatora wysyłane jest potwierdzenie mające zapewnić synchroniczność podziałów i złączeń.

7.3.12. Kontrola wypełnienia

Możliwe jest stworzenie pliku z kontrolą wypełnienia. Do tego celu należy podać progi wypełnienia w jakich ma się znajdować faktyczne wypełnienie. Kontrola wypełnienia zaimplementowana jest zgodnie z algorytmem przedstawionym w rozdziale o LH*. W momencie przepełnienia kubełka do koordynatora wysyłana jest informacja o bieżącym wypełnieniu przepełnionego kubełka, a koordynator szacuje wypełnienie dla całego pliku i porównuje z podanym progiem. Inicjacja podziału kubełka następuje dopiero jeśli wyliczone wypełnienie przekracza zadany próg.
Jeśli zachodzi potrzeba korzystania ze złączeń kubełków należy określić także dolny próg

wypełnienia. W takiej sytuacji w kubełkach umieszczony jest dodatkowy parametr – odpowiednio zmodyfikowany dolny próg wypełnienia. Jeśli w skutek kasowania rekordów wypełnienie spadnie w kubełku poniżej progu to serwer tak jak w powyższym przypadku wysyła do koordynatora odpowiednią informację. Także i w tej sytuacji następuje oszacowanie wypełnienia dla całego pliku i w przypadku spadku poniżej zadanego progu uruchamiana jest procedura złączenia kubełków.

7.3.13. Zapytania równoległe

Zapytania równoległe nie są tak proste w implementacji i tanie w użyciu przy użyciu protokołu TCP/IP, a dodatkowo ze względu na semantykę kluczy (a właściwie jej brak) nie zdecydowano się na implementację tego typu operacji. W wykonywanej na bazie SDDS-LH* implementacji indeksu klucze są przetwarzane przez funkcję mieszającą nie zachowującą kolejności co powoduje utratę jakiejkolwiek informacji, którą klucz mógł zawierać. Jedyną sensowną operacją wyszukiwania w takim indeksie to wyszukiwanie w rodzaju *WHERE zmienna = wartość*, a że SDDS-LH* potrafi w jednym rekordzie przechowywać referencję do wszystkich obiektów spełniających ten warunek to nigdy nie będzie potrzebne wyszukiwanie większej ilości rekordów na raz.

7.3.14. API SDDS-LH*

SDDS-LH* wymaga do działania uruchomienia jednego procesu koordynatora oraz dowolnej ilości procesów serwerów.

Wykorzystanie dostępu do SDDS w dowolnej aplikacji pisanej Javie wymaga jedynie dołączenia odpowiedniego pakietu.

Zainicjowanie klienta SDDS wykonuje się jednym z czterech konstruktorów:

- SDDSConnection() – wyszukanie koordynatora w sieci lokalnej na domyslnym porcie
- SDDSConnection(int port) – wyszukanie koordynatora w sieci lokalnej na podanym porcie
- SDDSConnection(InetAddress address) – połączenie z serwerem o podanym adresie bez wyszukiwania, port domyślny
- SDDSConnection(InetAddress address, int port) - połączenie z serwerem o podanym adresie bez wyszukiwania, port podany

Utworzony obiekt jest związany z jedną strukturą SDDS – w jednej aplikacji można korzystać z wielu różnych struktur rozmieszczonych np. w różnych sieciach. Wszystkie podane poniżej operacje są metodami utworzonego obiektu SDDSConnection.

Utworzenie pliku wykonuje się operacjami:

- void createFile(String name, int bucketSize) – stworzenie pliku o podanej nazwie i wielkości

kubełka

- void createFile(String name, int bucketSize, float minThreshold, float maxThreshold) – tak jak wyżej i dodatkowo włączenie mechanizmu kontroli wypełnienia

Można także pobrać listę dostępnych plików operacją:

- String[] listFiles()

Wstawianie wykonuje się dwoma operacjami w zależności od dostarczanych argumentów:

- void insertRecord(String fileName, int key, byte[] arr) – jeśli argumentem ma być, tak jak w SDDS-2000, tablica bajtów
- void insertRecord(String fileName, int key, Object ob) – jeśli wstawiany jest obiekt Javy

Usuwanie jest bardzo proste:

- void deleteRecord(String fileName, int key)

Do wyszukiwania obiektów służą dwie funkcje API:

- byte[] retrieveRecord(String fileName, int key) – pole danych rekordu zostaje zwrócone w postaci tablicy bajtów
- Object retrieveObjectRecord(String fileName, int key) – pole danych zostaje odtworzone jak obiekt Javy

Wszystkie operacje potrafią wyrzucić wyjątki dziedziczące z klasy SDDSException, które dotyczą błędów w rodzaju, wyszukiwania w nieistniejącym pliku, wstawiania drugiego rekordu z tym samym kluczem, itp.

7.3.15. Wielowątkowość

SDDS-LH* został tak skonstruowany aby w pełni obsługiwać wielowątkowość. W przypadku serwera i koordynatora mogą one obsługiwać dowolną ilość połączeń na raz. Każde połączenie obsługiwane jest przez osobny wątek, a aby uniknąć powstania niespójności w danych w każdym programie wydzielono liczne sekcje krytyczne. Jako metodę synchronizacji wykorzystano obecne w Javie monitory, a tam gdzie ich zastosowanie powodowało zbyt gruboziarniste blokady skorzystano z semaforów binarnych (własna produkcja na potrzeby SDDS-LH*).

Także moduł kliencki posiada wyodrębnione sekcje krytyczne co pozwala na obsługiwanie wielowątkowych aplikacji użytkownika końcowego (między innymi obsługa wielu wątków korzystających z tego samego połączenia SDDS).

7.3.16. Podsumowanie

Na koniec przedstawiamy porównanie implementacji SDDS-2000 autorstwa zespołu prof. Litwina oraz SDDS-LH*, wykonanej przez autorów.

Środowisko sieciowe

System SDDS-2000 może działać jedynie w obrębie sieci lokalnej, SDDS-LH* może korzystać ze wszystkich serwerów internetowych. SDDS-LH* może także tworzyć kilka kubełków na pojedynczym serwerze oraz wiele serwerów na jednym fizycznym komputerze.
Przewaga: SDDS-LH*

Środowisko systemu operacyjnego

SDDS-2000 został napisany w C z wykorzystaniem bibliotek systemowych Windows i może funkcjonować jedynie pod tym systemem, SDDS-LH* zaimplementowany jest w języku Java i może być uruchamiany na każdym systemie posiadającym implementację maszyny wirtualnej Javy.
Przewaga: SDDS-LH*

Operacje podstawowe

Jak napisano wcześniej SDDS-2000 nie obsługuje aktualizacji oraz usuwania rekordów. Obie operacje są obecne w SDDS-LH*.
Przewaga: SDDS-LH*

Operacje zaawansowane

SDDS-LH* nie obsługuje wyszukiwania równoległego. SDDS-2000 ma operację równoległego wyszukiwania, choć jej sens jest mocno ograniczony przez zawężenie przestrzeni kluczy. Teoretycznie jako klucze w algorytmie RP* zastosowanym w SDDS-2000 mogą być stosowane dowolne zbiory na których jest określony całkowity porządek elementów. Operacja była by bardzo interesująca gdyby wykonywać zapytania w przedziale określonym na przykład na całej przestrzeni ciągów znakowych. W takiej postaci, jaka jest obecna w SDDS-2000, możliwości jej praktycznego użycia są znacznie mniejsze, ale niemniej jednak operacja jest dostępna.
Przewaga: SDDS-2000

Kontrola wypełnienia oraz złączenia kubełków

Jako, że SDDS-2000 nie ma ani kontroli wypełnienia, a jako że nie obsługuje usuwania elementów to nie ma złączeń kubełków, które można włączyć w SDDS-LH*.
Przewaga: SDDS-LH*

Przestrzeń kluczy

SDDS-2000 sztucznie ogranicza przestrzeń kluczy do przedziału 0-1'000'000, SDDS-LH* wykorzystuje dodatnią część typu int Javy. Jest to ponad 2000 razy więcej, a należy dodać, że zastosowanie 64-bitowego typu long nie wymagałoby zmiany więcej niż kilkudziesięciu linii kodu.
Przewaga: SDDS-LH*

Wielkość pola danych

SDDS-2000 nakłada bardzo poważne ograniczenie na długość pola danych – maksymalnie 100 bajtów. SDDS-LH* został tak pomyślany, aby długość pola danych była ograniczona jedynie fizyczną ilością pamięci dostępnej na komputerze, a więc teoretycznie nie ma ograniczeń.
Przewaga: SDDS-LH*

Wydajność

Co pokazane zostało w testach SDDS-LH* dzięki znacznie efektywniejszej implementacji, okazuje się znacznie szybszy od SDDS-2000.
Przewaga: SDDS-LH*

Dodatkowo można jeszcze stwierdzić, że API SDDS-LH* jest znacznie wygodniejsze w użyciu to wpływa na efektywność programowania z jego użyciem, ale ten punkt nie będzie brany pod uwagę. Widać jednak z powyższego porównania, że przewaga SDDS-LH* jest niezaprzeczalna – jest on lepszy w siedmiu z ośmiu kategorii. Niemniej jednak trudno uznać SDDS-LH* za coś więcej niż akademicka zabawkę, brak algorytmów poprawiających niezawodność wyklucza jego praktyczne zastosowania. W przewidzianym dla niego zadaniu – stworzeniu indeksu dla systemu OOP, sprawdził się jednak doskonale.

7.4. Wrapper systemu OfficeObjects Portal

Jest to zestaw interfejsów Javy pozwalających czytanie informacji w abstrakcyjnej bazie danych. Z Punktu widzenia klienta interfejsy te mogą być stosowane bez żadnych zmian dla dowolnego systemu. Każdy serwer jednak musi dostarczać własnej implementacji dla odpowiednich interfejsów.
Pakiet ten pozwala na czytane metadanych o:

- klasach
- atrybutach klas
- asocjacjach między klasami

oraz zawartości bazy danych, czyli konkretnych wartości atrybutów w obiektach.

Pakiet com.mt.abstractdatabase udostępnia następujące interfejsy:
AdAssociacion – definiuje informacje o asocjacjach pomiędzy klasami.
AdAttribute – definiuje informacje o atrybutach danej klasy
AdAttributeValue – definiuje wartości atrybutów dla konkretnego obiektu
AdClass – definiuje informacje o klasach istniejących w Abstrakcyjnej Bazie Danych
AdCriterion – interfejs do tworzenia podstawowych kryteriów wyszukiwania obiektów
AdInteger – reprezentuje typ Integer
AdItem – podstawowy interfejs definiujący informacje o obiektach, atrybutach itp. Klasa implementująca ten interfejs powinna być abstrakcyjna (abstract).
AdLabel – definiuje informacje o etykiecie obiektu identyfikowanego przez AdUniqueID
AdLink – definiuje informacje o połączeniu pomiędzy obiektami
AdObject – definiuje informacje o konkretnym obiekcie (instancji klasy)
AdUniqueID – definiuje informacje o obiekcie będącym unikalnym identyfikatorem dowolnego bytu w bazie danych, a więc obiektu, asocjacji, klasy, referencji itp.
AdWrapper – dostarcza funkcjonalność do czytania informacji z bazy danych
AdWrapper.AdMetaData – dostarcza funkcjonalność do czytania meta informacji z bazy danych
AdWrapperSBQL – dostarcza metod niezbędnych do implementacji języka zapytań SBQL (Stack Based Query Language)

7.5. API indeksu dla systemu OOP

Podstawowym celem tej pracy magisterskiej było stworzenie interfejsu programistycznego pozwalającego na zastosowanie zewnętrznego systemu do indeksowania danych w OfficeObjects Portal. Jako repozytorium indeksów planowano zastosować system SDDS-2000 (Scalable Distributed Data Structures) będący implementacją idei haszowania liniowego opracowaną przez prof. Witolda Litwina.
Rozwiązanie to w zamierzeniu miało istotnie przyspieszyć pracę z OOP. Ze względu na swoją architekturę system OfficeObjects Portal jest dość powolny i nawet nieskomplikowane operacje dostępu do danych zabierają wiele czasu i zasobów. Wykorzystanie indeksu powinno w dość znaczny sposób przyspieszyć operacje wyszukiwania danych czyniąc system OOP znacznie wygodniejszym w użyciu.
Interfejs udostępniony programiście miał pozwalać na tworzenie nowych indeksów, oraz wstawianie do niego danych. Indeks nie jest przezroczysty dla użytkownika. Gdy zachodziła potrzeba skorzystania z indeksu należało odwołać się bezpośrednio do obiektu reprezentującego połączenie z SDDS w wyniku czego można było otrzymać referencje do obiektów z OOP.
Podczas prac nad interfejsem stwierdzono jednak, że system SDDS cechuje się niską

wydajnością i stabilnością, sporą ilością błędów a także niepełną funkcjonalnością. Doprowadziło to do podjęcia decyzji o stworzeniu własnej wersji SDDS. Z założeniu miała ona być wolna od wszystkich błędów pierwowzoru, oraz uzupełniona o nowe funkcje nie istniejące (jeszcze) w oryginale.

Podczas opracowywania architektury wzorowano się na systemie SDDS-2000 wytworzonym przez zespół prof. Litwina.

W rezultacie doprowadziło to do powstania dwóch wersji API, jednej współpracującej z SDDS-2000 oraz drugiej z własną wersją tego systemu. Ze względu na różnice w architekturze obu produktów niezbędne było inne podejście do projektowanego rozwiązania. Różnice wynikały z takich powodów jak:

- *język programowania* - SDDS-2000 stworzony jest z języku C, dostęp do OOP odbywa się poprzez biblioteki języka Java; wymagało to stworzenia pomostu pomiędzy oboma systemami wykorzystującego połączenia sieciowe. Własna implementacja SDDS stworzona jest z kolei w Javie- dzięki temu nie ma potrzeby tworzenia kolejnej warstwy wymiany danych pomiędzy oboma systemami.
- *ograniczenia pól danych* - SDDS-2000 posiada ograniczenie na niekluczowe pola danych wynoszące 100 bajtów. Wprowadza to konieczność zapisywania nie całej referencji do obiektu, lecz tyko danych tworzących tę referencję. Dodatkowo trzeba tworzyć osobny plik danych na rekordy o zdublowanej wartości kluczowej. W nowym SDDS pola danych nie są ograniczone.
- *długość nazwy pliku* - SDDS-2000 posiada ograniczenie na długość nazwy pliku indeksowego. Nie jest to może cecha uciążliwa i wymagająca istotnych zmian z architekturze API, jednak w nowym SDDS nie istnieje konieczność skracania nazw plików.

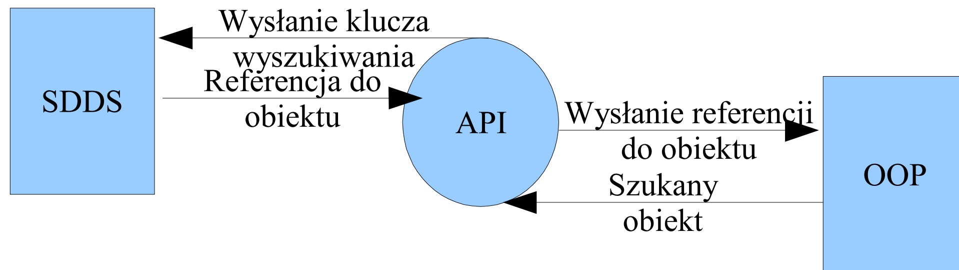
Pomimo podanych różnic ogólne założenia dla obu wersji API pozostają takie same. Dotyczy to zarówno procedury wymiany danych, jak i sposobu komunikacji z OOP.

W obu przypadkach API zostało stworzone w języku Java. Dostęp do źródła danych opiera się o interfejs com.mt.abstractdatabase stworzony przez Mariusza Trzaskę. Pozwala to na zastosowanie tego API z każdym systemem implementującym com.mt.abstractdatabase (w przypadku wersji dla SDDS-2000 niezbędne będą niewielkie modyfikacje). Dla OOP został stworzony odpowiedni pakiet przez twórców OfficeObjects Portal.

Indeks jest tworzony ręcznie. Tak samo aktualizacje danych wymagają ręcznego uruchomienia odpowiednich funkcji przez programistę.

Idea działania indeksu dla obiektów z OOP polega na pobraniu unikalnego identyfikatora określającego w sposób jednoznaczny obiekt istniejący w systemie OOP. Następnie taki identyfikator jest przesyłany do systemu SDDS, oraz indeksowany na podstawie zhaszowanej wartości wybranego atrybutu. Proces wyszukiwania polega na podaniu wyszukiwanych wartości atrybutu, które po przetworzeniu funkcją mieszającą służą do odszukania referencji obiektów w indeksie SDDS. Na podstawie odszukanych referencji z systemu OOP pobierane są

faktyczne obiekty.



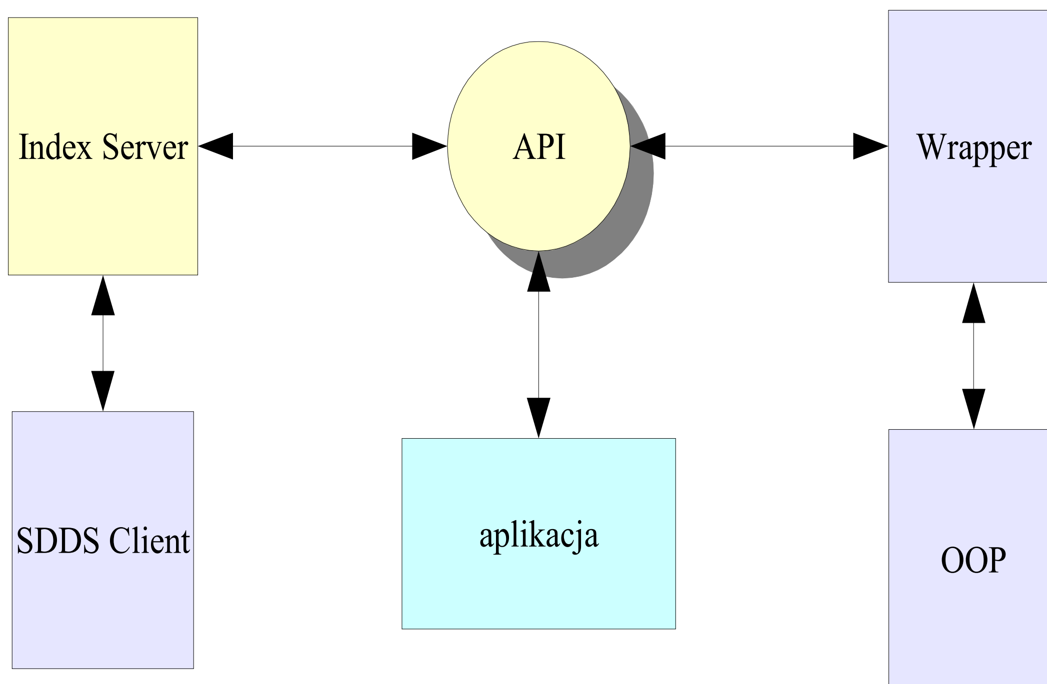
Ilustracja 37. Zasada działania indeksu

7.5.1. Indeks zbudowany na systemie SDDS-2000

API oparte o system SDDS-2000 było pierwotnie planowaną wersją bibliotek. Niestety na skutek ewidentnych braków tego systemu powstała równoległa wersja oparta o autorski projekt systemu SDDS. Mimo to wersja dla SDDS-2000 jest w pełni funkcjonalna i oferuje podstawowe możliwości tworzenia indeksów.

Architektura API dla SDDS-2000 jest rozbudowana. Z uwagi na fakt iż API na być przygotowane w języku Java, a SDDS napisany jest w języku C niezbędne było stworzenie pośrednika służącego do komunikacji pomiędzy oboma pakietami. Komunikację zrealizowano za pośrednictwem gniazd sieciowych. Funkcje SDDS są wywoływane za pośrednictwem interfejsu SDDSClient dostarczonego wraz z pakietem.

Komunikacja z OOP została zrealizowana na podstawie interfejsu com.mt.abstractdatabase.



Ilustracja 38. Schemat współpracy aplikacji z SDDS-200 oraz OOP

IndexServer

IndexServer odpowiedzialny jest za pośrednictwo w wymianie danych pomiędzy API a SDDS. Jest to program napisany w C. W chwili uruchomienia łączy się on SDDS (w razie niepowodzenia zostanie wypisana odpowiednia informacja), a następnie przechodzi w stan oczekiwania na połączenie z klientem. Aplikację należy uruchamiać na maszynie na której uruchomiony jest także klient SDDS.

Inicjując obiekt reprezentujący połączenie z SDDS, API łączy się właśnie z IndexServer. W wyniku przesyłanych komend oraz potrzebnych danych IS uruchamia odpowiednie procedury zarządzania danymi na SDDS.

Poza przekazywaniem żądań z API do SDDSClient, IndexServer ma tez inne zadania. Zostały w nim zrealizowane procedury, które są niezbędne do wykonania z uwagi na braki funkcjonalności SDDS-2000. Przede wszystkim zawarty jest moduł bufora służącego do budowania indeksu. Struktura rekordów przechowywanych w systemie SDDS zostanie opisana w części dotyczącej budowy indeksu. Istotny w tej chwili jest fakt, że w przypadku wystąpienia kolizji niezbędna jest aktualizacja wstawionego rekordu (np. w celu zapisania wskaźnika do pliku nadmiarowego). Jak już wielokrotnie o tym pisano SDDS-2000 nie obsługuje operacji usuwania oraz modyfikacji rekordów. Jedynym dostępnym rozwiązaniem było zbudowanie całego indeksu w buforze, tak aby po rozpoczęciu wstawiania rekordów do SDDS nie zaszła już

konieczność ich modyfikacji.

Szczegóły implementacji

Do wymiany danych i identyfikacji żądań pomiędzy API oraz IndexServer zaprojektowano protokół służący do wywoływania odpowiednich funkcji przez API w module klienckim SDDS oraz do wymiany informacji pomiędzy oboma pakietami. Transmisja została zrealizowana za pomocą gniazd sieciowych (Sockets) na podstawie protokołu TCP/IP.

W momencie wywołania określonej funkcji w API do IndexServer zostanie wysłany kod informujący o typie zadania. Następnie zależnie od konkretnego zadania nastąpi obustronna wymiana informacji przerywana potwierdzeniami o sukcesie danej transmisji. Transmisja kończy się zwróceniem kodu końcowego sukcesu przez IndexServer, lub informacji o błędzie oraz kodu błędu.

Dane przesyłane są w postaci tablicy bajtów. Z tego względu po każdym ciągu danych trzeba wysłać informację o końcu przesyłania. Niezbędne jest to aby "poinformować" IndexServer, że odebrane bajty stanowią całość (np. była to nazwa pliku lub klucz wyszukiwania).

Architektura API , w odróżnieniu od tej stworzonej na potrzeby własnej implementacji SDDS musi być bardziej rozbudowana. Niezbędne jest uwzględnienie takich elementów jak transmisja danych. Istotne różnice w implementacji pomiędzy wersją API dla SDDS-2000 oraz własnej implementacji SDDS to:

- Przesyłanie danych: w tym przypadku wszystkie elementy przesyłania danych należy zamieścić w API. Trzeba pamiętać o typach danych, różnicach implementacyjnych pomiędzy C oraz Javą, obsłudze błędów itp.
  - API opiera połączenia sieciowe na klasie java.io.Socket.
  - Jako kanał komunikacyjny stosuje się java.io.DataInputStream oraz java.io.DataOutputStream.
  - Dane wysyła/odbiera się za pomocą metod readByte, sendByte, readBytes, sendBytes, ewentulanie readInt, sendInt.
  - Poza samym danymi musi też być przesyłana wielkość danych. W przypadku gdy obie strony komunikacji zrealizowane są np. w języku Java możliwe było by zastosowanie istniejących metod typu writeObject, pozwalających nie martwić się o niezgodność przesyłanego typu pomiędzy aplikacjami napisanymi w różnych językach. W tym przypadku jednak trzeba poinformować IndexServer o fakcie, że określony ciąg danych stanowi jeden cały obiekt i kolejne wysyłane dane są niezależną porcją informacji.
- W wersji dla SDDS-2000 funkcję haszującą umieszczono w IndexServer,

Żądania API

- SDDS_CREATE_INDEX – żądanie utworzenia nowego pliku indeksowego w SDDS
- SDDS_SEARCH_INDEX – żądanie rozpoczęcia wyszukiwania

Błędy

W przypadku wystąpienia błędu API zwraca wyjątek SDDSException dziedziczący z klasy Exception. Informacja przenoszona przez błąd zależna jest od tego jaki wystąpił błąd i co zostało zwrócone do API z OfficeObjects Portal lub SDDS (IndexServer).

IndexServer zwraca błędy w dwóch fazach: w pierwszej przesyłana jest informacja o miejscu wystąpienia błędu tj. czy wystąpił on podczas tworzenia pliku, wyszukiwania itp. Następnie przesyłana jest szczegółowa informacja o przyczynie pojawienia się danego problemu. Błędy przekazywane z IndexServer do API są błędami zwracanymi przez SDDS-2000 Client Service. Typy tych błędów pokrywają się z błędami SDDS Client Service.

Błędy przekazywane z IndexServer do API:

Główne miejsca wystąpienia błędu:

- SDDS_CREATE_MAIN_INDEX_FAILED - błąd wystąpił podczas tworzenia głównego pliku indeksowego
- SDDS_CREATE_OVERFLOW_INDEX_FAILED – błąd wystąpił podczas tworzenia nadmiarowego pliku indeksowego
- SDDS_INSERT_INTO_MAIN_INDEX_ERROR – błąd podczas wstawiania rekordu do głównego pliku
- SDDS_INSERT_INTO_OVERFLOW_INDEX_ERROR – błąd podczas wstawiania rekordu do nadmiarowego pliku
- SDDS_SEARCH_MAIN_INDEX_ERROR – błąd podczas przeszukiwania głównego pliku indeksowego
- SDDS_SEARCH_OVERFLOW_INDEX_ERROR – błąd podczas przeszukiwania nadmiarowego pliku indeksowego

Błędy występujące podczas tworzenia pliku

- SDDS_CREATE_FILE_MAX_LESS_MIN – wartość podana jako maksymalny klucz jest mniejsza od wartości minimalnej
- SDDS_CREATE_FILE_BUCKET_TO_SMALL – podana liczba kubełków jest mniejsza niż minimalna
- SDDS_CREATE_FILE_NAME_TOO_LONG – nazwa podana jako nazwa tworzonego pliku jest zbyt długa
- SDDS_CREATE_FILE_FILE_EXISTS – plik o podanej nazwie już istnieje, należy wybrać inna nazwę dla tworzonego pliku

- SDDS_CREATE_FILE_SUCCESS – tworzenie pliku zakończone sukcesem
- SDDS_CREATE_FILE_NO_AVAILABLE_BUCKETS – w sieci lokalnej nie ma już wolnych serwerów SDDS do utworzenia nowego kubełka
- SDDS_CREATE_FILE_NO_RESPONSE – brak odpowiedzi od serwera SDDS

Błędy występujące podczas wstawiania do pliku

- SDDS_INSERT_INTO_FILE_UNKNOWN_ERROR – niezidentyfikowany błąd
- SDDS_INSERT_INTO_FILE_DATA_SIZE_TOO_BIG – wielkość wstawianych danych jest większa niż określony limit
- SDDS_INSERT_INTO_FILE_NAME_TOO_LONG – podana nazwa pliku jest zbyt długa
- SDDS_INSERT_INTO_FILE_KEY_EXISTS – rekord o podanej wartości kluczowej już istnieje
- SDDS_INSERT_INTO_FILE_SUCCESS – wstawianie zakończone sukcesem
- SDDS_INSERT_INTO_FILE_NO_RESPONSE – brak odpowiedzi od serwera
- SDDS_INSERT_INTO_FILE_DOESNT_EXIST plik o podanej nazwie nie istnieje

Błędy występujące podczas wyszukiwania

- SDDS_SEARCH_UNKNOWN_ERROR – niezidentyfikowany błąd
- SDDS_SEARCH_NAME_TOO_LONG – podana nazwa pliku w w którym na nastąpić wyszukiwanie jest dłuższa od dopuszczalnej
- SDDS_SEARCH_KEY_WAS_NOT_FOUND – brak rekordu o podanym kluczu wyszukiwania
- SDDS_SEARCH_SUCCESS = wyszukiwanie zakończone sukcesem
- SDDS_SEARCH_NO_RESPONSE – brak odpowiedzi od serwera SDDS
- SDDS_SEARCH_FILE_DOESNT_EXIST – brak pliku o podanej nazwie

Tworzenie indeksu (ze strony API):

- wysłanie przez API do IndexServer żądania utworzenia nowego pliku danych SDDS: SDDS_CREATE_INDEX
- wysłanie długości nazwy pliku
  - wysłanie nazwy pliku (w postaci tablicy bajtów)
  - wysłanie informacji o zakończeniu przesyłania nazwy pliku
- wysłanie długości adresu pierwszego serwera SDDS
  - wysłanie nazwy pierwszego serwera (w postaci tablicy bajtów)
  - wysłanie informacji o zakończeniu przesyłania nazwy serwera
- wysłanie wielkości kubełka (jako int)
- wysłanie maksymalnego klucza (jako int)

- odebranie komunikatu od SDDS o stanie tworzenia pliku. W przypadku wystąpienia błędu odebranie szczegółowych informacji o miejscu wystąpienia i przyczynie danego zdarzenia, przerwanie procedury oraz zwrócenie wyjątku. W przypadku sukcesu kontynuowanie procedury tworzenia indeksu.
- pobranie z OOP poprzez interfejs AdWrapperSBQL pierwszego obiektu będącego instancją wskazanej klasy
  - pobranie wszystkich atrybutów danego obiektu, następnie przejrzenie ich w celu odnalezienia wartości wskazanego atrybutu
  - wysłanie do IndexServer długości przesyłanych danych
  - wysłanie danych tworzących referencję do obiektu (w postaci tablicy bajtów)
  - wysłanie typu tworzącego referencję
  - wysłanie długości ciągu będącego wartością atrybutu wg. którego następuje indeksowanie
  - wysłanie ciągu będącego wartością atrybutu (w postaci tablicy bajtów)
  - wysłanie informacji o zakończeniu przesyłania obiektu
  - pobranie informacji o stanie wstawiania referencji do indeksu. Jeśli wszystko przebiegło pomyślnie kontynuowanie procedury. Jeśli nie, pobranie odebranie szczegółowych informacji o miejscu i przyczynie wystąpienia błędu, przerwanie procedury tworzenia indeksu oraz zwrócenie wyjątku.
- powtórzenie procedury do czasu pobrania wszystkich obiektów określonej klasy z SDDS
- wysłanie informacji o zakończeniu procedury indeksowania
- w przypadku odebrania błędu pobranie informacji o miejscu oraz typie błędu, oraz zwrócenie błędy przez funkcję do programu.

Tworzenie indeksu (ze strony IndexServer)

- Odebranie żądania utworzenia nowego indeksu, wywołanie własnej funkcji odpowiedzialnej za tworzenie nowego indeksu
- odebranie długości nazwy nowo tworzonego pliku
- odebranie danych stanowiących nazwę dla nowego pliku
- odebranie długości adresu pierwszego serwera SDDS
- odebranie adresu pierwszego serwera SDDS
- odebranie wielkości kubełka
- odebranie maksymalnej możliwej wartości klucza
- wywołanie w SDDSClient procedury tworzącej nowy plik. Jeśli tworzenie pliku zakończyło sie sukcesem wysłanie odpowiedniej informacji do API. Jeśli nie, wysłanie informacji o błędach oraz przerwanie procedury tworzącej indeks
- odebranie długości referencji, która będzie zachowana w SDDS
- odebranie referencji

- pobranie długości ciągu będącego wartością po której obiekt będzie indeksowany
- pobranie wartości atrybutu
- zachowanie odebranych danych w pamięci
- powtarzanie procedury tak długo, dopóki otrzymuje się informacje że następne referencje będą przesyłane
- przepisanie danych zachowanych w pamięci do pliku w SDDS

Z punktu widzenia programisty stosowanie API jet bardzo proste. Należy wywołać funkcje z odpowiednimi argumentami i oczekiwać na rezultat. Jednak w SDDS-2000 brak jest funkcji aktualizującej rekordy. W przypadku wystąpienia kolizji na danym kluczu istniejące dane musiały by zostać zamazane. Dlatego też niezbędne jest początkowe pobranie wszystkich obiektów do zaindeksowania, obliczenie wartości klucza na podstawie funkcji haszującej i dopiero wtedy wstawienie wszystkich rekordów do SDDS. Wiąże się to z koniecznością przetrzymywania wszystkich obiektów w pamięci, co w przypadku dużych indeksów dotyczącej ogromnej ilości obiektów może okazać się niezbyt wygodne.

Następną kwestią jest wielkość niekluczowego pola danych. Pole to w SDDS-2000 ma wielkość 100 bajtów. W przypadku pojawienia się kilku wartości na jednym kluczu danych może okazać się że wszystkie dane nie mieszczą się w przeznaczonym im miejscu. Dlatego niezbędne jest utworzenie nadmiarowego pliku danych przeznaczonego do trzymania tych rekordów, które nie mieszczą się w pliku głównym. Tworzy się go w tej samej chwili, w której tworzy się główny plik danych. Podczas wstawiania nowych rekordów, jeśli zostanie stwierdzone że w głównym pliku danych brak miejsca na wstawienie nowych referencji, zostaną one umieszczone w nowym pliku.

W głównym pliku organizacja danych w polu niekluczowym jest następująca:

- wartość atrybutu po którym nastąpiło indeksowanie
- referencja do obiektu
- pozycja indeksu dla właściwego klucza wyszukiwania w pliku nadmiarowym
- ilość rekordów odpowiadających dane mu kluczowi wyszukiwania w pliku nadmiarowym

	Pole niekluczowe						
Klucz	Liczba referencji	Wartość atrybutu	Referencja	Wartość atrybutu	Referencja	Indeks w pliku nadmiaro-wym	Ilość rekordów w pliku nadmiaro-wym
3	2	Kowalski	sd223	Nowak	s232	10	3
4	1	Wiśniewski	f223			0	0
5	2	Jasiński	x23	Jasiński	wqq21	13	1

Ilustracja 39. organizacja rekordu danych w pliku głównym

Wygląd niekluczowego pliku danych jest podobny. Zawiera on:

- wartość haszowanego atrybutu
- referencję do obiektu

	Pole niekluczowe	
Klucz	Wartość atrybutu	Referencja
10	Kowalski	ff6564
11	Kowalski	gh2378
12	Nowak	fd34
13	Jasiński	dwe3423

Ilustracja 40. organizacja rekordów danych w pliku nadmiarowym

Wyszukiwanie (ze strony API)

- wysłanie przez API do IndexServer żądania wyszukiwania danych w SDDS: SDDS_SEARCH_INDEX
- wysłanie długości nazwy pliku do przeszukiwania
  - wysłanie nazwy pliku (w postaci tablicy bajtów)
  - wysłanie informacji o końcu przesyłania nazwy pliku
- wysłanie długości nazwy klucza wg którego nastąpi wyszukiwanie
  - wysłanie nazwy klucza (w postaci tablicy bajtów)
  - wysłanie informacji o końcu przesyłania wartości klucza
- odebranie informacji o stanie wyszukiwania danych w SDDS. W przypadku braku błędów kontynuowanie operacji. W razie stwierdzenia błędów, odebranie szczegółów dotyczących miejsca wystąpienia problemów, oraz zwrócenie wyjątku.
- odebranie informacji o ilości znalezionych obiektów
- odebranie pierwszego znalezionego obiektu w postaci:
  - odebranie ciągu tworzącego referencję

- odebranie wartości atrybutu wg którego obiekt był indeksowany
- odebranie typu referencji będącego niezbędnym elementem pozwalającym na odtworzenie unikalnego identyfikatora obiektu w OOP
- sprawdzenie czy odebrany obiekt spełnia warunek wyszukiwania (możliwość kolizji na wartości kluczowej w SDDS z uwagi na stosowanie funkcji haszującej). Jeśli tak, odtworzenie unikalnego identyfikatora SknUniqueId będącego referencją do obiektów z OOP. Następnie poprzez interfejs AdWrapper pobranie wyszukiwanego obiektu z OOP.
- Powtórzenie operacji pobierania dla wszystkich referencji odebranych z SDDS
- Zwrócenie tablicy znalezionych obiektów przez funkcje wyszukującą
- odebranie odpowiedniej ilości obiektów (wartości klucza, wartości tworzącej referencję oraz typu)
- sprawdzenie wartości klucza i ewentualne odrzucenie niewłaściwych obiektów (niezbędne z uwagi na możliwość kolizji wartości dla różnych danych wejściowych dla funkcji haszującej).

W przypadku wystąpienia błędu komunikacji z SDDS lub błędu w OOP funkcja wyszukująca zwróci wyjątek SDDSException z opisem błędu.

Wyszukiwanie (ze strony IndexServer)

- odebranie żądania uruchomienia procedury wyszukiwania SDDS
- odebranie długości nazwy pliku w którym nastąpi wyszukiwanie
    - odebranie nazwy pliku
- odebranie długości wartości atrybutu po którym nastąpi wyszukiwanie
    - odebranie wartości atrybutu
- wysłanie informacji o sukcesie wyszukiwania, bądź pojawieniu sie błędu
- wysłanie liczby znalezionych referencji
    - wysłanie długości referencji
    - wysłanie referencji
    - wysłanie długości wartości atrybutu po którym obiekt był indeksowany
    - wysłanie wartości atrybutu
    - wysłanie typu referencji
    - powtarzanie procedury dla wszystkich znalezionych rekordów

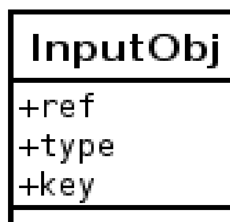
W indeksie (pliku głównym oraz nadmiarowym) wyszukiwane są odpowiednie rekordy a następnie zwracana kolekcja znalezionych danych. Wraz z danymi tworzącymi referencję przesyłany jest wartość atrybutu po którym obiekt został indeksowany. Po odebraniu informacji sprawdzane jest czy wszystkie zwrócone dane spełniają warunki wyszukiwania. Jeśli wskutek kolizji zwracanych wartości przez funkcję haszującą znalazły się ty obiekty niepożądane,

zostaną one odrzucone.

Ze względu na ilość dostępnego miejsca na składowanie danych w SDDS-2000 przechowywane są nie referencje do obiektów, a ciągi tekstowe tworzące (będące argumentem dla konstruktora) unikalny identyfikator obiektów. Dlatego też taki identyfikator musi zostać odtworzony. Wymaga to użycia konstruktora SknUniqueId implementującego interfejs AdUniqueID.

Aby uprościć operacje na odebranych danych, informacje dotyczące jednego obiektu zbierane są w jednej klasie. Są to:

- ciąg tekstowy reprezentujący referencję do obiektu, służący do odtworzenia identyfikatora
- typ referencji
- wartość tekstowa po której obiekt był indeksowany



Ilustracja 41. Klasa zawierające informacje o obiekcie znalezionym w SDDS

Aktualizacja, Usuwanie

Na aktualnej wersji systemu SDDS-2000 realizacja tej funkcjonalności okazała się niemożliwa.

Funkcje API dla SDDS-2000

Indeks oparty na SDDS-2000 w interfejsie programistycznym dostarcza następujące metody:

- *SDDSConection (String serverAddress, AdWrapperSBQL sbql)* – konstruktor głównej klasy API łączącej SDDS oraz OOP. Jako argumenty przyjmuje adres serwera SDDS oraz odnośnik do obiektu implementującego AdWrapper oraz AdWrapperSBQL.
- *createIndex(String fileName, int bucketSize, String className, String attrName)* – funkcja tworzy plik indeksowy o zadanej nazwie fileName. Następnie plik wypełniany jest referencjami do obiektów klasy className, oraz indeksowany wg. wartości atrybutu attrName. Dodatkowo, zapisywane są także wszystkie utworzone indeksy, co pozwoli uzyskać listę zapamiętanych plików.
- *AdObject[] getObjects(String fileName, String attribute)* – zwraca tablicę obiektów OOP (implementujących interfejs AdObject). Jako argumenty funkcji należy podać nazwę pliku, oraz wartość atrybutu.

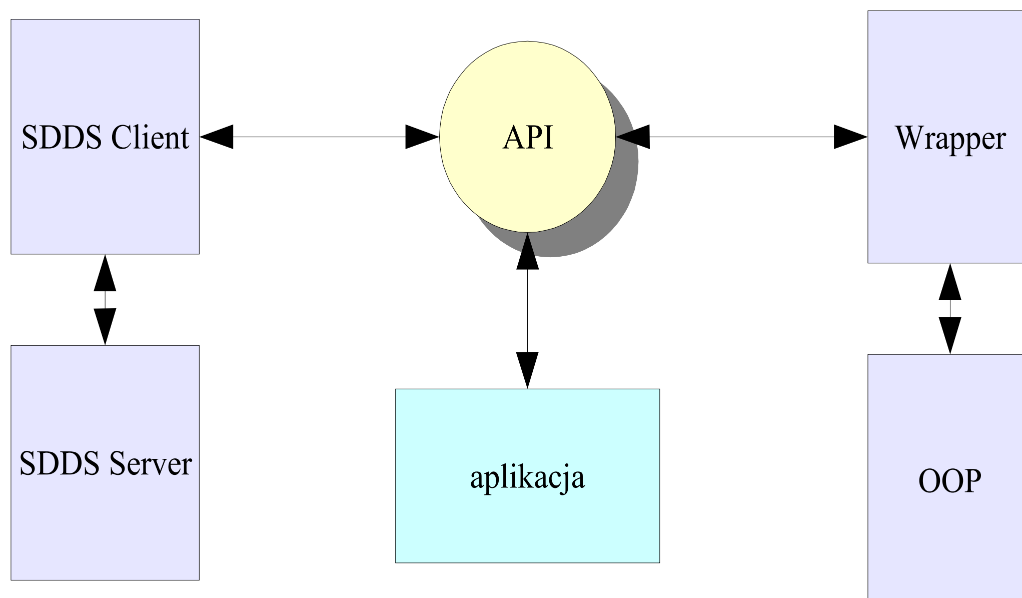
7.5.2. Indeks zbudowany na systemie SDDS-LH*

Architektura API dla nowej wersji SDDS jest prosta i przejrzysta. Do komunikacji z OOP wykorzystuje się implementację wrappera com.mt.abstractdatabase. W ten sposób uzyskuje się metadane o nazwach klas istniejących w OOP, oraz typach i nazwach atrybutów. Wrapper ten wykorzystywany jest także do pobierania obiektów oraz referencji.

Do komunikacji z SDDS stosuje się pakiet SDDSClient, będący częścią systemu SDDS. Pozwala on na manipulację danymi w systemie (wstawianie, usuwanie, aktualizacja), oraz tworzenie nowych indeksów. Osiąga się to poprzez wywoływanie odpowiednich funkcji, jak np. getObjects czy createFile.

Budowa nowego SDDS pozwala na pominięcie w samym API szczegółów dotyczących połączeń sieciowych, typie przesyłanych danych oraz błędach komunikacji. Wykorzystuje się jedynie funkcje dostępne w pakiecie klienckim. Transmisja jest w całości zrealizowana w SDDSClient.

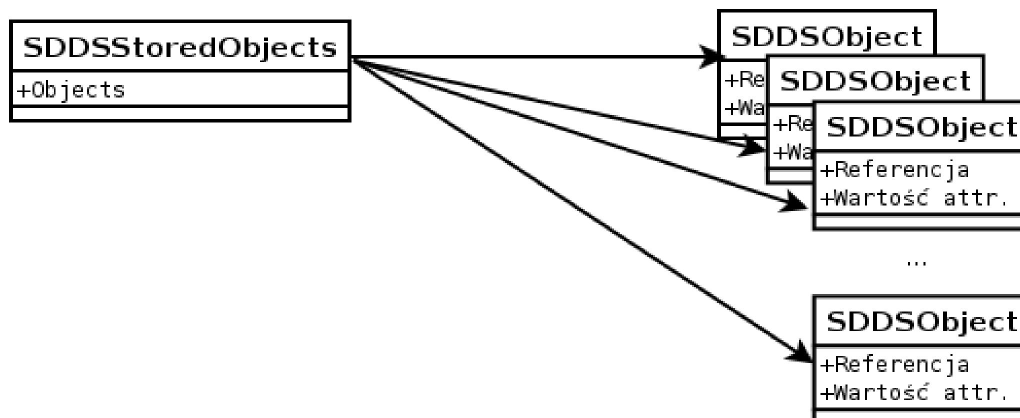
Wszystkie komponenty komunikujące się ze sobą napisane są w języku Java. Nie ma więc potrzeby stosowania dodatkowych modułów przejściowych pomiędzy API a SDDS. Nie występują też problemy związane z niekompatybilnością typów przesyłanych danych.



Ilustracja 42. Schemat współpracy aplikacji z SDDS oraz OOP

Rekord danych w SDDS

Rekord danych SDDS składa się z klucza oraz wartości niekluczowej. Wartością niekluczową może być dowolny ciąg bajtów, w szczególności obiekt Javy. Do przechowywania referencji po stronie SDDS wykorzystywana jest struktura zawierająca jedną lub wiele referencji do obiektów OOP. Dodatkowo, wraz z każdą referencją zapamiętywana jest także wartość atrybutu po którym indeksowano dany obiekt. Zarówno SDDSStoredObjects, jak i SDDSObject są serializowane, co pozwala na przesyłanie ich prze sieć oraz późniejsze odtwarzanie.



Ilustracja 43. Struktura danych przechowująca referencje do obiektów

Jako referencje do obiektów stosowane są struktury danych implementujące interfejs AdUniqueId wchodzący w skład com.mt.abstractdatabase. Identyfikuje on w sposób jednoznaczny wszystkie obiekty występujące w systemie zgodnym z tym wrapperem (a wiec także w OOP). Istnieje możliwość, aby jako referencje do obiektów przechowywać nie całe obiekty, ale wartości tekstowe tworzące identyfikatory do obiektów. Dzięki temu ilość danych przesyłanych pomiędzy końcową aplikacją a SDDS była by mniejsza. Wymagało by to jednak użycia konstruktora do konkretnej klasy implementującej interfejs AdUniqueId (w przypadku systemu OOP jest to SknUniqueID) do odtworzenia obiektu. Opieranie się w całości na AdWrapper pozwala zachować większą uniwersalność. Stosowanie API możliwe będzie z każdym systemem mającym swoją implementację wrappera bez żadnych zmian z kodzie bibliotek.

Błędy

API zwraca wyjątek SDDSException. Obiekt wyjątku zawiera informację o miejscu wystąpienia błędu. Zwracane błędy mogą być odzwierciedleniem wyjątków rzucanych przez AdWrapper lub SDDSClientService. Jednak nie jest to reguła. W niektórych przypadkach wyjątki SDDS są wykorzystywane do podjęcia określonej akcji, której wynikiem może być określony rezultat np. zwrócenie przez funkcję API wartości null lub wstawienie nowego

rekordu zamiast aktualizacji.

Tworzenie indeksu

- Zostanie wywołana funkcja tworząca nowy plik danych o zadanej nazwie. Jeśli taki plik juz istnieje SDDS zwróci wyjątek SDDSFileExistsException.
- Najpierw należy określić jakiej klasy obiekty, oraz wg którego atrybutu będą indeksowane
- Następnie za pomocą obiektu reprezentującego połączenie z OfficeObjects Portal, oraz implementującego interfejs AdWrapperSBQL pobierany jest pierwszy obiekt będący instancją wyszukiwanej klasy
- Za pomocą funkcji haszującej oblicza się wartość zadanego atrybutu w wybranym obiekcie; otrzymana wartość jest kluczem wyszukiwania pod którym zapisywane są rekordy w SDDS
- Sprawdza się, czy pod obliczonym kluczem wyszukiwania znajdują się już jakieś referencje. Jeśli tak w wyniku sprawdzenia zostanie zwrócony obiekt zawierający istniejące dane. Jeśli nie SDDSClient zwróci wyjątek SDDSNoSuchKeyException.
- Jeśli nie zostały znalezione żadne referencje tworzony jest nowy obiekt który zostanie zapisany w SDDS pod obliczonym wcześniej kluczem. Jeśli w SDDS istnieją już referencje, nowy identyfikator zostanie dodany do obiektu zawierającego wszystkie identyfikatory spod danego klucza. Następnym krokiem będzie aktualizacja rekordu danych w SDDS
- zostanie pobrany następny obiekt z OOP. Operacja będzie powtarzana tak długo, aż zostaną pobrane i zaindeksowane wszystkie obiekty z OOP spełniające zadane warunki.

Wyjątki SDDSFileExistsException oraz SDDSNoSuchKeyException są obsługiwane w kodzie i ich wystąpienie wywołuje wykonanie określonego kodu w ciele funkcji. Dodatkowo mogą pojawić się błędy podczas transmisji danych w sieci lub podczas pobierania danych z OOP. Zdarzenia te spowodują zwrócenie wyjątku przez funkcję tworzącą indeks. Informacja o błędzie będzie zgodna z informacją zawartą w wyjątku występującym podczas wykonania funkcji.

Wyszukiwanie

- Wywoływana jest funkcja pobierająca dane z SDDS na podstawie podanego pliku oraz wartości kluczowej. Wartość kluczowa obliczana jest przy użyciu funkcji haszującej.
- Jeśli pod wskazanym kluczem wyszukiwania znajduje sie obiekt z referencjami zostanie on zwrócony. Jeśli nie, funkcja getObject z SDDSClient zwróci wyjątek SDDSNoSuchKeyException.
- W przypadku istnienia danych we wskazanym miejscu zostanie sprawdzone czy wszystkie referencje znajdujące się w obiekcie spełniają warunek wyszukiwania (istnieje możliwość iż na skutek kolizji na funkcji haszującej niektóre referencje mogą być niezgodne w wyszukiwana wartością atrybutu).
- Po przefiltrowaniu znalezionych referencji z OOP zostaną pobrane obiekty którym

odpowiadają unikalne identyfikatory pobrane z SDDS

Podobnie jak podczas tworzenia indeksu; wyjątek SDDSNoSuchKeyException jest obsługiwany w kodzie i wykorzystywany jest do uruchomienia odpowiednich procedur. Błędy modułów SDDS oraz OOP, a także błędy transmisji powodują zwrot wyjątku SddsApiEception.

Aktualizacja całego indeksu

- Należy określić, który indeks będzie aktualizowany.
- następnie (podobnie jak w przypadku tworzenia) należy podać nazwę klasy oraz nazwę atrybutu do indeksowania
- każdy obiekt pobrany z OOP zostanie sprawdzony, czy jest już zaindeskowany w określonym pliku czy nie; sprawdzenie polega na pobraniu całej struktury danych przechowującej referencje po stronie SDDS, a następnie porównaniu obiektów których identyfikatory istnieją w systemie z nowym obiektem

Usuwanie rekordu

- Należy podać nazwę pliku indeksowego oraz wartości atrybutów jakie zostaną usunięte z indeksu w SDDS. W przypadku braku podanego pliku SDDSClient zwróci błąd SDDSNoSuchFileException.
- Jeśli plik istnieje, zostanie wywołana funkcja wyszukująca dane w określonym pliku. Jeśli w indeksie brak rekordu o zadanym kluczu wyszukiwania zostanie zwrócony wyjątek SDDSNoSuchKeyException. Jeśli w indeksie istnieje odpowiedni wpis zostanie zwrócona struktura danych zawierająca referencje do obiektów (tak jak w przypadku wyszukiwania).
- W przypadku znalezienia rekordu danych niezbędne będzie sprawdzenie czy wszystkie referencje dotyczą obiektów o określonej wartości atrybutu. Jeśli tak, w SDDSClient zostanie wywołana funkcja usuwająca cały rekord danych w SDDS. Jeśli nie, z obiektu zawierającego referencje do danych w OOP zostaną usunięte identyfikatory a następnie wskazany rekord zostania zaktualizowany nową wersją danych.

Podobnie jak w poprzednich przypadkach, błąd podczas transmisji pomiędzy SDDS lub OOP powoduje zwrócenie wyjątku SDDSException.

Funkcje API dla nowego SDDS

Funkcje oferowanie programiście przez API to:

SDDSConnection(String serverAddress, AdWrapperSBQL sbql)

konstruktor głównej klasy API łączącej SDDS oraz OOP. Jako argumenty przyjmuje adres serwera SDDS oraz odnośnik do obiektu implementującego AdWrapper oraz AdWrapperSBQL

createIndex(String fileName, int bucketSize, String className, String attrName)

funkcja tworzy plik indeksowy o zadanej nazwie fileName. Następnie plik wypełniany jest

referencjami do obiektów klasy className, oraz indeksowany wg. wartości atrybutu attrName. Dodatkowo, zapisywane są także wszystkie utworzone indeksy, co pozwoli uzyskać listę zapamiętanych plików.

AdObject[] getObjects(String fileName, String attribute)

zwraca tablicę obiektów OOP (implementujących interfejs AdObject). Jako argumenty funkcji należy podać nazwę pliku, oraz wartość atrybutu. Wartość ta zostanie przetworzona przez funkcję haszującą, a kolekcja otrzymanych obiektów będzie spełniać zadany warunek. (Ponieważ pod takim samym kluczem mogą zostać zaindeksowane obiekty o różnych wartościach atrybutu, w chwili pobrania referencji z SDDS zostanie sprawdzone, czy są one właściwe, a dopiero później odpowiednie dane zostaną ściągnięte z OOP).

String[] getIndices()

Zwraca tablicę stringów zawierającą listę indeksów utworzonych w SDDS

deleteObjects(String fileName, String attrValue)

usuwa rekordy z pliku fileName zaindeksowane pod kluczem będącym zhaszowaną wartością attrValue

updateIndex(String fileName, String className, String attrName)

aktualizuje indeks fileName obiektami klasy className, oraz za pomocą atrybutu attrName. Identyfikatory obiektów pobieranych z OOP są porównywane z identyfikatorami istniejącymi juz z SDDS. Jeśli określone dane występują już w indeksie, nie są one dodawane do pliku.

7.6. Funkcja mieszająca

Funkcja mieszająca stosowana jest do obliczania wartości kluczowej pod którą zostaną zaindeksowane referencje do obiektów. Funkcja ta nie powinna zawierać prostych zależności pomiędzy argumentem a zawracaną wartością, oraz powinna generować różne wartości dla różnych danych wejściowych. Jeśli będzie dochodziło zbyt często do kolizji (powtarzania się rezultatu funkcji dla różnych danych) może to doprowadzić do znacznego spadku wydajności przy wyszukiwaniu, jak również przy wstawianiu.

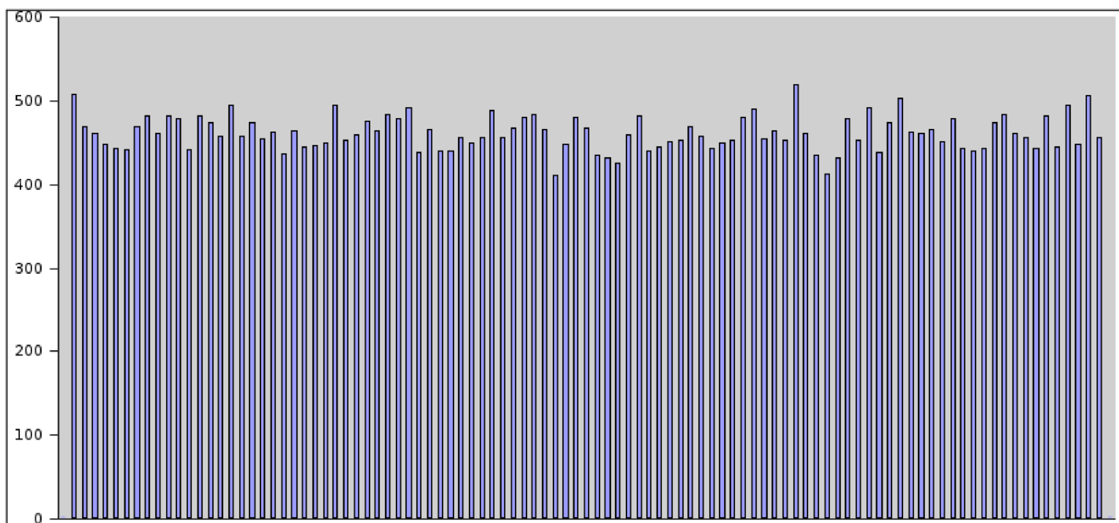
Aby wybrać odpowiednią funkcję mieszającą przetestowano kilka znanych algorytmów przekształcających tekstowy łańcuch wejściowy w wartość liczbową. Testów dokonano na 46000 słów wchodzących w skład brytyjskiego słownika przeznaczonego dla edytora tekstów OpenOffice Writer.

Wszystkie testowane algorytmy cechowały się odpowiednią szybkością działania, a także dobrym rozkładem wartości wyjściowych. Ostatecznie zdecydowano się na funkcję autorstwa Alana Partowa.

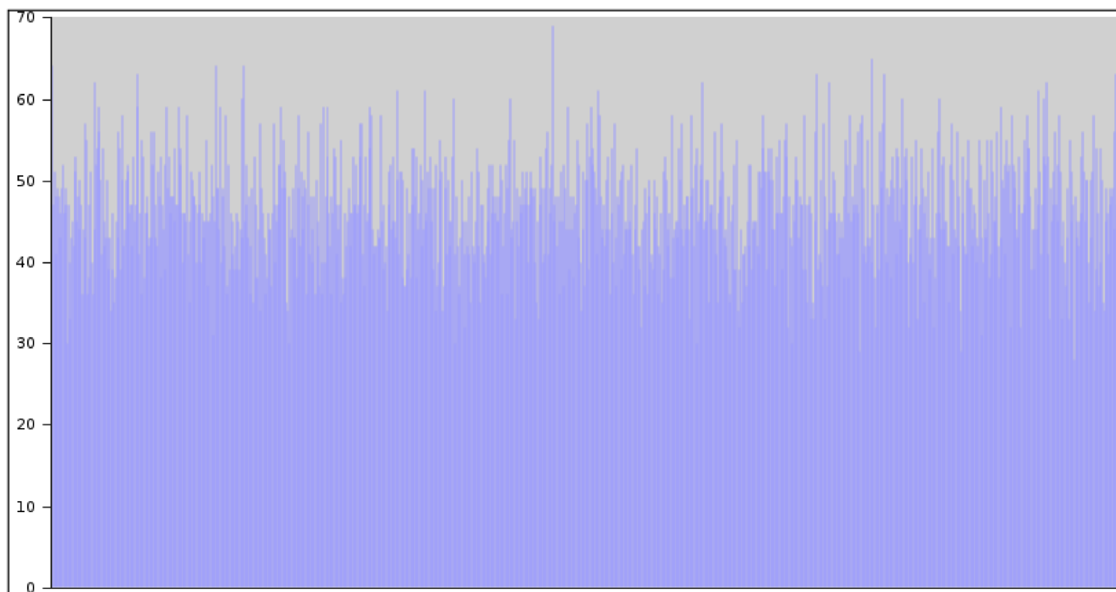
```
public long APHash(String str)
 {
   int hash = 0;
   for(int i = 0; i < str.length(); i++)
   {
     if ((i & 1) == 0)
     {
       hash ^=((hash << 7)^str.charAt(i)^(hash >> 3));
     }
     else
     {
       hash ^= (~((hash << 11)^str.charAt(i)^(hash >> 5)));
     }
   }
   return (hash & 0x7FFFFFFF);
 }
```

Ilustracja 44. Kod zastosowanej funkcji haszującej

Zamieszczone wykresy pokazują rozkład wartości dla testowych słów z angielskiego słownika do OpenOffice. W pierwszym przypadku rezultaty ograniczono do 100 wartości, z drugim do 1000



Ilustracja 45. Rozkład dla funkcji haszującej modulo 100



Ilustracja 46. Rozkład dla funkcji haszującej modulo 1000

7.7. Testowanie

Do testowania poprawności działania oraz funkcjonalności API stworzono prostą aplikację testową. Aplikacja ta pozawala na zapoznanie się z oferowanymi funkcjami oraz sprawdzenia ich działania w praktyce. Program pozawala na utworzenie nowego indeksu, sprawdzenie listy indeksów, przeglądanie zaindeksowanych danych itp. Tak jak i API, program jest napisany w języku Java.

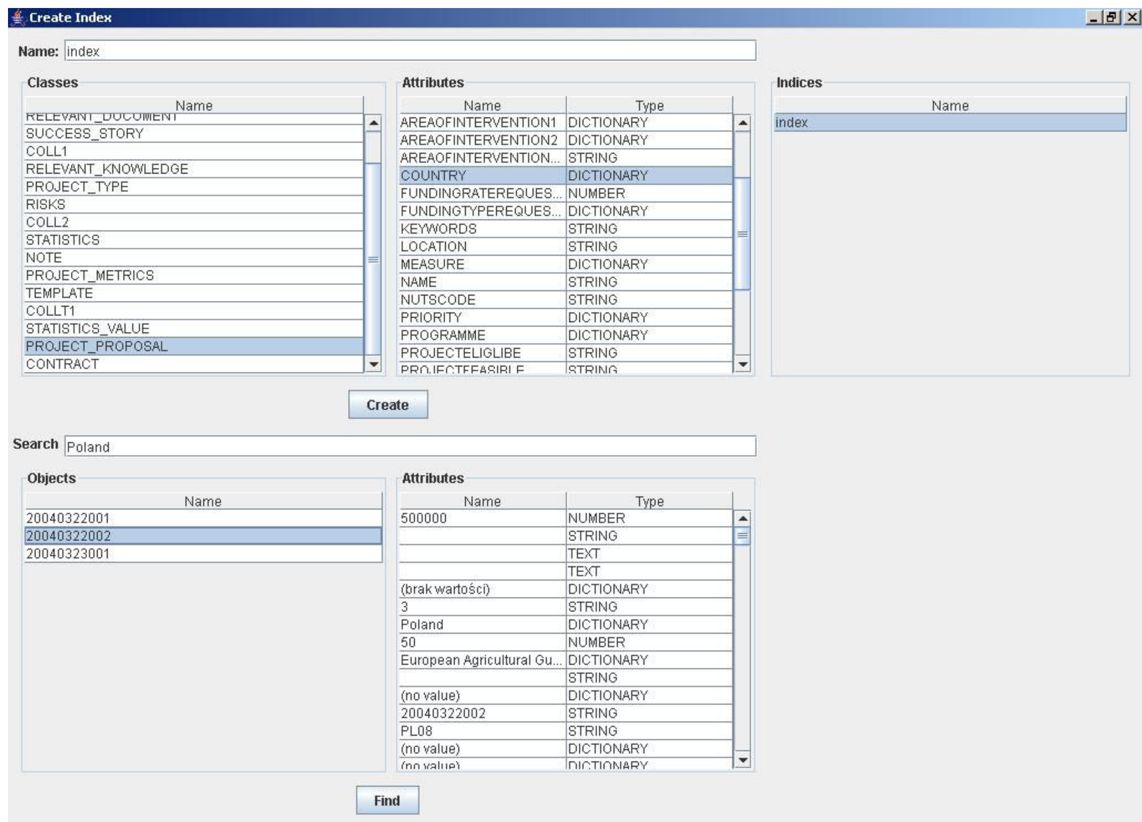


Illustration 1Ekran aplikacji tetsowej

Możliwości programu:

<u>Tworzenie indeksu</u>

Name - miejsce na wpisanie nazwy jaka będzie nadana nowemu indeksowi

Classes - podczas uruchomienia programu tabela ta wypełnia się nazwami klas występujących w systemie OOP; nazwa ta zostanie podana jako argument metody tworzącej i wypełniającej nowy indeks.

Attributes - po wybraniu nazwy klasy z tabelki po lewej stronie, tabela attributes zostanie wypełniona atrybutami danej klasy; także ta wartość zostanie przekazana do funkcji budującej indeks obiektów z OOP.

Przycisk Create - wciśnięcie przycisku spowoduje utworzenie nowego indeksu. Potrzebne dane zostaną pobrane z tabel Classess oraz Attributes.

Indices - tabela ta pokazuje listę indeksów istniejących w SDDS. Jest ona aktualizowana po każdym utworzeniu nowego indeksu. Do wyszukiwania należy wskazać jedną z dostępnych pozycji.

Search - należy tu wpisać wartość atrybutu (wg. którego dane zostały zaindeksowane) która będzie poszukiwana w indeksie

Objects - tabela ta zostanie wypełniona nazwami wszystkich znalezionych obiektów

Attributes – po wskazaniu jednego z listy znalezionych obiektów tabela attributes zostanie wypełniona wartościami atrybutów właściwych dla danego obiektu.

Przycisk Find- wciśnięcie przycisku spowoduje uruchomienie procedury wyszukiwania

danych, oraz jeśli odpowiednie rekordy zostaną znalezione, wypełni się tabela Objects. Jako argumenty funkcji wyszukiwania zostaną pobrane wartości z tabeli Indices, oraz z pola tekstowego Search.

Przycisk Delete - wciśnięcie przycisku spowoduje usunięcie z indeksu zaznaczonego w tabeli Indices wartości zaindeksowanej po atrybucie SearchText.

7.8. Testy wydajnościowe

Po stworzeniu interfejsów programistycznych łączących SDDS oraz OfficeObjects Portal przeprowadzono proste testy wydajnościowe pozwalające zorientować się czy stosowanie zaproponowanego rozwiązania przyniesie wymierne korzyści.

7.8.1. Metoda testowania

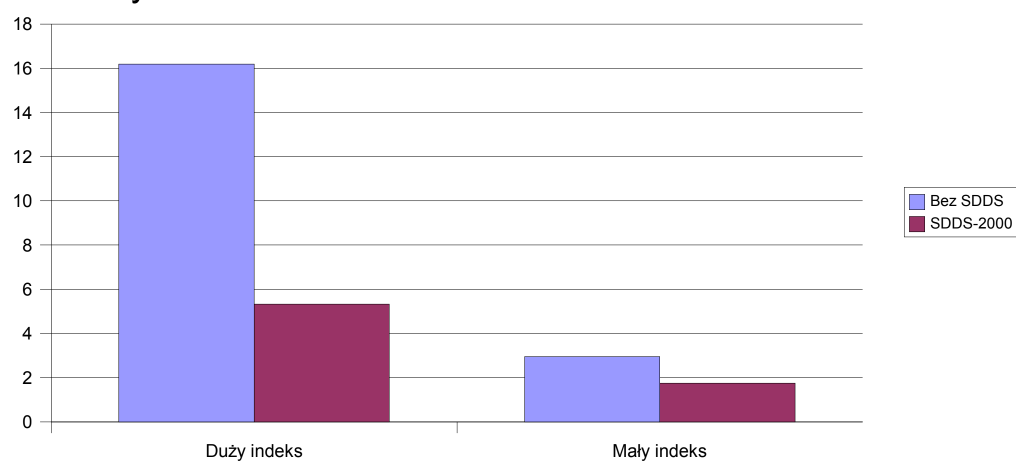
Do testowania napisano prosty program. Jego zadaniem było łączenie się ze źródłem danych , a następnie pobieranie wskazanych obiektów. Obiekty z OfficeObjects Portal pobierano bezpośrednio, przy pomocy warppera com.mt.abstractdatabase, lub za pośrednictwem SDDS. Dostęp do zaindeksowanych obiektów zrealizowano przy pomocy stworzonych interfejsów API.

Jako że nie dysponowano dużym laboratorium komputerowym testy zostały wykonane na dwóch komputerach. Mogło to stanowić ograniczenie w przypadku SDDS-2000, gdyż ze względu na specyficzną implementację tego systemu zużywa on o wiele więcej zasobów niż mogło by to wynikać z jego funkcjonalności.

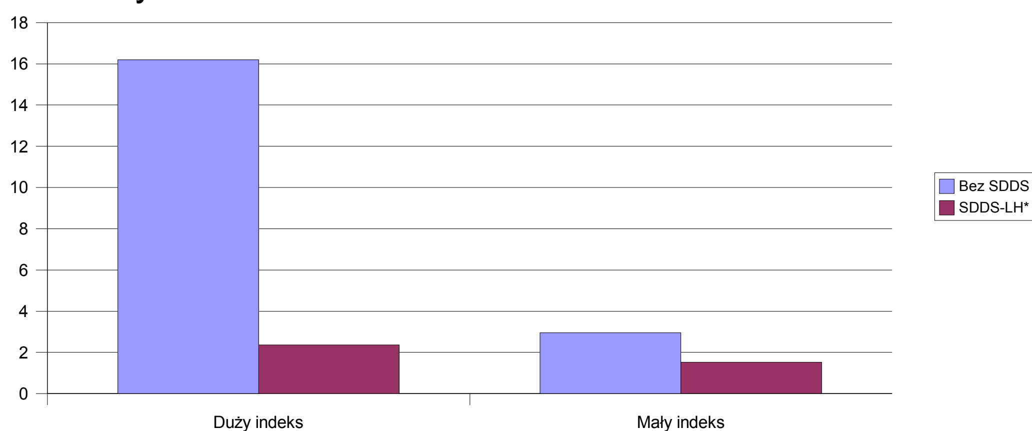
Jako źródło danych testowych zastosowano dostarczoną wraz z OfficeObjects Portal przykładową bazę danych. Nie jest ona jednak bardzo duża (co nie zmienia faktu, że wyciągnięcie z niej kilkuset obiektów zajmuje kilka minut czasu), w związku z tym zbiór danych musiał być ograniczony i mniejszy niż zazwyczaj spotykane w rzeczywistości

7.8.2. Wyniki

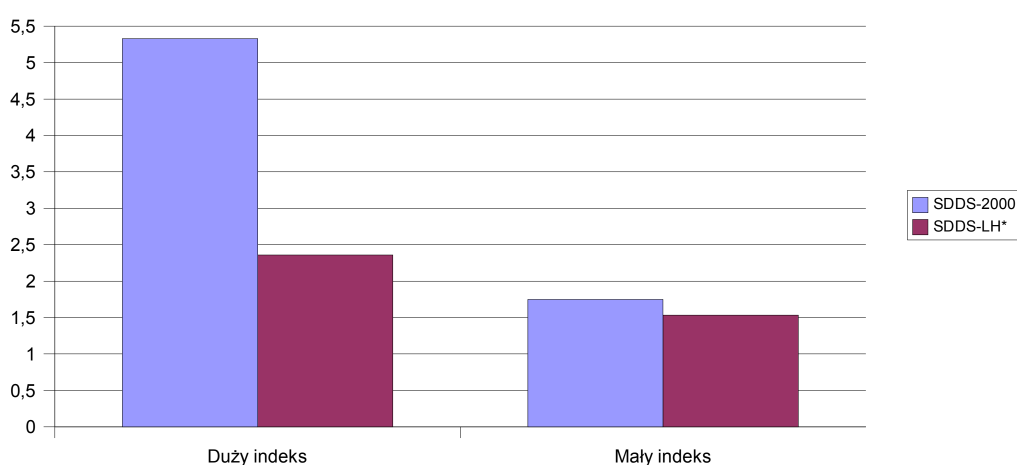
Dane zebrane podczas testowania zostały zapisane i przedstawione w formie wykresów. Porównać można czasy wyszukiwania w samym OfficeObjects Portal, jak i z pomocą indeksów stworzonych w SDDS.



Ilustracja 47. Wyniki wyszukiwania w OOP z zastosowaniem indeksu opartego na SDDS-2000



*Ilustracja 48. Wyniki wyszukiwania w OOP z zastosowaniem indeksu opartego na SDDS-LH**



*Ilustracja 49. Porównanie wyników wyszukiwania dla SDDS-2000 i SDDS-LH**

7.8.3. Podsumowanie

Jak widać na powyższych wykresach zastosowanie systemu SDDS jako repozytorium indeksów dla systemu OOP przynosi wymierne korzyści. Nawet na ograniczonym zbiorze danych testowych można zaobserwować znaczny przyrost prędkości wyszukiwania. Wraz ze wzrostem ilości danych zysk okazuje się być coraz większy. O ile dla małego zbioru obiektów zastosowanie SDDS pozwala osiągnąć przyspieszenie blisko dwukrotne, to dla niewiele większego jest to odpowiednio przyspieszenie trzykrotne dla SDDS-2000 i prawie siedmiokrotne dla SDDS-LH*.

Można przypuszczać, że w prawdziwych zastosowaniach, gdzie ilość danych w sposób znaczny przekroczy zasób dostępny autorom pracy, osiągnięte korzyści będą znacznie większe.

Należy zaznaczyć, iż testy obejmowały jedynie współdziałanie systemów SDDS wraz z OOP. Przeprowadzone były na dostępnych wersjach programów, niekoniecznie stabilnych i ostatecznych, na ograniczonych zasobach sprzętowych.

Nie obejmowały też bezpośrednich porównań obu systemów, SDDS-2000 oraz SDDS-LH*, jak i samej szybkości działania OfficeObjects Portal.

Rozdział 8. Napotkane trudności

8.1. Problemy

Podczas prac natknięto się na szereg problemów i przeszkód. Niektóre z nich doprowadziły do opóźnień i przesunięć ustalonych terminów. Inne z kolei zmusiły do zmiany przyjętych założeń lub nawet rezygnacji z projektowanych rozwiązań. Przyczyny przeszkód były bardzo różne (techniczne jak i ludzkie), co zostanie dokładniej wyjaśnione w punktach dotyczących konkretnego elementu systemu.

8.1.1. SDDS-2000

SDDS-2000 przysporzył autorom szeregu problemów dokładnie opisanych w sekcji opisującej system SDDS-LH*.

8.1.2. OfficeObjects Portal

System OOP nie jest systemem ogólnie dostępnym. Stworzony został przez firmę Rodan do zastosowania jako silnik we własnych projektach. W związku z tym brak do niego szczegółowej dokumentacji zawierającej wskazówki instalacyjne (przydatne z uwagi na skomplikowaną i rozbudowaną architekturę). Brak ten był bardzo odczuwalny, szczególnie w pierwszej fazie zapoznawania się z systemem OOP, tym bardziej, że system nie został stworzony z myślą o samodzielnej instalacji.
Istnienie kilku wersji poszczególnych pakietów wchodzących w skład systemu, jak i różnych wersji danych testowych wprowadzało pewną konfuzję podczas instalacji systemu.

Nie możliwe było przetestowanie stworzonego pakietu na wystarczająco dużym zbiorze danych. Składało się na to kilka czynników:

- w pierwszej fazie problemy z systemem SDDS-2000
- niedostępność wystarczająco dużego zbioru danych testowych
- niedostępność wystarczająco dużego laboratorium pozwalającego przetestować mechanizm podziały pliku danych na inne maszyny z funkcjonującym systemem SDDS.

8.2. Plany

Ewentualne dalsze kroki możliwe do podjęcia w związku z tą pracą magisterską i produktami

powstałymi w jej ramach to:

- W przypadku powstania w pełni funkcjonującej wersji systemu SDDS-2000 – dodanie nowych elementów do API .
- System SDDS-LH* powstał w stosunkowo krótkim czasie. Niemożliwe było więc jego dokładne przetestowanie pod możliwie dużym obciążeniem. Dlatego też pełne testy mogły by wskazać ew. elementy wymagające dopracowania.
- W obecnej formie SDDS-LH* nie jest odporny na awarie serwerów, zastosowanie opisanych modyfikacji algorytmu LH* zwiększyło by praktyczną użyteczność systemu.

Rozdział 9. Zakończenie

W ramach pracy zrealizowany wszystkie założone cele. Powstało funkcjonalne API umożliwiające wyszukiwanie obiektów systemu OfficeObjects Portal przy wykorzystaniu indeksu. Główną ideą przyświecającą w trakcie tworzenia pracy było praktyczne zastosowanie systemu SDDS-2000. Wyjściowy system SDDS okazał się jednak zbyt niedojrzały, aby efektywnie przechowywać na nim pliki indeksu. Najważniejszym efektem pracy jest własna implementacja struktury SDDS oparta o algorytm LH*, która istotnie przewyższa funkcjonalnością system SDDS-2000 rozwijany już ponad 4 lata przez zespół prof. Litwina, oferując przy tym większą wydajność.

Powstały indeks realizuje wszystkie zaplanowane funkcje, mimo to nadal pozostaje pytanie o praktyczne zastosowania dla struktur SDDS. W tym świetle rzuca się w oczy brak propozycji jakichkolwiek mechanizmów transakcyjnych, a bez nich potencjalne zastosowania są mocno ograniczone.

Być może SDDS mógłby się sprawdzić w niszowych zastosowaniach, gdzie wykorzystuje się bardzo duże indeksy przeznaczone głównie do odczytu.

Bibliografia

[J1] Sun, The Java Language - An Overview, , http://java.sun.com/docs/overviews/java/java-overview-1.html

[J2] Javasoft.pl, Charakterystyka Jezyka Java, , http://javasoft.pl/java/wprowadzenie.html

[L1] Witold Litwin, Scalable Distributed Data Structures, State of the art,

[L80] Witold Litwin, Linear Hashing: A new tool for file and table addressing, 1980

[LN96] Witold Litwin, Marie-Anne Neimat, High-Availability LH* Schemes with Mirroring, 1996

[LNGNS1] Witold Litwin, Marie-Anne Neimat, G. Levy, S. Ndiaye, T. Seck, LH*s: a High-availability and High-security Scalable Distributed Data Structure,

[LNS94] Witold Litwin, Marie-Anne Neimat, Donovan Schneider, RP*: A Family of Order Preserving Scalable Distributed Data Structures, 1994

[LNS96] Witold Litwin, Marie-Anne Neimat, Donovan A. Schneider, LH* - A Scalable, Distributed Data Structure,

[LR1] Witold Litwin, Tore Risch, High-availability Scalable Distributed Data Structure By Record Grouping,

[M01] Mike Morgan, Poznaj Język Java, 2001

[OOP03] Rodan Systems S.A., Opis Porduktu OfficeObjects Portal 1.2, 2003

[S04] Kazimierz Subieta, Teoria i konstrukcja obiektowych języków zapytań, 2004

[S99] Robert Sedgewick, Algorytmy w C++, 1999