WYDZIAŁ AUTOMATYKI, ELEKTRONIKI I INFORMATYKI
POLITECHNIKI ŚLĄSKIEJ

INSTYTUT INFORMATYKI

ROZPRAWA DOKTORSKA

Model przetwarzania strumieniowego uwzględniający zarówno synchronizację jak i język zapytań łączący paradygmaty języka obiektowego i deklaratywnego

mgr inż. Aleksander Chrószcz

Promotor: dr hab. inż. **Marcin Gorawski**
Prof. nzw. Politechniki Wrocławskiej

Gliwice, 2012

Pracę dedykuje rodzicom i moim babciom

Życie nie zostało stworzone z myślą, aby było proste. Na szczęście istnieją ludzie, dzięki którym odnajdujemy swoje zainteresowania oraz inspiracje. Podziękowania składam promotorowi dr hab. inż. Marcinowi Gorawskiemu prof. nzw. Politechniki Wrocławskiej.

SPIS TREŚCI

Rozdział 1. Wstęp 8
1.1 Tezy rozprawy 11
1.2 Organizacja pracy 11
Rozdział 2. Algebra przetwarzania strumieniowego 13
2.1 Podstawy 17
2.1.1 Operatory jednowejściowe (unarne) 19
2.1.2 Operatory dwuwejściowe (binarne) 19
2.2 Kryteria oceny poprawności operatorów strumieniowych 20
2.3 Klasyfikacja strumieni 22
2.4 Strumienie 26
2.5 Monotoniczność strumieni 29
2.6 Operatory 32
2.7 Operatory okien 33
2.7.1 Okno przesuwno-czasowe 35
2.7.2 Okno stało-czasowe 36
2.7.3 Okno liczebnościowe 36
2.8 Operatory pochodzące z relacyjnych baz danych 37
2.8.1 Operator selekcji 37
2.8.2 Operator mapujący 38
2.8.3 Operator Θ złączenie 39
2.8.4 Operator agregacji 44
2.8.5 Eliminacja duplikatów 52
2.8.6 Różnica 57
2.8.7 Unia bez usuwania duplikatów 61
2.9 Test kompresji strumieni dla agregacji 62
2.10 Optymalizacja zapytania w oparciu o monotoniczność strumieni 66
2.11 Wnioski i uwagi 70
Rozdział 3. Język zapytań 74
3.1 Kompilator języka StreamAPAS 82
3.2 Typy danych 83

3.3 Drzewo atrybutów 86
3.4 Język StreamAPAS 90
3.5 Fabryka danych 91
3.6 Składnia Select-From 93
3.6.1 Operacje na zbiorach 93
3.6.2 Klauzula select 95
3.6.3 Klauzula from 97
3.6.4 Klauzula where 98
3.6.5 Klauzula group by i having 100
3.6.6 Operatory agregacji 100
3.7 Bezpośrednie tworzenie obiektów Task 100
3.7.1 Klasa `benchmark` 101
3.7.2 Klasa `gui` 102
3.8 Obiektowa reprezentacja zapytania 102
3.9 Zestaw testowy 105
3.10 CQL 107
3.11 Zapytania w StreamAPAS 110
3.12 Budowa kompilatora 112
3.13 Tabela symboli 113
3.13.1 Drzewo atrybutów 114
3.13.2 Translacja obiektowości StreamAPAS na język Java 119
3.13.3 Przestrzenie nazw 121
3.14 Funkcje 122
3.15 Reprezentacja indeksów oraz operacji na hurtowniach danych 127
3.16 Wnioski i uwagi 129
Rozdział 4. Architektura 132
4.1 Dobór architektury 134
4.1.1 Pojęcia podstawowe 136
4.1.2 Podstawowy model puli wątków 138
4.1.3 Pula wątków zorientowana na szybki transport krotek 141
4.1.4 Architektura operatora fizycznego 142
4.1.5 Transport krotek 144

4.1.6 Implementacja synchronizowanej warstwy zasilania 146
4.1.7 Implementacja lokalnej warstwy zasilania 151
4.1.8 Partycje operatorów 151
4.1.9 Testy 152
4.1.10 Wnioski i uwagi 156
4.2 Modelowanie czasu odpowiedzi 157
4.2.1 Analiza wpływu synchronizacji na czasy odpowiedzi 158
4.2.2 Podstawowy model opóźnień 163
4.2.3 Podejście rozszerzone 165
4.2.4 Testy 167
4.2.5 Wnioski i uwagi...174
4.3 Podział operatorów na partycje 176
4.3.1 Optymalizacja partycji...177
4.3.2 Rozszerzone rozwiązanie 180
4.3.3 Algorytmy tworzenia partycji 185
4.3.4 Testy 186
4.3.5 Wnioski i uwagi...191
Rozdział 5. Podsumowanie 193
5.1 Plany przyszłych prac 194
Bibliografia..198
Dodatek A. Pełna gramatyka języka StreamAPAS 204
Spis symboli i skrótów...208
Spis ilustracji 209
Spis tabel 211

Rozdział 1. Wstęp

W wielu dziedzinach szybko przyrasta liczba danych, które należy przetwarzać w sposób ciągły. Aby sprostać temu wymaganiu coraz częściej korzysta się z przetwarzania strumieniowego. Wyróżnia się ono tym, że nie wszystkie dane są dostępne w chwili uruchomienia zapytania. Takie podejście wymaga innego spojrzenia na źródła danych [30]. Przyjmuje się, że nie są one ograniczone w czasie i wstawiają dane do strumieni na bieżąco. Ponadto system nie ma wpływu na ich kolejność pojawiania się oraz zawartość. Dodatkowo odczyt ze strumienia jest operacją jednorazową i po jego zakończeniu element zostaje usunięty. Należy zauważyć, że nie każde źródło danych w systemie strumieniowym musi być strumieniem, co sprawia, że systemy strumieniowe mogą oprócz przetwarzania strumieniowego wykorzystywać przetwarzanie znane z relacyjnych baz danych.

Do obszarów stawiających największe wymagania wobec systemów strumieniowych należy zaliczyć systemy finansowe takie jak: giełdy, aukcje oraz systemy wykrywające nadużycia. Instytucja OPRA (Options Price Reporitnig Authority), która przetwarza dane wykorzystywane przez zapytania o wartości opcji na giełdzie oraz ich obrotu, tworzy co roku zestawienie opisujące liczbę przetworzonych danych. O ile w roku 2005 intensywność napływu monitorowanych zgłoszeń wynosiła około 100 tysięcy zgłoszeń na sekundę [51], to w roku 2009 ta intensywność przekroczyła 2 miliony [52]. Wysoka dynamika wzrostu tych wartości narzuca konieczność stosowania nowych rozwiązań.

Drugim obszarem generującym dużą liczbę danych wymagających przetwarzania na bieżąco są systemy sensorowe i monitorujące. Rozwój gospodarczy wiąże się z utrzymaniem wielu infrastruktur takich jak: sieci energetyczne, sieci teleinformatyczne, sieci transportowe, sieci wodociągowe i inne. Aby usprawnić ten proces tworzone są aplikacje monitorujące i wspomagające podejmowanie decyzji dla powyższych infrastruktur. Ponadto popularność technologii RFID, postępująca miniaturyzacja i malejąca cena zdalnych czujników oraz powszechna dostępność wielofunkcyjnych urządzeń z wbudowanym odbiornikiem nawigacji satelitarnej

stwarza warunki do powstania nowych zastosowań przetwarzania strumieniowego [82]. Wyróżniamy tutaj prace nad inteligentnymi systemami transportu (ITS), dynamicznym naliczaniem opłat za korzystanie z autostrad [6], usługami opartymi na lokalizacji (LBS) i monitorowaniem zjawisk atmosferycznych. Systemy sensorowe odnajdują również szerokie zastosowanie w wojsku. Przykładem są systemy wspomagające logistykę, monitorujące na bieżąco działania na polu walki, jak również monitorujące parametry witalne żołnierzy w trakcie wykonywania misji. Dzięki zastosowaniu tych technologii uzyskiwana jest większa mobilność oraz znacząco usprawnia się funkcjonowanie zaplecza wojskowego. Obecnie nowym zagadnieniem jest zastosowanie systemów strumieniowych do analizy danych z portali społecznościowych. Liczące od 100 do 700 mln użytkowników portale stają się silnymi opiniotwórczo jednostkami, których analiza na bieżąco jest atrakcyjna zarówno dla instytucji śledzących zagrożenie terrorystyczne; jak również przemysłu, który może lepiej poznać oczekiwania klientów.

W zależności od zastosowania, kładzie się nacisk na odmienne funkcjonalności systemu strumieniowego. W przypadku analiz giełdowych kluczowy jest czas opóźnień. O ile dopuszczalne jest pominięcie części danych o tyle nie akceptowalne są opóźnienia sięgające minut, ponieważ otrzymane wyniki nie będą użyteczne. W przypadku systemów monitorujących bezpieczeństwo w portach lotniczych wyłączenie części danych z analiz w celu skrócenia opóźnień jest nie dopuszczalne. Innym elementem charakteryzującym systemy strumieniowe jest liczba zapytań. W systemach aukcyjnych spotykamy się z pewną grupą zapytań, które często są tworzone i usuwane, podczas gdy w systemach monitorujących zużycie mediów – takich jak: woda lub prąd – liczba zapytań jest niska i rzadko podlegająca zmianom. Powyższa różnorodność sprawia, że budowa systemu ogólnego zastosowania do przetwarzania danych strumieniowych wymaga stworzenia szeregu nowych rozwiązań. Obecnie taki typ systemu został nazwany strumieniową bazą danych SDMS (Stream Database Management System). W [80] wymieniono osiem wymagań stawianych wobec wysoko wydajnych strumieniowych baz danych. Wymagania te definiują jednocześnie następujące kluczowe obszary badawcze:

1) Zgodnie z pierwszym wymaganiem, strumieniowa baza danych SDMS musi przetwarzać dane ze strumieni bez ich wcześniejszego zapisu, co jest konieczne do uzyskania krótkich opóźnień. Rozwiązanie takie stoi

w opozycji do relacyjnych baz danych DBMS (Relational Database Management System), gdzie dane muszą zostać wpierw utrwalone, a dopiero potem mogą zostać przetworzone.

2) Drugim wymaganiem stawianym wobec SDMS jest udostępnienie wysokopoziomowego i rozszerzalnego języka zapytań, który usprawni proces tworzenie zapytań strumieniowych.
3) Przetwarzanie strumieniowe jest procesem ciągłym, który nie może być wstrzymywany na czas nieokreślony. Dlatego ważne są metody obsługi niedoskonałości strumieni takich jak: opóźnienia, utrata danych lub brak ich uporządkowania. W przeciwnym wypadku wystąpienie powyższych zjawisk może doprowadzić do niekontrolowanego wstrzymania przetwarzania.
4) Logika przetwarzania strumieniowego musi być prosta do interpretacji. Tak, aby kompozycja kilku podzapytań była łatwa do interpretacji przez analityka. W szczególności wpływ kolejności, w jakiej napływają dane oraz ograniczenia czasowe, przy których prowadzona jest analiza muszą być łatwe do interpretacji.
5) Rozwój zaawansowanych SDMS wymaga integracji danych strumieniowych z danymi składowanymi.
6) Przetwarzanie strumieniowe jest procesem ciągłym, przez co DSMS powinien dysponować mechanizmami gwarantującymi bezpieczeństwo i dostępność na wypadek awarii.
7) Szybki wzrost liczby danych przetwarzanych strumieniowo oraz ciągły charakter działania nakłada wymaganie, aby strumieniowa baza danych działała w środowisku rozproszonym, w którym DSMS automatycznie równoważy obciążenia.
8) Ostatnie wymaganie obejmuje konieczność opracowania nowych optymalizatorów oraz architektur strumieniowych baz danych.

1.1 Tezy rozprawy

Aby usystematyzować rozwiązywanie kolejnych wymagań stawianych strumieniowym bazom danych zostały one podzielone na komponenty. Zaliczamy tutaj między innymi algorytmy operatorów, polityka kolejności uruchamiania operatorów, rozpraszania danych (load shedding), język zapytań i inne. Problem polega na tym, że komponenty te działają autonomicznie, nie uwzględniając współistnienia innych elementów. W konsekwencji ich potencjalna wydajność jest ograniczona. Do słabych elementów strumieniowych baz danych zaliczamy język zapytań. Wraz ze wzrostem złożoności realizowanych procesów, ważną rolę odgrywają typy złożone, które pozwalają skrócić i uprościć zapis zapytań. Dodatkowo strumieniowa baza danych ogólnego zastosowania powinna umożliwiać dołączanie nowych operatorów po to, aby dostosować ją do specyfiki rozwiązywanego zadania. Przedstawione słabe strony strumieniowych baz danych stanowią motywacje do weryfikacji zapisanych poniżej tez rozprawy.

I. **Możliwe jest rozszerzenie języka zapytań strumieniowych o drzewo atrybutów oraz elementy języka obiektowego, co ograniczy potrzebę zmiany jego składni oraz ułatwi budowę złożonych zapytań.**

II. **Możliwe jest skrócenie czasów odpowiedzi lub zmniejszenie obciążenia pamięciowego poprzez taką integrację komponentów strumieniowej bazy danych, która skutkuje redukcją obszarów synchronizowanych.**

1.2 Organizacja pracy

Rozdział pierwszy stanowi wprowadzenie do zagadnień związanych z tematyką strumieniowych baz danych oraz zawiera tezy rozprawy.

Drugi rozdział pracy przedstawia istniejące algebry stosowane w strumieniowych bazach danych pod kątem ich wymagań pamięciowych oraz ograniczeń w opisie procesów ciągłych. W oparciu o to zestawienie zaproponowane zostało rozwiązanie hybrydowe, które łączy krotki temporalne z krotkami negatywnymi w celu zmniejszenia liczby krotek w strumieniach. Prowadzi

to do minimalizacji zapotrzebowania pamięciowego, jak również zmniejsza wpływ synchronizacji. Dodatkowo do modelu hybrydowego została dołączona analiza monotoniczności strumieni, która służy klasyfikacji operatorów ze względu na złożoność realizacji. W oparciu o tą właściwość zaproponowany został optymalizator regułowy.

Trzeci rozdział opisuje prototypowy język zapytań. Na początku zrealizowany zostaje przegląd istniejących języków zapytań stosowanych w strumieniowych bazach danych w celu zdefiniowania listy wymagań oraz słabości obecnych rozwiązań. W oparciu o to zestawienie zdefiniowano język StreamAPAS łączący elementy języków obiektowych i języków deklaratywnych. Cecha ta jest użyta do minimalizacji potrzeby rozbudowy i zmiany składni języka zapytań podczas rozwoju funkcjonalności strumieniowej bazy danych. Dodatkowo wprowadzono złożony typ nazwany drzewem atrybutów, który poprawia zwięzłość zapisu definicji zapytań strumieniowych. W drugiej części rozdziału opisano elementy architektury kompilatora, dzięki którym osiągnięto wysoki poziom integracji języka z technologiami udostępnianymi w języku Java.

W rozdziale czwartym omówiono moduły strumieniowej bazy danych pod kątem ich wpływu na czasy odpowiedzi. Rezultatem tej analizy jest szereg rozwiązań mających na celu taką integrację istniejących komponentów, aby zmniejszyć opóźnienia wynikające z synchronizacji oraz braku wymiany wiedzy pomiędzy komponentami.

Szósty – ostatni rozdział pracy – jest podsumowaniem nowych idei wprowadzonych do strumieniowej bazy danych. W tym rozdziale znajdują się dowody tez postawionych na początku dysertacji. Punkt ten zawiera także opis planowanych przyszłych prac związanych z rozwojem przedstawionych idei.

Rozdział 2. Algebra przetwarzania strumieniowego

Rozdział ten omawia autorską algebrę przetwarzania strumieni, która łączy rozwiązania oparte na krotkach temporalnych oraz krotkach negatywnych. W RDMS cykl pracy jest zorientowany na zapis. Wpierw dane są zapisywane oraz aktualizowane indeksy. Następnie realizowane są zapytania, których wyniki są reprezentowane, jako tabele rekordów. Z kolei aplikacje strumieniowe są zorientowane na przetwarzanie. Pojedyncza porcja danych tzw. krotka nie jest zapisywana, lecz natychmiast przetwarzana przez sieć połączonych ze sobą operatorów. Ponadto źródła w takich systemach generują strumienie krotek o nieograniczonym rozmiarze. Poniżej zamieszczono główne cechy wyróżniające taki model przetwarzania [15]:

- ciągłe przetwarzania – w RDMS, klient zadaje zapytania na zgromadzonych danych (np. „czy na skrzyżowaniu pojawił się nowy samochód„). W SDMS, użytkownik uruchamia długookresowe zapytanie (np. „powiadom mnie, gdy na skrzyżowaniu pojawi się nowy samochód”).
- „wypychanie” danych – w SDMS, sterowanie jest przekazywane wraz z przepływem danych. Źródła danych oraz operatory strumieniowe przekazują krotki kolejnym obiektom w systemie. Z sytuacją odwrotną spotykamy się w RDMS, gdzie sterowanie jest przekazywane od strony klienta w kierunku danych zgromadzonych w tabelach. Klient najpierw zadaje zapytanie, następnie operatory relacyjnej bazy danych odpytują źródła danych, finalnie jest generowana odpowiedź.
- krótki czas latencji – wiele systemów strumieniowych monitoruje zachodzące zjawiska przy rygorze krótkiego czasu odpowiedzi. Przykładowo dla systemu nadzorującego ruch uliczny ważniejsza jest bieżąca informacja o stanie zatłoczenia ulicy niż informacja o tym stanie sprzed kilku godzin.
- potokowość – przetwarzanie danych bazuje na braku blokowania. Oznacza ona, że nie musi być znany pełen zbiór danych wejściowych, aby wygenerować

odpowiedź cząstkową. Własność ta umożliwia realizację wydajnej współbieżnej pracy operatorów strumieniowych.

- skalowalność – każdy operator strumieniowy stanowi samodzielną jednostkę, co ułatwia stworzenie systemu rozproszonego przetwarzania strumieniowego.
- przepustowość – system strumieniowy jest zorientowany na przetwarzanie danych a nie ich gromadzenie. Brak dziennika logów oraz konieczności uprzedniego zapisu danych skraca czas odpowiedzi. W relacyjnych bazach danych konieczne jest uprzednie zapisanie danych przed realizacją zapytań, co wydłuża czas odpowiedzi.

Zapytanie strumieniowe jest definiowane przy użyciu acyklicznego grafu skierowanego (Directed Acyclic Graph), gdzie węzły reprezentują operatory strumieniowe, a krawędzie odpowiadają połączeniom strumieniowym. Obecnie jest znanych wiele optymalizacji operatorów takich jak selekcja [34,48,72], złączenie [46,58,90,31,33,7,29], agregacja [59] i inne [49]. Jeżeli implementacja operatorów sprowadza się tylko do implementacji zadanych interfejsów programistycznych, wówczas automatyczna optymalizacja taka jak: reorganizacja operatorów i dobór algorytmów jest niedostępna. Jeżeli sformalizowana zostanie algebra opisująca proces przetwarzania strumieniowego, wówczas można zdefiniować reguły tożsamościowe przekształcające zapytanie. W konsekwencji optymalizator SDMS korzystając z nich potrafi automatycznie wybrać najefektywniejszą implementację do realizacji zadanego zapytania. SDMS jest systemem ogólnego przeznaczenia, wiąże się z tym dopuszczenie rozbudowy o dodatkowe operatory charakterystyczne dla budowanego procesu przetwarzania. Z tego punktu widzenia, algebra przetwarzania strumieniowego musi być otwarta na dołączanie operatorów skonstruowanych przez użytkownika. Oznacza to, że powinna ona definiować zbiór zasad, do których musi przystawać każdy algorytm operatora. Gwarantuje to, że operatory niezależnie od miejsca użycia będą poprawnie interpretowały dane wejściowe oraz będą generowały wyniki kompatybilne z innymi elementami systemu. Powyższy opis wskazuje, że algebra przetwarzania strumieniowego musi precyzować co najmniej dwa obszary. Pierwszy z nich to sposób interpretacji strumieni danych oraz sposób opisu operatorów tak, aby użytkownik mógł swobodnie konstruować własne operatory. Drugi obszar obejmuje definicje operatorów standardowych i reguł tożsamości, które stanowią podstawę budowy optymalizatorów.

Przyglądając się historii ewolucji przetwarzania strumieniowego, wyróżniamy dwie drogi:

- Pierwsze podejście jest wynikiem postawienia pytania: jakie operacje należy wykonać na krotce aby osiągnąć wynik.
- Drugie podejście stawia pytanie: przez jaką sieć połączeń operatorów krotka musi zostać przetworzona.

Realizacją pierwszego podejścia jest system ogólnego zastosowania Telegraph [74,60] a następnie TelegraphCQ [23]. Rozwiązanie to wyróżnia się planem produkcji zapytania, który jest tworzony dla każdej krotki indywidualnie, co pozwala osiągnąć lepsza optymalizację. Autorzy tego systemu przeprowadzili szereg udoskonaleń silnika przetwarzania strumieniowego [71], który w kolejnych wersjach miał nazwy: Eddy i PSoup. Słabością tego podejścia jest trudność w zbudowaniu systemu rozproszonego, w chwili, gdy plan produkcji jest dla każdej krotki uzgadniany indywidualnie.

Dlatego większość badań koncentruje się nad systemami, gdzie w danym momencie tylko jeden plan produkcji jest aktywny. Pierwsza implementacja takiego podejścia została wprowadzona w systemie Tapestry [87]. Rozwiązanie to zbudowano jako nakładkę na relacyjną bazę danych. Idea polegała na tym, aby zapytanie strumieniowe zrealizować jako zapytanie przyrostowe. Zgodnie z tym planem, dane strumieniowe są wpierw zapisywane do bazy danych a następnie wyliczana jest różnica, o którą zmieniła się odpowiedź zarejestrowanego zapytania od czasu poprzedniej aktualizacji. Do pierwszej realizacji, gdzie zastosowano operatory strumieniowe zaliczamy system Tribreca [83] zbudowany z myślą o analizie ruchu sieciowego. Niestety system ten jest rozwiązaniem dedykowanym, w którym architektura dopuszcza wyłącznie operatory jednowejściowe, co ogranicza jego rozszerzanie o operatory wielowejściowe, takie jak operator łączenia. Innym przykładem dedykowanego rozwiązania jest system OMCAT [32] służący do obsługi ciągłych zapytań o trajektorie. Do pierwszych systemów ogólnego zastosowania realizujących drugie podejście zaliczamy strumieniową bazę danych Aurora&Borealis [1]. Udostępnia ona możliwość rozszerzania zbioru operatorów o operatory jedno i wielowejściowe. Wadą tego sytemu jest uboga logika opisująca operatory strumieniowe jako czarne skrzynki, które posiadają n wejść i m wyjść. Taki

opis sprawia, że automatyczna reorganizacja operatorów jest niemożliwa. Ponadto użytkownik musi znać szczegóły implementacyjne operatorów, aby efektywnie z nich korzystać. Powyższa słabość stała się powodem intensywnych badań nad opisem logiki operatorów strumieniowych. Pierwszą strumieniową bazą danych definiującą logikę operatorów jest system STREAM [8], który korzysta bezpośrednio z definicji operatorów relacyjnych bazach danych. Osiągnięto to poprzez wprowadzenie operatorów konwertujących strumień w tabelę i tabelę w strumień. Rozwiązanie to jednak nie jest w pełni strumieniowe, ponieważ nie zawiera operatorów przekształcających bezpośrednio strumienie wejściowe w strumienie wyjściowe. Inną logikę zaproponowano w systemie PIPES [56,60]. Słabością tamtego rozwiązania jest ograniczenie algebry do krotek temporalnych.

Obecnie najogólniejsze spojrzenie na logikę przetwarzania strumieniowego zawarto w pracy [88]. Autor tego modelu przyjął, że krotki w strumieniu nie muszą być uporządkowane chronologicznie. Następnie przeprowadził analizę, jakie warunki muszą zostać spełnione, aby operator mógł realizować przetwarzanie strumieniowe. Dużą wadą tej algebry jest opis operatorów na niskim poziomie abstrakcji. Problem można zilustrować następującym przykładem. Operator złączenia jest zdefiniowany poprzez sekwencję operacji na poziomie operacji wejścia wyjścia. Gdy dane zapytanie zostanie przetłumaczone na poziom tych operacji, wtedy ponowna identyfikacja czy dana sekwencja operacji reprezentuje operator złączenia jest zadaniem trudnym. Stanowi to źródło problemów podczas projektowania optymalizatora. Powyższa niedoskonałość sprawiła, że w realizacjach przemysłowych to podejście się nie przyjęło. Z drugiej strony zaproponowany sposób opisu przetwarzania strumieniowego umożliwił zdefiniowanie koniecznych reguł, które musi spełniać każdy algorytm operatora strumieniowego.

Inne podejście do budowy algebry operatorów strumieniowych odnajdujemy w systemie CEDR [16]. Autorzy tego rozwiązania przyjęli, że krotka jest opisana trzema znacznikami: czasem serwera, czasem odnotowania obserwacji i czasem obserwacji. Taka struktura krotki pozwala zbudować zapytanie strumieniowe, które obejmuje zarówno teraźniejszość jak i historie.

Przedstawione w tym rozdziale badania zostały opublikowane w czasopiśmie [44] oraz na międzynarodowej konferencji [41]. Dodatkowo, część

wyników została wydana w obszernym artykule przedstawiającym projekt StreamAPAS w międzynarodowym journalu Complex Intelligent Systems and Their Applications [43].

2.1 Podstawy

Pierwotnym założeniem strumieniowych baz danych było zbudowanie systemu składającego się z analogów do operatorów relacyjnych baz danych. Operatory strumieniowych baz danych ze względu na wymagania pamięciowe dzielimy na dwie grupy:

- **bezstanowe** – operator bezstanowy wylicza wynik w oparciu o krotkę wejściową, bez potrzeby korzystania z dodatkowych danych przechowywanych wewnątrz operatora. Przykładowymi operatorami bezstanowymi są operatory: selekcji i projekcji.
- **stanowe** – operator stanowy wymaga przechowywania dodatkowych danych. Przykładowo, niektóre operatory muszą przechowywać historię krotek, które pojawiły się na wejściu. Zaliczamy tutaj operator różnicy, który wysyła na wyjście te krotki z pierwszego strumienia wejściowego, które nie pojawiły się w drugim strumieniu. Realizacja takiego algorytmu sprowadza się do przechowywania historii krotek, które pojawiły się w drugim strumieniu w wewnętrznej strukturze operatora. Przetworzenie krotki z strumienia pierwszego polega na sprawdzaniu, czy krotka nie występuje w wewnętrznej kolekcji drugiego strumienia; jeżeli nie to zostaje przekazana na wyjście.

Wielu operatorów relacyjnej bazy danych nie można użyć w strumieniowej bazie danych. Wyróżniamy dwie cechy, które ograniczają zastosowanie tych operatorów:

- **blokowalność operatora** – operator musi odczytać pełny zbiór danych wejściowych zanim będzie mógł wygenerować wynik (np. operacja grupowania, sortowania). Użycie takiego operatora skutkowałoby zablokowaniem pracy do momentu otrzymania informacji o zamknięciu strumienia. Strumień nie jest ograniczony w czasie, dlatego operator finalnie nigdy nie przetworzyłby danych. Warto zauważyć, że jeżeli znamy charakter

danych strumieniowych, pojęcie blokowalności staje się bardziej złożonym zagadnieniem. Przyjmijmy, że chcemy pogrupować dane względem atrybutu A oraz wiemy, że krotki uporządkowane są rosnąco względem atrybutu A. Operator grupowania, który w ogólnym zastosowaniu jest blokowany w tym przypadku jest nieblokowany, ponieważ na podstawie napływających danych można zidentyfikować ostatnią krotkę podgrupy. Uogólniając ten przypadek, operator grupowania zdefiniowany w relacyjnych bazach danych można zastosować do przetwarzania strumieni monotonicznie niemalejących względem atrybutów grupowania.

- **nieograniczony stan operatora** – każdy operator w rzeczywistym systemie dysponuje ograniczonym rozmiarem zasobów. Operatory strumieniowe przetwarzają nieograniczone strumienie krotek, zatem jeżeli do wyliczenia wyniku należy przechowywać wszystkie otrzymane wcześniej dane wejściowe, w pewnym momencie praca operatora zostanie przerwana z powodu braku wolnych zasobów. Do grona tych operatorów zaliczamy operatory blokowalne np. sortowanie.

W strumieniowej bazie danych wyróżniamy dwa typy czasów. Rozważmy system monitorujący ruch drogowy, który przesyła wiadomości opatrzone znacznikiem czasowym wskazującym na ich czas wystąpienia. Znaczniki te informują o wystąpieniu wydarzeń w aplikacji, dlatego ten typ czasu nazwano czasem aplikacji. Gdy wydarzenia dotrą do systemu strumieniowego oraz zostaną przetworzone upływa czas, do jego pomiaru służy zegar czasu serwera. Zauważmy, że po stronie źródeł danych leży obowiązek synchronizacji zegarów, aby nie doprowadzić do błędów wynikających z braku synchronizacji napływających danych. Rola zegara serwera ogranicza się do sterowania optymalizacją. Znajomość czasu serwera pozwala zidentyfikować, które krotki najdłużej zalegają w systemie lub zmierzyć obciążenie w systemie rozproszonym. W zależności od aplikacji czas serwera i czas aplikacji mogą być niezależne lub zależne. W dalszej części pracy mówiąc o znacznikach czasu należy mieć na uwadze znaczniki osadzone w czasie aplikacji, o ile nie zostanie zasygnalizowana inna interpretacja.

Poniżej podano operatory relacyjnej bazy danych, które odnajdujemy w strumieniowych bazach danych.

2.1.1 Operatory jednowejściowe (unarne)

Selekcja(σ_p) – operator ten odfiltrowuje ze strumienia wejściowego krotki zgodnie z predykatem p. Krotki, dla których wartość predykatu jest prawdą zostają przekazane na wyjście. Operator ten jest bezstanowy oraz nieblokowany, dzięki czemu może być stosowany w strumieniowej bazie danych bez dodatkowych zmian.

Projekcja(π_A) – projekcja przekazuje na wyjście krotkę składającą się z podzbioru atrybutów krotki wejściowej. Operator ten podobnie jak selekcja może przetwarzać dane strumieniowe bez dodatkowych zmian semantyki.

Eliminacja duplikatów(δ) – operator ten przekazuje na wyjście tylko te krotki, które są unikalne. Do działania tego operatora konieczne jest przechowywanie historii strumienia wejściowego. W konsekwencji mamy do czynienia z operatorem stanowym. Oznacza to, że w takiej postaci nie może być stosowany w strumieniowej bazie danych.

Grupowanie(G_A^{agg}) – operator ten grupuje krotki w oparciu o wartości atrybutów A. Gdy wszystkie dane zostaną odczytane następuje wywołanie funkcji agregującej agg i propagacja wyniku. Operator ten wymaga całego zbioru danych przed wyliczeniem agregatu, dlatego jest on blokowany i nie może przetwarzać danych strumieniowych.

2.1.2 Operatory dwuwejściowe (binarne)

Złączenie($\bowtie_p$) – operator wylicza iloczyn kartezjański w oparciu o dane z dwóch wejść A i B, następnie na wyjście wysyła te pary krotek (a, b) dla których predykat p jest prawdą. Aby wyliczyć iloczyn kartezjański konieczne jest przechowywanie krotek strumieni wejściowych. Operator w tej postaci nie może być zastosowany w strumieniowej bazie danych, ponieważ strumienie posiadają rozmiar nieograniczony.

Unia(∪) – operator ten odczytuje krotki z dwóch wejść i przekazuje je na pojedyncze wyjście, nie jest on blokowany ani nie przechowuje stanu strumieni wejściowych, dzięki czemu można go w takiej postaci stosować dla strumieni danych.

Przecięcie(∩) – operator ten wysyła na wyjście te krotki, które występują w historii obu strumieni wejściowych. Oznacza to, że jest on blokowany jak również ma nieograniczone wymagania pamięciowe.

Różnica(−) – operator ten wysyła na wyjście te krotki ze strumienia pierwszego, które nie mają swoich odpowiedników w strumieniu drugim. Operator ten jest stanowy oraz blokowany, ponieważ musi odczytać wszystkie dane ze strumienia drugiego.

2.2 Kryteria oceny poprawności operatorów strumieniowych

Aby wyjaśnić, jak przenosić operatory występujące w relacyjnej bazie danych do strumieniowej bazy danych zbudowana została teoria strumieni z interpunkcją [88]. Wyróżniamy w niej dwa typy krotek. Krotki z danymi oraz krotki interpunkcji. Każda krotka posiada znacznik identyfikujący typ treści. Interpunkcją nazywamy predykat opisujący, jakie krotki nie napłyną już do strumienia. Przykładowo krotką interpunkcji może być predykat informujący, że wszystkie krotki danych z atrybutem $b < 3$ już napłynęły. Przy użyciu interpunkcji realizowane są trzy cele:

- Operator blokowalny może wygenerować wynik, pomimo że strumień wejściowy nie został zamknięty. Osiągamy to, gdy krotka interpunkcji sygnalizuje, że pewien podzbiór danych został już przesłany. Przykładowo, jeżeli krotka interpunkcji informuje, że podzbiór zawierający atrybuty $a \in [3, 5]$ został przesłany, wtedy operator agregacji może wyznaczyć liczność wystąpień wartości dla atrybutu a w zadanym przedziale.
- Drugim celem osiąganym przez interpunkcję jest usunięcie z operatora własności nieograniczonego stanu. Dla powyższego przykładu, po otrzymaniu krotki interpunkcji i wyliczeniu agregatów dla zadanego podzbioru krotek, mogą one zostać usunięte, ponieważ w przyszłości już nie posłużą do wyliczania wyniku.
- Trzecim zastosowaniem interpunkcji jest integracja pojedynczych operatorów w złożone zapytanie. Osiągnięcie tego wiąże się z koniecznością generowania krotek interpunkcji po to, aby kaskadowo usuwać blokowalność i nieskończony stan operatorów.

Aby sprawdzić czy dany operator może przetwarzać strumienie oraz tworzyć złożone zapytania strumieniowe w [88] zdefiniowano trzy typy niezmienników. Dla uproszczenia i czytelności przyjęto konwencję definiującą operatory jako funkcje przekształcające zbiór krotek wejściowych zaobserwowany od początku działania systemu w zbiór krotek wyjściowych wygenerowany od początku pracy systemu. Zastosowanie tej notacji wymaga przekształcenia definicji przyrostowej operatorów strumieniowych do postaci operującej na zbiorach krotek wejściowych wygenerowanych od początku pracy systemu.

- **niezmiennik przejścia** (*pass invariant*) –Zdefiniowano funkcję *Pass*:

$$Pass(op, T_1, P_1, \dots, T_n, P_n) = T_O \tag{2.1}$$

gdzie:

op – oznacza operator,

T_i – jest zbiorem krotek danych, które napłynęły na wejście i,

P_i – jest zbiorem krotek interpunkcji, które napłynęły na wejście i oraz

T_O – jest zbiorem krotek wynikowych.

Jeżeli dla operatora można zdefiniować funkcję *Pass*, która wyznacza zbiór krotek wynikowych w oparciu o bieżący zbiór krotek wejściowych, wtedy zachodzi niezmiennik przejścia. Reguła ta gwarantuje, że operator jest nieblokowalny.

- **niezmiennik propagacji** (*propagation invariant*) – Zdefiniowano funkcję *Prop*:

$$Prop(T_1, P_1, \dots, T_n, P_n) = P_O \tag{2.2}$$

gdzie:

P_O – reprezentuje zbiór krotek interpunkcji wysłanych na wyjście,

T_i – oznacza zbiór krotek danych na wejściu i oraz

P_i – jest zbiorem krotek interpunkcji na wejściu i.

Operator generuje krotkę interpunkcji wtedy, gdy wszystkie krotki następujące po niej spełnią warunek predykatu. Jeżeli dla operatora można zdefiniować funkcję *Prop*, która wyznacza zbiór krotek interpunkcji w oparciu o bieżący zbiór krotek wejściowy, wtedy zachodzi niezmiennik propagacji. Reguła ta gwarantuje, że krotki interpunkcji są generowane na bieżąco.

- **niezmiennik stanu** (*keep invariant*) – Zdefiniowano funkcję *Keep*:

$$Keep_j(T_1, P_1, \dots, T_n, P_n,) = \hat{T}_j \tag{2.3}$$

gdzie:

T_i – jest zbiorem krotek danych, które napłynęły na wejście i,
P_i – jest zbiorem krotek interpunkcji, które napłynęły na wejście i oraz
$\hat{T}_j$ – reprezentuje zbiór danych, który musi pozostać w operatorze dla wejścia j.

Jeżeli dla wejścia j funkcja $Keep_j$ definiuje ograniczony zbiór krotek tworzący stan operatora, zachodzi wtedy niezmiennik stanu. Niezmiennik ten odpowiada na pytanie, czy operator nie wymaga nieograniczonych zasobów. W odróżnieniu do wcześniejszych niezmienników, jest on zdefiniowany osobno dla każdego wejścia j operatora.

Podsumowując, jeżeli chcemy sprawdzić czy operatory strumieniowe zostały poprawnie zdefiniowane, wystarczy zweryfikować czy: a) niezmiennik stanu gwarantuje ograniczony rozmiar stanu operatora; b) niezmiennik przejścia gwarantuje brak nieblokowalnoci operatora; i c) niezmiennik propagacji zapewnia swobodne komponowanie zapytań w oparciu o pojedyncze operatory strumieniowe. Zaproponowaną metodę cechuje to, że nie nakłada założeń na kolejność, w jakiej napływają krotki w strumieniach i nie ogranicza postaci krotek interpunkcji. Dzięki temu, że niezmienniki zostały zdefiniowane na strumieniach z minimalnymi ograniczeniami konstrukcyjnymi, można je użyć do weryfikacji dowolnego operatora strumieniowego.

2.3 Klasyfikacja strumieni

Rozważmy dwa przykłady. W pierwszym przykładzie, operator wylicza wartość średnią z jaką poruszają się samochody na odcinku drogi przez ostatnie pięć minut. Z chwilą, gdy czujnik zmierzy szybkość kolejnego samochodu, operator wylicza nową wartość średnią i wysyła ją na wyjście. Przy czym nowa wartość na wyjściu dezaktualizuje poprzednią. Drugim przykładem jest zapytanie, które przekazuje informacje o zgłoszeniach aukcyjnych z ostatnich 5 minut. Strumień w tym zadaniu zawiera kolejne zgłoszenia opatrzone znacznikiem czasu t.

Chociaż schemat strumieni w obu przypadkach może być identyczny, ich interpretacja jest skrajnie odmienna. W pierwszym przykładzie obserwujemy jeden byt, nadejście kolejnej krotki prowadzi do dezaktualizacji poprzedniej wartości. W drugim, kolejne krotki oznaczają kolejne nowe byty. Powyższe przykłady zarysowują problem interpretacji strumieni. Jeżeli system składa się z różnych typów strumieni, wtedy nie zachodzi zasada, że strumień może zasilać dowolny operator. Brak pełnej kompatybilności operatorów prowadzi do utrudnień interpretacji oraz ogranicza ponowne użycie podzapytań. Chcąc zbudować system wolny od powyżej słabości, przeanalizujmy istniejące typy strumieni:

- Strumienie z interpunkcją,
- Strumienie wyłącznie z krotkami danych,
- Strumienie z krotkami pozytywnymi i negatywnymi,
- Strumienie temporalne,
- Strumienie z krotkami trój-czasowymi.

Strumienie z interpunkcją zostały zaproponowane w pracy [88]. Przyjęto tam, że strumień jest sekwencją krotek danych i interpunkcji bez nałożonych warunków na ich uporządkowanie. Każda krotka danych należąca do strumienia *S* posiada schemat *T*. Krotka interpunkcji definiuje własności krotek danych, które już nie wystąpią w strumieniu. Przyjmijmy, że strumień *S* przesyła krotki o schemacie składającym się z liczby całkowitej. Przykładowy strumień ma postać: (1), (5), (8), p([0,32)), (40). Oznacza on, że na początku pojawiły się elementy o wartościach: 1, 5, 8; następnie przybyła krotka interpunkcji, wskazująca że wszystkie krotki z wartościami od 0 do 32 zostały wygenerowane; na koniec pojawia się kolejna krotka danych z wartością 40. Taki typ strumienia w najogólniejszy sposób ujmuje dane strumieniowe, co czyni go super-typem dla innych typów. Słabością tego podejścia jest trudność w zdefiniowaniu wydajnych operatorów dla tego typu strumieni. Wynika ona z braku ograniczeń, co do kolejności napływu danych oraz dopuszcza dowolną postać krotek interpunkcji. Z drugiej strony, dzięki uniwersalności tego podejścia zasady zdefiniowane dla tego typu można przenieść na operatory zdefiniowane dla mocniej ograniczonych typów strumieni.

Systemy przetwarzające strumienie składające się wyłącznie z krotek danych wyróżniają się tym, że zawierają krotki uporządkowane zgodnie ze znacznikami

czasu. Każda krotka strumienia *S* ma składnię: *(t, data)* gdzie: *t* – oznacza znacznik czasu wygenerowania oraz *data* – reprezentuje zbiór wartości atrybutów. Wprowadzone uporządkowanie danych względem znacznika *t* można interpretować jako formę realizacji interpunkcji, która sygnalizuje moment wygenerowania wszystkich krotek o znacznikach mniejszych od *t*. W konsekwencji możliwe jest zdefiniowanie wszystkich operatorów strumieniowych zgodnie z regułami niezmienników podanymi w sekcji 2.2. Dodatkowo, aby wynik przetwarzania strumieni był niezależny od kolejności uruchamiania operatorów, przyjmuje się założenie, że operatory n-arne pobierają krotki w kolejności niemalejących wartości znacznika *t*. Strumień tego typu nie przekazuje pełnej informacji o okresie aktywności danych reprezentowanych przez krotki. Cecha ta utrudnia tworzenie i analizę złożonych zapytań, ponieważ pełna interpretacja danych z zadanego strumienia wymaga uwzględnienia operatora, które je wygenerował. Tego typu strumienie są stosowane w systemach: TelegraphCQ [23], STREAM [8], Gigascope [27] i Aurora-Borealis [14].

Definicja strumieni z krotkami pozytywnymi i negatywnymi zakłada, że krotki napływają do strumieni zgodnie z niemalejącym porządkiem znaczników czasowych. Dodatkowo krotka pozytywna reprezentuje pojawienie się pewnego bytu w systemie, podczas gdy krotka negatywna go usuwa. Każda krotka strumienia *S* ma składnię: *(t, <sign>, data)* gdzie: *t* – oznacza znacznik czasowy utworzenia krotki; *<sign>* – przyjmuje wartość **+** dla krotki pozytywnej oraz **–** dla krotki negatywnej; oraz *data* – jest zbiorem wartości atrybutów krotki. Przykładowy strumień ma postać: (1, +, 4), (4, +, 3), (5, -, 3), (6, +, 7). Jego interpretacja jest następująca, wpierw docierają krotki pozytywne o wartościach 4 i 3, następnie krotka negatywna (5, -, 3) dezaktualizuje krotkę (4, +, 3), po czym przybywa kolejna krotka pozytywna o wartości 7. Dzięki wprowadzeniu krotek pozytywnych oraz negatywnych okres aktywności danych można wyznaczyć korzystając tylko z krotek w strumieniu. Ta własność pozwala na zbudowaniu zestawu operatorów, które można ze sobą swobodnie łączyć. Informacje szczegółowe dla tej semantyki można odnaleźć w pracach [36,47].

W strumieniach temporalnych czas życia danych jest definiowany poprzez pojedynczą krotkę. Każda krotka strumienia *S* ma składnię: *([t_s,t_e), data)* gdzie: t_s – jest znacznikiem początku czasu istnienia; t_e – oznacza czas końca życia;

dodatkowo pole *data* – jest zbiorem wartości atrybutów. Aby wyniki generowane przez n-arne operatory były zdeterminowane przyjęto, że krotki są przetwarzane w porządku leksykograficznym $\leq_{t_s,t_e}$, gdzie wpierw następuje uporządkowanie względem atrybutu t_s, a następnie względem t_e. Ten typ strumieni również pozwala na zbudowanie zestawu operatorów, które można swobodnie ze sobą łączyć, ponieważ zawartość strumienia wystarcza do wyznaczenia okresu aktywności danych. Z kolei przedstawienie czasu życia krotki poprzez dwa czasy skutkuje dwukrotnym zmniejszeniem liczby danych w strumieniach w odniesieniu do modelu z krotkami pozytywnymi i negatywnymi. Warto zaznaczyć, że własność ta prowadzi również do odciążenia operatorów. Jeżeli chcemy zastosować przetwarzanie strumieniowe do prognozowania i wspierania decyzji, wówczas prosty dostęp do informacji o czasie aktywności danych jest dodatkowym atutem. Przedstawiony model udostępnia tą własność w przeciwieństwie do wcześniej omówionych rozwiązań. Dokładne omówienie takiego podejścia zaprezentowano w [55,56].

Przedstawione dotąd modele strumieni były tworzone z myślą o systemach przetwarzających wyłącznie dane bieżące. Jeżeli takie dane mają być porównywane z danymi historycznymi, wówczas pojawia się potrzeba rozszerzenia modelu strumieni. Wyobraźmy sobie, że o 21:00 dyskutujemy o wyniku 2:1 dla meczu piłki nożnej, który został rozegrany między 19:00 do 19:45 a wyemitowany przez TV między 20:00 a 20:45. W myśl logiki trój-czasowej krotka reprezentująca mecz może mieć postać: (21:00, [20:00, 20:45), [19:00, 19:45), 2, 1). Godzina 21:00 reprezentuje czas w którym krotkę rozpoczęto analizować przez system strumieniowy. Okres [20:00, 20:45) odpowiada czasowi obserwacji/rejestracji zjawiska, z kolei [19:00, 19:45) definiuje okres występowania zjawiska. Tak zdefiniowany model rozdziela czas analizy od czasu występowania zjawiska, ponadto rozróżnia czas obserwacji od czasu występowania danego wydarzenia. Szczegóły dotyczące budowy operatorów strumieniowych dla takiego modelu można odnaleźć w pracach nad systemem CEDR [54] i Microsoft StreamInsight [3].

Przyjmijmy, że zapytanie Q jest zasilane strumieniem X. Jeżeli interpretacja strumienia X zależy od operatora go zasilającego, wówczas nie można użyć zapytania Q dla innych operatorów zasilających strumień X. Własność ta uniemożliwia hermetyzację zapytania Q, co komplikuje rozwój i pielęgnację systemu

strumieniowego. Przegląd architektur wskazuje na dominację systemów opartych na strumieniach zawierających jedynie krotki danych. O ile taka architektura jest najprostsza od strony implementacyjnej, ogranicza ona sposoby definiowania czasu życia danych. Skutkuje to potrzebą tworzenia kilku wersji interpretacji strumieni, co uniemożliwia hermetyzację zapytań. Aby usunąć powyższą wadę przeprowadzono badania nad typami strumieni. Jeżeli ograniczymy się do strumieni przeznaczonych do analizy danych bieżących, powyższej wady nie posiadają rozwiązania korzystające z krotek pozytywnych i negatywnych, oraz rozwiązanie bazujące na krotkach temporalnych. Wadą pierwszego jest podwójna liczba krotek w strumieniach. Wadą drugiego jest konieczność definiowania czasu życia krotki w chwili jej tworzenia, przy czym wartość ta nie zawsze jest dostępna a-priori.

Przeprowadzona analiza rozwiązań skłoniła mnie do zbudowania autorskiego typu strumieni, który łączy zalety obu wcześniejszych rozwiązań. Istnienie w jednym systemie krotek temporalnych oraz negatywnych pozwoliło zmniejszyć liczbę danych przesyłanych w strumieniach. Operatory zbudowane na takim typie wspierają hermetyzację zapytań. Ponadto rozwiązanie to również otwiera nowe możliwości analizy zmian zachodzących w czasie wcześniej niedostępne. Rozwiązaniami konkurencyjnymi dla tego podejścia jest architektura składająca się z skomplikowanego mechanizmu interpunkcji lub architektura korzystająca z krotek trój-czasowych. Pierwsze rozwiązanie oferuje skąpe możliwości optymalizacji przetwarzania strumieniowego. Drugie podejście jest nadmiarowe, jeżeli rozpatrujemy systemy analizujące dane bieżące.

2.4 Strumienie

Skonstruowana algebra operatorów strumieniowych została oparta na mieszanym typie strumieni składającym się z krotek temporalnych oraz krotek negatywnych. Krotka temporalna reprezentuje pojedyncze wydarzenie, którego czas życia rozpoczyna się w chwili t_s i kończy nie później niż w chwili t_e. Lista wartości atrybutów opisujących to wydarzenie jest zapisana w polu e. Krotka negatywna służy do wcześniejszego wycofywania krotki temporalnej, przy czym jej znacznik t_s jest równy t_e. Użycie krotek negatywnych wiąże się z koniecznością zdefiniowania klucza

głównego, który składa się z atrybutów zapisanych w polu *e*. Przy jego użyciu identyfikowana jest krotka temporalna, do której odwołuje się krotka negatywna. Strumień typu mieszanego definiuje także krotkę graniczną (Boundary [15]) składającą się wyłącznie ze znaczników czasu. Sygnalizuje ona, że napłynęły wszystkie krotki ze znacznikami mniejszymi lub równymi od jej znaczników t_s, t_e zgodnie z porządkiem leksykalnym. Poniżej podano formalną definicję strumienia.

Definicja 2.1. Strumień

Niech T będzie dyskretną dziedziną czasu; wtedy $I \coloneqq \{[t_s, t_e) | t_s, t_e \epsilon T \wedge t_s \leq t_e\}$ *jest zbiorem możliwych interwałów czasu. Para* $S = (M, \leq_{t_s,t_e})$ *jest strumieniem jeżeli:*

- *M – jest nieskończoną sekwencją krotek* $([t_s, t_e), type, e,)$
- *gdzie:*
  $[t_s, t_e) \in I$
  type – typ krotki:
    - *+ – pozytywna,*
    - *- – negatywna,*
    - *B – graniczna (Boundary).*
  e – pole zawierające wartości atrybutów krotki
- *Wszystkie elementy M posiadają ten sam schemat danych*
- $\leq_{t_s,t_e}$ *– porządek leksykograficzny na sekwencji M odpowiednio względem* t_s *i* t_e

Definicja 2.2. Operacja redukcji krotek

Operacja redukcji krotek polega na zastąpieniu sekwencji składającej się z krotki temporalnej i krotki negatywnej o tej samej wartości klucza głównego pojedynczą krotką temporalną, której czas życia jest skrócony zgodnie z znacznikiem t_e *krotki negatywnej.*

Definicja 2.3. Strumień domknięty dla chwili *t*

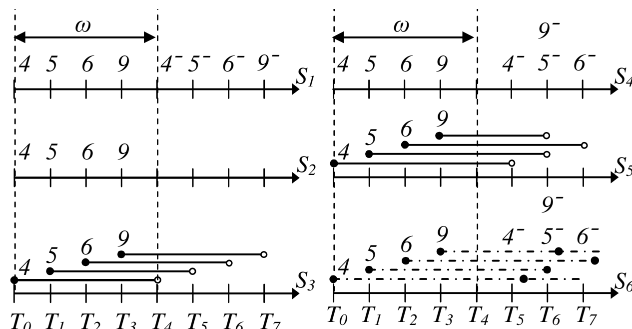
Strumień domknięty dla chwili t, zawiera krotki ze znacznikami t_s *większymi lub równymi t.*

Definicja 2.4. Prefiks strumienia dla krotki *t*

Prefiksem strumienia dla krotki t nazywamy sekwencję krotek poprzedzającą krotkę t. Przy czym krotkę t również włączamy do tej sekwencji.

Definicja 2.5. Tabela historii

Funkcja realizująca konwersję strumienia w tabelę historii będzie oznaczana $H = \mathcal{S}[\![S]\!]$. *Na sekwencji krotek ze strumienia S wykonana jest operacja redukcji krotek, następnie usunięte są krotki graniczne. W wyniku powstaje zbiór krotek temporalnych, który jest ładowany do tabeli historii H będącej relacją zawierającą rekordy o schemacie* ($[t_s, t_e)$, *e*).



Rys. 2.1. Zestawienie różnych definicji strumieni

Aby przybliżyć budowę mieszanego typu strumienia rozważmy przykład ilustrujący różnice pomiędzy typem mieszanym a rozwiązaniami konkurencyjnymi. Przyjmijmy, że monitorujemy pewne zjawisko opisane kilkoma atrybutami, ponadto każda ze zmian ma ustalony czas występowania. Strumienie S_1, S_2 oraz S_3 na rys. 2.1 opisują obserwację tego samego procesu. Różnica polega w realizacji powiadamiania o czasie wystąpienia i trwania kolejnych wydarzeń:

a) W strumieniu S_1 początek wydarzenia sygnalizuje krotka pozytywna a koniec wydarzenia sygnalizuje krotka negatywna.

b) Jeżeli czas trwania każdego wydarzenia jest identyczny i wynosi ω jednostek czasu, wówczas wystarczy tylko transmisja krotek pozytywnych. Znając bieżący czas oraz interwał czasu można bezpośrednio wyliczyć krotki aktywne. Przypadek ten obrazuje strumień S_2.

c) Jeżeli czas trwania wydarzenia jest z góry znany w chwili zaobserwowania, znacznik początku życia jak i znacznik końca życia można zapisać w chwili tworzenia krotki, co ilustruje strumień S_3.

Analogicznie do powyższego przykładu, strumienie S_4, S_5 oraz S_6 reprezentują zapis obserwacji innego procesu. Jeżeli znamy z góry czas trwania wydarzeń, możemy dwukrotnie zmniejszyć liczbę krotek w strumieniu przy użyciu modelu temporalnego, co ilustrują strumienie S_4 i S_5. Z drugiej strony reprezentowanie wydarzeń jest utrudnione przy użyciu tego modelu [56], gdy nie znamy czasu ich trwania w chwili zgłaszania. Wtedy skorzystanie z modelu z pozytywnymi i negatywnymi krotkami prowadzi do zmniejszenia liczby krotek w strumieniu. Stworzony nowy model strumienia łączy zalety obu rozwiązań. Strumień S_6 ilustruje wydarzenia, dla których czas aktywności nie jest dostępny na wstępie. Dlatego tworzone są krotki o nieskończonym czasie życia. Jeżeli znane jest przybliżone górne ograniczenie czasu aktywności danych, wartość tą wstawiamy do znacznika końca życia krotki. Krotka negatywna jest wstawiana tylko wtedy, gdy monitorowane wydarzenie zakończy się zanim czas aplikacji zrówna się z znacznikiem końca życia. Aby zaznaczyć fakt, że czas życia krotki jest tylko ograniczony przez górną granicę, linie reprezentujące okres aktywności krotki na rys. 2.1 są przerywane.

2.5 Monotoniczność strumieni

W zależności od źródła, strumień może zawierać wyłącznie krotki temporalne albo zarówno krotki temporalne jak i negatywne. Znajomość tej własności pozwala dobrać wydajniejszy algorytm realizujący zadany operator strumieniowy. Przykładowo, jeżeli strumień zawiera wyłącznie krotki temporalne nie ma konieczności, aby operator tworzył strukturę indeksującą krotki względem klucza głównego.

Do opisu powyższych własności zaadaptowano klasyfikację monotoniczności strumieni opisaną w [37]. Definicję monotoniczności strumieni w tym ujęciu przedstawiono jako własność operatora. Niech Q oznacza zapytanie strumieniowe a τ reprezentuje pewien punkt na osi czasu. Przyjmijmy, że w chwili τ wszystkie krotki ze znacznikami t_s mniejszymi lub równymi τ zostały już przetworzone. Zbiór krotek

wejściowych w chwili τ jest oznaczony przez $S(\tau)$, podczas gdy wszystkie krotki od czasu *0* do chwili bieżącej są reprezentowane przez $S(0,\tau)$. Ponadto, niech $P_S(\tau)$ jest zbiorem wynikowym wyprodukowanym w chwili τ oraz niech $E_S(\tau)$ jest zbiorem krotek, które wygasły w chwili τ. Poniższe równanie definiuje funkcję aktualizacji wyniku:

$$\forall\tau\ Q(\tau+1) = Q(\tau) \cup P_S(\tau+1) - E_S(\tau+1) \tag{2.4}$$

Wyróżniamy następujące typy monotoniczności operatorów:

1. Operator monotoniczny produkuje krotki, których czas życia nie wygasa. Formalnie opisujemy taki operator równaniem:

$$\forall\tau\forall\mathrm{S}\ E_S(\tau) = 0 \tag{2.5}$$

2. Najsłabiej niemonotonicznym operatorem nazywamy taki, którego krotki wynikowe mają znany i stały czas życia. Cecha ta sprawia, że porządek w którym krotki rozpoczynają czas życia jest identyczny z porządkiem ich wygasania. Formalnie cechę tą zapisujemy:

$$\forall\tau\forall\mathrm{S}\ \exists c \in \mathbb{N}_+\ E_S(\tau) = P(\tau - c) \tag{2.6}$$

3. Jako słabo monotoniczny operator definiujemy taki, którego czasy życia krotek są znane w chwili tworzenia, lecz są one różne. Czyli porządek, w którym krotki rozpoczynają czas życia oraz porządek ich wygasania są różne. Monotoniczność tą definiujemy:

$$\begin{gathered}\text{Jeżeli } \forall\tau\forall S\ \forall S'\ S(\tau) = S'(\tau) \text{ wtedy:}\\ \forall t \in P_S(0,\tau)\exists r\ t \in E_S(r) \wedge t \in E_{S'}(r)\end{gathered} \tag{2.7}$$

   Wyrażenie to tłumaczymy następująco: każda krotka wynikowa ma ograniczony czas życia, przy czym czas ten nie zależy od krotek, które napłyną w przyszłości.

4. Mocno niemonotonicznym operatorem nazywamy taki, który w chwili tworzenia krotek wynikowych nie może podać ich czasu wygasania. Czas ten zależy od krotek, które pojawią się w przyszłości na wejściu operatora. Formalnie tą monotoniczność opisujemy:

$$\begin{gathered}\text{Jeżeli } \exists\tau\ \exists S\ \exists S'\ S(0,\tau) = S'(0,\tau) \text{ wtedy:}\\ \exists r\exists t \in P_S\big(0, \tau\big)\ t \in E_S(r) \wedge t \notin E_{S'}(r)\end{gathered} \tag{2.8}$$

Przedstawione typy monotoniczności zostały zilustrowane na rys. 2.1. Strumień S_3 reprezentuje monotoniczność typu 2. Strumień S_5 jest monotoniczności typu 3. Ostatni typ monotoniczności ilustruje S_6.

Monotoniczności strumienia przekłada się na złożoność jego interpretacji. Czym wyższy stopień monotoniczności tym bardziej złożone jest działanie struktury przechowującej krotki dla operatorów stanowych. Monotoniczność typu pierwszego oznacza, że krotki strumienia nigdy nie wygasają a zatem strumień tego typu składa się wyłącznie z krotek temporalnych o nieskończonym czasie życia. Dla typów monotoniczności 2 i 3 działanie kolekcji sprowadza się do sprawdzania, dla których krotek upłynął czas życia. W przypadku typu 2 wiemy dodatkowo, że kolejność usuwania krotek z kolekcji odpowiada kolejności, w której zostały one wstawione. Wiedza ta sugeruje implementację opartą na prostej kolejce FIFO. W przypadku typu 3 realizację operacji usuwania krotek wygasłych można przeprowadzić na dwa sposoby. Albo sprawdzana jest każda krotka kolekcji albo tworzona jest kolejka priorytetowa porządkująca krotki w porządku wygasania, co przyspiesza etap wyszukiwania krotek do usunięcia.

Jeżeli strumień jest mocno niemonotoniczny wtedy krotka temporalna wskazuje na potencjalny czas ważności wydarzenia. Jeżeli czas życia wydarzenia ulega zmianie, wstawiana jest krotka negatywna. W [36] przyjęto, że krotka negatywna i pozytywna są identyczne gdy posiadają te same wartości atrybutów. W celu przyspieszenia przetwarzania krotek negatywnych, w rozwijanym autorskim systemie przeniesiono z RDMS do systemu strumieniowego pojęcie klucza głównego(PK). O ile w innych strumieniowych bazach danych konieczne jest porównanie wszystkich atrybutów krotki, zaproponowane rozwiązanie umożliwia utworzenie indeksu haszującego na atrybutach klucza głównego. Zauważmy, że w przeciwieństwie do RDMS w strumieniowej bazie danych, tylko dla strumieni mocno nie monotonicznych uzasadnione jest budowanie indeksu na kluczu głównym. Aby zabezpieczyć spójność operatora strumieniowego na wypadek wystąpienia krotek o identycznych wartościach klucza głównego operator odfiltrowuje duplikaty. Podobną strategie implementują DBMS, które nie zezwalają na wstawienie rekordu o identycznej wartości PK.

2.6 Operatory

Semantyka operatorów w autorskiej strumieniowej bazie danych jest zdefiniowana przez operatory logiczne. Opis na tym poziomie abstrakcji odseparowuje logikę przetwarzania od algorytmów implementujących tą funkcjonalność.

Definicja 2.6. Operator logiczny

Operatorem logicznym nazywamy funkcję, która przekształca wejściowe tabele historii w wynikową tabelę historii. Operator logiczny ***o*** *zdefiniowany na n wejściowych tabelach historii: H_1, H_2,..., H_n będzie oznaczany:* $o[\![H_1, H_2, \dots, H_n]\!]$.

Tabela historii jest relacją, co pozwala opisać operatory strumieniowe korzystając bezpośrednio z definicji operatorów relacyjnych baz danych.

Definicja 2.7. Operator fizyczny

Operatorem fizycznym ***f*** *nazywamy algorytm, który przekształca n strumieni wejściowych: S_1, S_2,..., S_n w strumień S_{out}:* $S_{out} = o[\![S_1, S_2, \dots, S_n]\!]$. *Jeżeli operator fizyczny posiada kilka wejść, wtedy do przetworzenia pobierana jest krotka z najmniejszym znacznikiem czasu spośród wszystkich strumieni wejściowych. Warunek ten zapewnia determinizm w działaniu operatora. Jeżeli operator* ***o*** *ma co najmniej jeden ze strumieni wejściowych pusty, wtedy nie można wyznaczyć wartości minimalnej. Mówimy wtedy, że operator* ***o*** *nie jest gotowy do przetwarzania danych; w przeciwnym wypadku operator jest gotowy do przetwarzania. Ponadto algorytm operatora fizycznego* ***f*** *spełnia warunki określone przez niezmienniki: przejścia, stanu i propagacji.*

Każdy operator fizyczny odmierza czas lokalny, wraz z odczytem kolejnej krotki *t* czas lokalny jest ustawiany na $t.t_S$. Czas ten obrazuje, jaka cześć danych wejściowych została przetworzona. Korzystając z czasu lokalnego operatora definiujemy aktywne krotki w tabeli historii, jako zbiór krotek z czasem życia obejmującym lokalny czas operatora. Pozostałą grupę krotek w tabeli historii zaliczamy do wygasłych.

Definicja 2.8. Implementacja operatora logicznego

*Operator fizyczny **f** implementuje operator logiczny **o** wtedy i tylko wtedy gdy:* $o[\![\mathcal{S}[\![S_1, S_2, \dots, S_n]\!]]\!] = \mathcal{S}[\![o[\![S_1, S_2, \dots, S_n]\!]]\!]$. *Wzór ten oznacza, że dla dowolnego domknięcia strumieni wejściowych; tabela historii zbudowana na strumieniu wygenerowanym przez operator fizyczny jest tożsama z wynikiem operatora logicznego zbudowanego na tabelach historii strumieni wejściowych.*

Na potrzebę dalszych analiz rozważmy dwa przypadki, kiedy operator fizyczny spełnia niezmiennik stanu:

- Jeżeli operator fizyczny jest bezstanowy, wówczas stan operatora jest zbiorem pustym, a zatem zachodzi niezmiennik stanu.
- Zgodnie z definicją 2.5, liczba krotek aktywnych w tabeli historii ma ograniczony rozmiar, gdy zachodzi jeden z warunków:
    - po uprzednim przetworzeniu krotek negatywnych każdy rekord tabeli historii posiada znacznik t_e mniejszy od nieskończoności,
    - liczba krotek w strumieniu jest ograniczona.

  W strumieniowej bazie danych rozmiar tabeli historii jest ograniczony, tylko gdy zachodzi pierwszy warunek, ponieważ strumień nie jest ograniczony rozmiarem. Podsumowując, operator fizyczny spełnia niezmiennik stanu, jeżeli przechowuje tylko dane zawarte w tabelach historii dla strumieni wejściowych i wyjściowych, przy czym zbiór ten ogranicza się do krotek aktywnych. Ponadto strumienie wejściowe nie zawierają krotek o nieskończonym czasie życia.

2.7 Operatory okien

Zgodnie z definicją strumienie danych są nieograniczone w czasie. Przechowywanie wszystkich danych przekazywanych w strumieniu jest technicznie nie wykonalne. Aby operatory stanowe przechowywały ograniczoną liczbę krotek wprowadzono do strumieniowych baz danych operatory okien. Ich rolą jest ograniczenie okresu aktywności krotek, co prowadzi do zmniejszenia liczby krotek w tabeli historii. Okna dzielimy na dwie grupy. Okna czasowe, które definiują czas życia krotki w oparciu o wartości znaczników czasu. Przyjmijmy, że chcemy wyliczać

średnią szybkość samochodów, które pojawiły się na pewnym odcinku drogi w przeciągu ostatniej godziny. Na badanej drodze zainstalowano czujnik rejestrujący szybkość pojazdu oraz czas pomiaru. Wyznaczenie wartości średniej wymaga zastosowania okna czasowego, które każdej krotce t ustawia wartość $t.t_e$ = $t.t_s$ + 1 godzina. W konsekwencji pomiar t będzie uwzględniany w agregatach przez następną godzinę. Omówiony operator w literaturze nazwano oknem czasowo-przesuwnym. Do drugiej grupy operatorów zaliczamy okna fizyczne. Operatory te definiują ograniczenia czasu życia krotek, korzystając z innych własności niż tylko znaczniki czasu. Powracając do przykładu drogowego. Tym razem chcemy wyznaczyć średnią szybkość n ostatnich samochodów na wskazanym odcinku drogi. W tym celu należy stworzyć bufor n elementowy. Jeżeli na wejściu pojawi się nowa krotka a bufor jest wypełniony, wpierw dezaktualizowana jest najstarsza krotka przy użyciu krotki negatywnej. Następnie na wyjście przekazywana jest nowa krotka. Opisane okno w literaturze nazwano liczebnościowym. Tablica historii zasilona przez takie okno będzie zawierała, co najwyżej n aktywnych krotek.

Podsumowując, okna czasowe pozwalają skrócić czas życia krotek, w konsekwencji operatory stanowe zasilane takimi strumieniami mają mniejsze wymagania pamięciowe. Z drugiej strony okno czasowe nie determinuje liczby krotek aktywnych w tabeli historii, ponieważ liczba elementów zależy od intensywności strumienia. W konsekwencji okna czasowe nie gwarantują, że rozmiar kolekcji lokalnej operatora nie przekroczy zadanego limitu. Okna fizyczne pozwalają na sztywne ograniczenie rozmiaru kolekcji lokalnych. W przypadku okna liczebnościowego, bezpośrednio ograniczamy liczbę krotek aktywnych w dowolnym momencie czasu. Stosując okna fizyczne należy być świadom pewnego specyficznego procesu. Przyjmijmy, że w strumieniu znajduje się m krotek z tymi samymi znacznikami t_s. Okno liczebnościowe ma rozmiar n. Jeżeli $m > n$, wtedy część krotek temporalnych z identycznymi wartościami t_s zostanie usunięta przez krotkę negatywną posiadającą $t_e = t_s$. Prowadzi to do powstania krotek o zerowym czasie życia. Z punktu widzenia logiki temporalnej [55] są one elementami neutralnymi, ponieważ krotka o zerowym czasie życia nie może być źródłem krotki o niezerowym czasie życia. Rozważmy zapytanie, składające się z połączonych kaskadowo dwóch operatorów zasilanych różnymi oknami liczebnościowymi. Okna liczebnościowe zliczają wszystkie krotki zarówno o zerowym jak i niezerowym czasie życia.

W konsekwencji krotki o zerowym czasie życia nie są elementami neutralnymi dla powyższego zapytania. Istnienie tego zjawiska nie zakłóca przetwarzania strumieniowego, ponieważ operatory nadal przetwarzają strumienie w sposób deterministyczny. W razie konieczności najbezpieczniejszym rozwiązaniem jest zdefiniowanie operatora tak, aby nie generował krotek o zerowym czasie życia. Przykładowo, gdy wartość znacznika czasu krotki do usunięcia jest równa znacznikowi czasu krotki przybyłej kolekcja jest tylko rozszerzana o nowy element. Dopiero po nadejściu krotki z większym znacznikiem czasu, z kolekcji wymiatane są krotki w celu ograniczenia rozmiaru bufora do n elementów.

2.7.1 Okno przesuwno-czasowe

Operator logiczny definiujemy:

$$winSlide_{\omega}(H) = \{([t_s, t_s + \omega), +, e) | ([t_s, t_e), e) \in H\} \tag{2.9}$$

Operatora ten iteruje po wszystkich elementach tabeli historii H. Jeżeli na wejściu jest krotka temporalna, wtedy na wyjście jest generowana krotka o identycznych wartościach atrybutów e ale o czasie życia ω. Jeżeli na wejściu pojawi się krotka negatywna jest ona usuwana, ponieważ każda krotka ma czas życia równy ω. Dodatkowo krotka graniczna jest przekazywana niezmieniona z wejścia na wyjście.

Operator ten jest bezstanowy, dzięki czemu zachodzi niezmiennik stanu. Niezmiennik propagacji jest zachowany, ponieważ krotki temporalne oraz graniczne są przekazywane na wyjście zgodnie z porządkiem leksykograficzny. Niezmiennik przejścia jest osiągany, ponieważ operator ten jest bezstanowy.

Zauważmy, że kiedy każda krotka ma ten sam czas życia, kolejność krotek uporządkowanych ze względu na t_s jest identyczna z uporządkowaniem względem t_e. Powyższa własność sprawia, że strumień wynikowy tego operatora jest najsłabiej nie monotoniczny niezależnie od typu monotoniczności strumienia wejściowego.

2.7.2 Okno stało-czasowe

Operator logiczny definiujemy:

$$winFixed_{\omega}(H) = \left\{ \begin{matrix} ([t_s, (n+1)\cdot\omega), e) | \\ ([t_s, t_e), e) \in H \wedge t_s \geq n\cdot\omega \wedge t_s < (n+1)\cdot\omega \end{matrix} \right\} \quad (2.10)$$

gdzie *n* jest liczbą całkowitą.

Przyjmijmy, że chcemy zmierzyć liczbę zgłoszeń na aukcji w kolejnych godzinach: 12:00, 13:00, 14:00, …, 24:00. W takim zapytaniu istnieją z góry narzucone interwały naliczania agregatów sumy, stąd pochodzi nazwa okno stało-czasowe. Niezależnie czy krotka przybyła o godzinie 12:15, 12:19 czy 12:50 jej nowy czas życia będzie kończył się o 13:00.

Implementacja tego operatora wyróżnia dwa przypadki. Jeżeli na wejściu jest krotka temporalna, wtedy na wyjście jest generowana krotka z czasem życia $[t_s, (n+1)\cdot\omega)$ i identyczną wartością atrybutów *e*; z kolei krotki negatywne są usuwane. Dodatkowo krotka graniczna jest przekazywana niezmieniona z wejścia na wyjście. Weryfikacja niezmienników dla tego operatora jest taka sama jak dla operatora okna przesuwno-czasowego.

Zauważmy, że operator ten przekazuje na wyjście krotki w takiej kolejności, jakiej napłynęły. Ponadto wartości kolejnych znaczników t_e tworzą niemalejący ciąg wartości. Powyższe własności oznaczają, że strumień wynikowy okna stało-czasowego jest typu najsłabiej nie monotonicznego.

2.7.3 Okno liczebnościowe

Okno liczebnościowe $winCnt_n(S)$ ogranicza rozmiar tabeli historii do *n* aktywnych elementów. Definicję tego operatora przedstawia alg. 2.1.

Implementacja operatora korzysta z kolejki FIFI *buf*. Przetwarzanie krotki temporalnej *t* rozpoczyna się od utworzenia krotki t_{new}, która w odróżnieniu do krotki *t* posiada czas życia zdefiniowany jako nieskończony. Jeżeli czas życia krotek wynikowych nie zostałby ustawiony na nieskończoność, wtedy liczba elementów byłaby mniejsza od *n*, gdy czas życia krotki upłynąłby zanim nowa krotka napłynęłaby do systemu.

Algorytm 2.1. Algorytm okna liczebnościowego.

```
Process(Tuple t)
1)if t jest temporalna
2)      utwórz krotkę t_new = ([t.ts, MAX_VALUE), t.e)
3)      if n == buf.size()
4)            usuń pierwszą krotkę z buf i zapisz ją w zmiennej t_r
5)            utwórz negatywną krotkę t_n = ([t.t_s, t.t_s), t_r.e)
6)            dodaj t_n do OUT
7)      dodaj t_new na koniec buf
8)      dodaj t_new do OUT
9)if t jest typu Boundary
10)     dodaj t do OUT
```

Operator ten jest zgodny z niezmiennikiem stanu, ponieważ przechowuje tylko *n* ostatnich krotek strumienia. Niezmiennik przejścia jest spełniony, ponieważ wystarczy znajomość *n* ostatnich krotek z wejścia, aby wyznaczyć wynik po przetworzeniu bieżącej krotki *t*. Niezmiennik propagacji jest zachowany, ponieważ krotki temporalne i krotki graniczne są uporządkowane zgodnie z porządkiem leksykograficznym.

Strumień wynikowy okna liczebnościowego posiada zarówno krotki temporalne jak i negatywne. Oznacza to, że strumień wynikowy jest mocno nie monotoniczny i własność ta nie zależy od monotoniczności typu strumienia zasilającego.

2.8 Operatory pochodzące z relacyjnych baz danych

W kolejnych punktach zdefiniowane zostaną operatory logiczne i fizyczne będące odpowiednikami operatorów relacyjnych baz danych dla strumieniowych baz danych. Każdy z tych operatorów logicznych jest zdefiniowany przy użyciu tabeli historii. Implementacja omawianych operatorów fizycznych przechowuje wyłącznie aktywne krotki w tabeli historii ponadto korzysta z def. 2.8. W wyniku tego zgodnie z wnioskowaniem przeprowadzonym pod koniec sekcji 2.6 operatory takie spełniają niezmiennik stanu.

2.8.1 Operator selekcji

Operator logiczny definiujemy:

$$\sigma_p(H) = \{([t_s, t_e), e) | ([t_s, t_e), e) \in H \wedge p(e)\} \quad (2.11)$$

gdzie p jest predykatem zbudowanym na atrybutach krotki.

Implementacja fizyczna operatora polega na wyliczaniu predykatu p dla każdej krotki pozytywnej i negatywnej. Jeżeli wartość predykatu jest prawdą, wtedy krotka jest przekazywana na wyjście. Dodatkowo każda krotka graniczna z wejścia jest przekazywana na wyjście.

Operator fizyczny spełnia niezmiennik stanu, ponieważ jest bezstanowy. Operator implementuje niezmiennik przejścia, ponieważ krotki wynikowe są generowane na bieżąco z chwilą przetworzenia krotki wejściowej. Operator fizyczny implementuje niezmiennik propagacji, gdy wszystkie krotki następujące po interpunkcji spełnią jego treść predykatu. Rolę interpunkcji w modelu mieszanym strumieni pełnią krotki specjalne: negatywne i graniczne oraz warunek determinujący uporządkowanie leksykograficzne krotek. Operator selekcji spełnia niezmiennik propagacji, ponieważ zachowuje uporządkowanie leksykograficzne krotek temporalnych, negatywnych i granicznych. Dodatkowo operator filtracji zachowuje spójność danych, ponieważ usuwa równocześnie odpowiadające sobie krotki temporalne i negatywne.

Operator selekcji nie zmienia czasu życia krotek, dlatego monotoniczność strumienia wynikowego jest równa monotoniczności strumienia wejściowego.

2.8.2 Operator mapujący

Operator logiczny definiujemy:

$$map_f(H) = \{([t_s, t_e), f(e)) | ([t_s, t_e), e) \in H\} \quad (2.12)$$

gdzie funkcja f wylicza atrybuty krotki wynikowej. W przeciwieństwie do operatora projekcji, który przepisuje do krotki wynikowej podzbiór atrybutów z krotki wejściowej, operator mapujący zezwala na użycie operacji arytmetycznych do wyliczenia atrybutów krotek wynikowych.

Implementacja fizyczna polega na wyliczeniu wartości funkcji f dla każdej krotki pozytywnej i negatywnej. Dodatkowo każda krotka graniczna z wejścia jest przekazywana na wyjście.

Operator mapowania spełnia niezmienniki stanu, przejścia i propagacji, ponieważ tak jak operator selekcji: jest operatorem bezstanowym, nie zmienia porządku krotek oraz propaguje krotki graniczne z wejścia na wyjście.

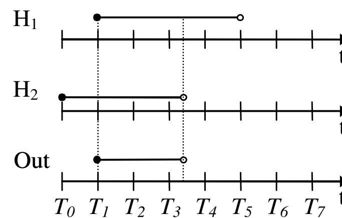
Operator selekcji nie zmienia czasu życia krotek, dlatego monotoniczność strumienia wynikowego jest równa monotoniczności strumienia wejściowego.

2.8.3 Operator Θ złączenie

Operator logiczny zdefiniowano:

$$\times_{\Theta}(H_1, H_2) = \left\{ \begin{matrix} ([\max(t_{s1}, t_{s2}), \min(t_{e1}, t_{e2})), e_1 \circ e_2) | \\ ([t_{s1}, t_{e1}), e_1) \in H_1 \wedge ([t_{s2}, t_{e2}), e_2) \in H_2 \wedge \Theta(e_1, e_2) \end{matrix} \right\} \quad (2.13)$$

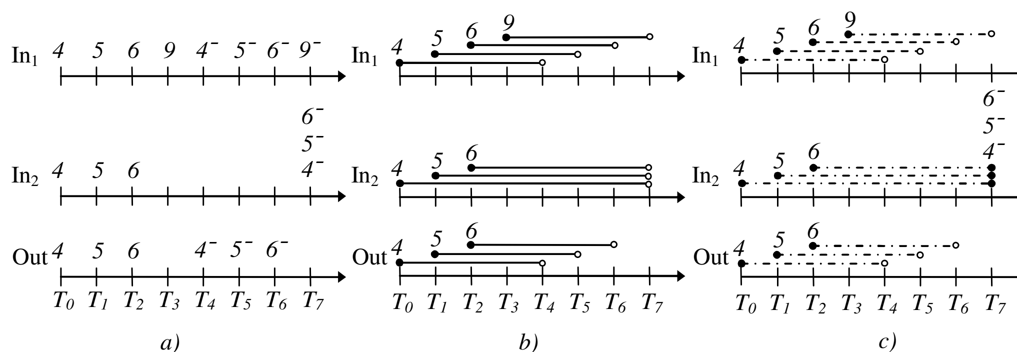
Operator Θ złączenie jest złączeniem warunkowym dwóch tabel historii H_1 i H_2 zgodnie z warunkiem Θ. Zawartość pola danych rekordu wynikowego jest konkatenacją pól e_1 i e_2. W odróżnieniu do relacyjnych baz danych rekordy tabeli historii precyzują czas życia, skutkuje to tym, że rekord wynikowy Θ złączenia posiada czas życia będący częścią wspólną rekordów składowych. Aby przybliżyć zasadę łączenia dwóch rekordów, rysunek 2.2 ilustruje zawartość tabel historii H_1 i H_2 oraz wynik *Out*.



Rys. 2.2. Przykładowy wynik operatora złączenia danych temporalnych

Aby przedstawić zalety omawianej algebry operatorów strumieniowych na rys. 2.3 zestawiono wyniki pracy operatora Θ złączenia dla trzech modeli strumieni:

a) dla modelu przetwarzania z krotkami pozytywnymi oraz negatywnymi [36],
b) dla modelu temporalnego [56],
c) dla modelu mieszanego.



Rys. 2.3. Rezultat łączenia strumieni dla różnych podejść

Przyjmijmy, że operator złączenia realizuje operację równości. Każda krotka na rys. 2.3 opisana jest liczbą reprezentującą wartość atrybutu, na którym jest zdefiniowany warunek złączania. Strumienie w zestawach a), b) i c) przedstawiają obserwację tego samego szeregu wydarzeń, różnica polega na sposobie ich prezentacji przy użyciu krotek. W opisie użyto terminów kolekcja 1 i kolekcja 2, które oznaczają struktury danych przechowujące krotki odpowiednio ze strumieni 1 i 2.

Analizę implementacji operatorów złączenia rozpoczniemy od podejścia temporalnego. Budowa operatora złączenia jest symetryczna dla obu wejść, dlatego analizę ograniczono do wejścia pierwszego. Przyjmijmy, że przetwarzamy krotkę t z wejścia pierwszego. Operator złączenia korzysta z kolejki *queue*, która porządkuje krotki zgodnie z porządkiem leksykograficznym. Wstawiane są do niej wyniki cząstkowe. Wywołanie funkcji *sweep(t)* wymiata z kolejki krotki o znacznikach t_e mniejszych od $t.t_s$. Działanie operatora fizycznego składają się z trzy faz:

1. Na początku z kolekcji związanych z wejściami 1 i 2 usuwane są krotki, których znaczniki czasu t_e są mniejsze od $t.t_s$, ponieważ krotki te już nie będą uczestniczyły w tworzeniu przyszłych wyników.
2. Następnie przeglądana jest kolekcja zawierająca aktywne krotki z wejścia 2. Jeżeli zostanie spełniony warunek równości, wtedy krotki są łączone oraz wyliczany jest czas życia będący częścią wspólną krotek składowych. Jeżeli wynikowy czas życia jest większy od zera, krotka jest wstawiana do kolejki *queue*.
3. Na koniec krotka t jest dodawana do kolekcji 1 a wynik wymiatany *sweep(t)*.

Przykładowy przebieg czasowy działania tego algorytmu ilustruje rys. 2.3b). W tym modelu strumień zawiera wyłącznie krotki temporalne [56]. Zauważmy, że

w rzeczywistości nie zawsze znamy czas życia wydarzenia w chwili jego zaistnienia. Pojawia się pytanie, jaki czas życia powinna mieć krotka, która opisuje takie zjawisko. Jedno podejście polega na zaczekaniu do zakończenia wydarzenia i wygenerowania wtedy krotki z właściwym czasem życia. Rozwiązanie takie generuje niestety wysokie opóźnienia. Drugnie rozwiązanie dzieli czas obserwacji na kwanty. Jeżeli obserwowane zjawisko zachodzi przez ustalony kwant, wtedy generowana jest krotka. Podejście to minimalizuje opóźnienia, z drugiej strony pojedyncza obserwacja jest reprezentowana przez wiele krotek, co dodatkowo obciąża system.

Model oparty o krotki pozytywne i negatywne cechuje się tym, że krotka posiada pojedynczy znacznik czasu z tego powodu na rysunku reprezentuje ją punkt w czasie. Krotki negatywne są wyróżnione przez znak minusa. Aby zamodelować wydarzenie należy zasygnalizować jego początek krotkś pozytywną oraz jego koniec krotką negatywną. Załóżmy, że przetwarzana krotka *t* pochodzi ze strumienia 1.

- Jeżeli krotka *t* jest pozytywna, wtedy przeglądana jest kolekcja zawierająca krotki aktywne z wejścia 2. Jeżeli zostanie spełniony warunek równości na wyjście jest wysłana krotka wynikowa będąca konkatenacją atrybutów obu krotek składowych. Na koniec krotka *t* jest dodawana do kolekcji 1.
- Jeżeli krotka *t* jest negatywna, przeglądana jest kolekcja zawierająca krotki aktywne z wejścia 2. Jeżeli zostanie spełniony warunek równości, na wyjście wysyłana jest krotka negatywna będąca konkatenacją atrybutów obu krotek składowych. Na koniec z kolekcji 1 jest usuwana krotka *t*.

Przykładowy przebieg działania takiego operatora zilustrowano na rys. 2.3a). Słabością tego modelu jest potrzeba dwukrotnego wyszukiwania i złączania krotek podczas obserwacji pojedynczego wydarzenia odpowiednio dla krotki pozytywnej i negatywnej.

Operator fizyczny złączenia zdefiniowany dla strumieni typu mieszanego korzysta z kolejki *queue*, która porządkuje krotki wynikowe leksykograficznie. Kolejka udostępnia funkcję *sweep(t)* wymiatająca z kolejki krotki o znacznikach $[t_s,t_e)$ mniejszych od $[t.t_s,t.t_s+1)$. Poniżej zamieszczono algorytm operatora:

- Jeżeli krotka *t* jest temporalna, wtedy na początku z kolekcji 1 i 2 usuwane są krotki ze znacznikami czasu t_e mniejszymi od $t.t_s$, ponieważ krotki te już nie będą uczestniczyły w tworzeniu nowych wyników. Następnie realizowane jest

złączenie krotki t z kolekcją dla wejścia 2. Otrzymane krotki wynikowe są wstawiane do kolejki *queue*, po czym wywołana jest funkcja *sweep(t)*. Na wyjście wysyłane są tylko krotki których nowy czas życia jest większy od zera. Na koniec krotka t jest dodawana do kolekcji 1.

- Jeżeli krotka t jest negatywna, wtedy z kolekcji 1 i 2 usuwane są krotki których znaczniki czasu t_e są mniejsze od $t.t_s$. Nie ma konieczności dla nich generować krotek negatywnych, ponieważ uprzednio wygenerowane przez nie krotki wynikowe zawierają zdefiniowany czas życia. Aby zaktualizować czasy życia krotek, które zostały wcześniej wygenerowane przez krotkę t, realizowane jest złączenie krotki t z kolekcją dla wejścia 2. Nowe krotki negatywne mają znacznik $t_s = t.t_s$ oraz $t_e = t.t_e$. Otrzymane krotki wynikowe są wstawiane do kolejki *queue*, po czym wywołana jest funkcja *sweep(t)*. Na koniec, z kolekcji zasilanej wejściem 1 usuwana jest krotka t.
- Jeżeli na wejście dotrze krotka graniczna, wtedy z kolekcji 1 i 2 usuwane są krotki których znaczniki czasu t_e są mniejsze od $t.t_s$. Następnie jest ona wstawiana do kolejki *queue*, po czym wywołana jest funkcja *sweep(t)*.

Aby zaznaczyć graficznie fakt, że czas życia krotki w tym ujęciu może ulec zmianie, została ona oznaczona linią przerywaną. Na rys. 2.3c) przedstawiono przykładowy przebieg działania operatora złączenia. Gdy przyjrzymy się przykładom b) i c) zauważymy, że różnica pomiędzy modelem temporalnym a mieszanym polega na interpretacji strumieni. Model temporalny definiuje precyzyjnie czas życia, podczas gdy model mieszany definiuje odgórne ograniczenie czasu życia.

Podsumowując, temporalny typ strumieni upraszcza proces usuwania krotek z kolekcji lokalnych operatorów w odniesieniu do strumieni zawierających krotki pozytywne i negatywne, ponieważ każda krotka temporalna definiuje czas życia i nie ma potrzeby generowania krotek negatywnych. Warto zauważyć, że oczyszczanie kolekcji lokalnych operatora z elementów wygasłych polega na usuwaniu krotek w rosnącej kolejności znaczników t_e, co jest zadaniem o niskiej złożoności obliczeniowej. Drugą zaletą stosowania krotek temporalnych jest jednokrotne wywołanie opracji wyszukiwania i konkatenacji krotek, podczas gdy dla strumieni zawierających krotki pozytywne i negatywne pojedyncze wydarzenie jest reprezentowane przez dwie krotki, co prowadzi do dwukrotnego wywoływania tej operacji. Wada strumieni temporalnych ujawnia się, gdy czas życia wydarzenia jest

nie znany w chwili utworzenia krotki temporalnej. Zaproponowany typ mieszany strumieni łączy zalety poprzednich rozwiązań. Jeżeli czas życia krotki jest znany w chwili jej tworzenia, wtedy nie ma potrzeby korzystać z krotek negatywnych. Dzięki temu operacja łączenia jest wykonywana jednokrotnie. Jeżeli czas trwania danego wydarzenia jest nieznany w chwili jego rozpoczęcia, generowana jest krotka o nieskończonym czasie życia, następnie przy użyciu krotek negatywnych czas ten jest skracany. W ten sposób nie trzeba dzielić czasu życia wydarzenia na kilka przedziałów, tak jak ma to miejsce w strumieniu z krotkami temporalnymi [55].

Przystąpimy teraz do analizy poprawności implementacji operatora Θ złączenia. Kolejka *queue* jest regularnie wymiatana w takt czasu lokalnego operatora, zatem jej rozmiar jest ograniczony. Kolekcje 1 i 2 realizują funkcję tabel historii. Znajdują się w nich tylko aktywne krotki, ponieważ albo krotki wygasłe są wymiatane po nadejściu krotki t albo krotki temporalne zostają usunięte przez krotki negatywne. Przesłanki te wskazują, że zachodzi warunek przedstawiony w sekcji 2.6, zgodnie z którym implementacja operatora fizycznego spełnia niezmiennik stanu. Do wygenerowania krotki wynikowej konieczna i wystarczająca jest kolekcja 2 dla krotek ze strumienia 1 i kolekcja 1 dla krotek ze strumienia 2, podsumowując do wygenerowania krotki wynikowej potrzebne są tylko dane, które już napłynęły do systemu. Oznacza to, że operator spełnia niezmiennik przejścia. Zgodnie z definicją, niezmiennik propagacji w omawianej algebrze zachodzi, gdy krotki są uporządkowane zgodnie z porządkiem leksykograficznym oraz krotki graniczne są przekazywane z wejść na wyjście. Operator łączenia wygenerowane krotki wynikowe przekazuje do kolejki *queue* uporządkowanej leksykograficznie. Gdy operator przetworzy ostatnią krotkę z wartością znacznika t_s równą x, wtedy po nadejściu kolejnej krotki funkcja *sweep* wymiata wszystkie krotki ze znacznikami t_s mniejszymi lub równymi x. Należy zauważyć, że funkcja *sweep(t)* wymiata również krotki, których czas życia jest równy zero i znacznik t_s jest równy $t.t_s$. Warunek ten zapewnia, że krotki graniczne są przekazywane w tym samym cyklu na wyjście operatora z zachowaniem porządku leksykograficznego. Powyższy opis wskazuje, że operator spełnia niezmiennik propagacji.

W sekcji 2.5 zdefiniowano cztery typy monotoniczności, które uporządkowane ze względu na siłę ograniczeń tworzą szereg: typ monotoniczny, typ najsłabiej nie monotoniczny, typ słabo monotoniczny i typ mocno nie monotoniczny.

Na potrzebę dalszej analizy oznaczono te typy kolejno indeksami o wartościach od 1 do 4. Zauważmy, że jeżeli oba strumienie są monotoniczne, wtedy nie istnieje w tabeli historii krotka ze znaczkiem t_e innym niż nieskończoność, czyli wynik jest również monotoniczny. Kiedy jeden strumień jest monotoniczny a drugi najsłabiej nie monotoniczny, wtedy ciąg wartości znaczników t_e dla krotek wynikowych jest identyczny z ciągiem znaczników krotek ze strumienia najsłabiej nie monotonicznego. Oznacza to, że strumień wynikowy jest najsłabiej nie monotoniczny. Jeżeli oba strumienie są najsłabiej nie monotoniczne, wtedy szereg wartości znaczników t_e dla obu strumieni jest niemalejący. Przyjmijmy, że na wejście pierwsze napłyną trzy krotki ze znacznikami t_e tworzącymi ciąg rosnących wartości. Następnie na wejście drugie napływa krotka t_1 i jest połączona z wcześniejszymi trzema krotkami. Gdy kolejna krotka t_2 napłynie na wejście drugie i zostanie połączona z tymi samymi trzema krotkami oraz $t_1.t_s < t_2.t_s$, wtedy porządek krotek względem znacznika t_s będzie inny niż względem znacznika t_e. Podsumowując, strumień wynikowy będzie słabo monotoniczny, kiedy oba strumienie zasilające są słabo monotoniczne. Jeżeli w jednym ze strumieni zasilających pojawią się krotki negatywne, strumień wynikowy również zawiera krotki negatywne. Czyli jest on mocno nie monotoniczny. Powyższą analizę streszcza poniższa tabela.

Tabela 2.1. Monotoniczność wyniku dla Θ złączenia

Monotoniczność wyniku	**Warunek**
mocno nie monotoniczny	jeden ze strumieni zasilających jest mocno nie monotoniczny
najsłabiej nie monotoniczny	jeden strumień jest monotoniczny a drugi najsłabiej nie monotoniczny
słabo monotoniczny	strumienie zasilające są typu najsłabiej nie monotonicznego lub słabo monotonicznego
monotoniczny	strumienie zasilające są monotoniczne

2.8.4 Operator agregacji

W relacyjnych bazach danych wartości agregatów wyznaczane są dla grup danych. Grupę stanowią krotki, które posiadają te same wartości atrybutów grupujących G. Na początek zostanie zdefiniowany operator agregacji dla strumieniowej bazy danych przyjmując, że lista atrybutów G jest pusta, czyli

wszystkie krotki strumienia należą do tej samej grupy. Skorzystamy z podejścia przedstawionego w [54]. Stwórzmy dla tabeli historii H strukturę *ENDPT(H)* reprezentującą listę unikalnych znaczników czasu dla interwałów zdefiniowanych przez krotki należące do H, posortowaną w porządku rosnącym. Formalnie *ENDPT(H)* definiujemy jako: $ENDPT(H) = sort(\{e.t_s | e \in H\} \cup \{e.t_e | e \in H\})$. Dla wartości $v \in ENDPT(H)$ definiujemy *v.next* jako najmniejszą wartość $y \in ENDPT(H)$ gdzie $v < y$. Brak wartości *v.next* dla elementu o największej wartości w $ENDPT(H)$, dlatego definiujemy $ENDPT^{-}(H)$ który składa się z $ENDPT(H)$ za wyjątkiem elementu o największej wartości. Następnie dla każdego $v \in ENDPT^{-}(H)$ definiujemy zbiór F_v składający się z krotek należących do H i których czas życia nachodzi na interwał *[v, v.next]*, formalnie tą definicję zapisujemy: $F_v = \{e \in H | e.t_s \leq v \land e.t_e \geq v.next\}$. W oparciu o przedstawione komponenty operator agregacji definiujemy:

$$\alpha(H) = \bigcup_{v \epsilon ENDPT^{-}(H)} \bigcup_{l \epsilon P_v} (v, v.next, l) \text{ gdzie } P_v = \otimes(F_v) \qquad (2.14)$$

Przyjęto, że na zbiorze krotek F_v można zdefiniować kilka funkcji agregacji, wtedy wynikiem jest zbiór wartości P_V, który następnie jest zapisany do krotki wynikowej. Przykładowo, jeżeli w miejsce $\otimes$ podstawiamy zliczanie elementów, wtedy operator agregacji wyznaczy liczbę krotek należącą do kolejnych interwałów $v \in ENDPT^{-}(H)$. Jeżeli lista atrybutów G jest niepusta, wtedy należy podzielić tablicę historii na grupy i wyznaczyć agregat dla każdej z grup osobno zgodnie z przedstawioną powyżej definicją.

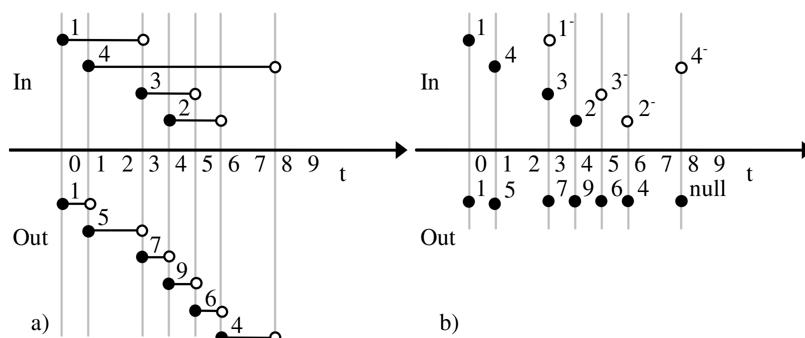
Aby przybliżyć zagadnienie wyznaczania agregatów w strumieniowej bazie danych, prześledzimy wyliczanie agregatu sumy dla różnych modeli strumienia wejściowego. Dla uproszczenia prezentacji rozważymy przypadek z pojedynczą grupą:

1. Funkcjonowanie operatora przetwarzającego krotki temporalne [56] ilustruje rys. 2.4 a). Po wejściu do sytemu nowej krotki t, jest ona zapisywana do struktury lokalnej H. Zgodnie z definicją strumienia, znaczniki t_s są uporządkowane niemalejąco. Czyli po nadejściu krotki t, krotki które pojawią się w przyszłości na wejściu, nie zmienią wartości agregatów których czas życia jest mniejszy od $t.t_s$. Przyjmijmy, że dysponujemy listą L uporządkowanych rosnąco znaczników t_s i t_e pochodzących z krotek przechowywanych w kolekcji

H. Działanie operatora polega na wyznaczaniu agregatu sumy dla każdego przedziału pomiędzy kolejnymi znacznikami mniejszymi lub równymi $t.t_S$ i wysyłanie go na wyjście. Na zakończenie z struktury *H* usuwane są krotki z t_e mniejszym lub równy $t.t_S$.

Podsumowując dla tego algorytmu wartość bieżąca agregatu jest wysyłana na wyjście w chwili, kiedy na wejściu operatora pojawi się kolejna krotka. W konsekwencji wyjście takiego operatora jest zawsze opóźnione. Aby temu przeciwdziałać autorzy tego rozwiązania zaproponowali transformowanie krotek o długim czasie życia na szereg krotek o krótkich interwałach. W wyniku tego skraca się czas opóźnienia kosztem większej liczby krotek w systemie.

Rozważmy dodatkowo przypadek, kiedy operator agregacji posiada zdefiniowaną listę atrybutów grupowania. Po zapisaniu *t* do struktury *H*, podobnie jak poprzednio wyliczane są agregaty, których czas życia jest mniejszy lub równy $t.t_S$. Tym razem na wyjście wysyłane są aktualizacje agregatów dla każdej grupy. Jeżeli w bieżącej chwili kolekcja *H* zawiera krotki należące do *n* grup, wtedy na wyjściu pojawi się co najmniej *n* krotek. Pomimo że zmienia się wartość agregatu tylko dla jednej grupy, na wyjście wysyłane są krotki dla każdej grupy, stanowi to wadę agregacji zbudowanej na strumieniu składającym się wyłącznie z krotek temporalnych.



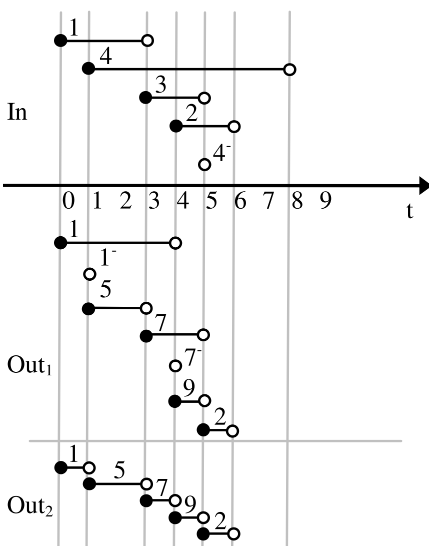
Rys. 2.4. Przebieg procesu agregacji dla modelu: a) temporalnego, b) z krotkami pozytywnymi i negatywnymi

2. Funkcjonowanie operatora agregacji przetwarzającego krotki pozytywne i negatywne[36] obrazuje rysunek 2.4 b). Operator ten po nadejściu nowej krotki na wejście wylicza wartość sumy i wysyła ją na wyjście. Wartość ta

dezaktualizuje poprzednią, przez co nie ma potrzeby wstawiania krotek negatywnych do strumienia wynikowego. W chwili, kiedy na wejściu pojawi się ostatnia krotka należąca do grupy, na wyjście jest wysyłana wartość *null*.

W odróżnieniu do poprzedniego rozwiązania, ten operator przekazuje na wyjście bieżącą wartość agregatu bez dodatkowego opóźnienia. W literaturze [36,15] spotykamy się z założeniem, że bieżąca wartość agregatu usuwa poprzedzającą, tak jak zilustrowano to na rys. 2.4 b). Dzięki takiemu rozwiązaniu zmniejsza się liczba krotek generowanych przez system, z drugiej strony prowadzi to do zmiany interpretacji strumienia. Każdy operator stanowy zasilany bezpośrednio lub pośrednio przez taki strumień musi automatycznie usuwać poprzednią wartość. Przyjmijmy, że istnieje zapytanie z strumieniem S będącym wyjściem operatora agregacji. Następnie zapytanie to jest zmodyfikowane tak, że ulega zmianie operator zasilający strumień S. Aby to osiągnąć należy również przebudować część zapytania, która jest zasilana strumieniem S. Nie można zatem zastosować inkapsulacji pod-rozwiązania, która chroni przed modyfikacją raz zbudowanego i przetestowanego fragmentu kodu. Powyższa niedoskonałość staje się dużym problemem, gdy operator agregacji posiada nie pustą listę atrybutów grupowania. Wtedy przetworzenie krotki t przez pod-zapytanie zasilane strumieniem S wymaga identyfikacji grupy G_t do której należy, po to aby poprawnie zidentyfikować krotkę przeznaczoną do usunięcia. Aby zrealizować ten algorytm, operatory należące do pod-zapytania muszą znać zastosowany algorytm grupowania. Przedstawioną powyżej wadę można usunąć, jeżeli każda nowa wartość agregatu jest poprzedzona krotką negatywną. Problem jednak pozostaje aktualny w systemach, w których nie istnieją krotki negatywne, należą do nich: STREAM [8,5], Aurora&Borealis [1] i Eddy[60].

Zaletą zastosowania strumieni z krotkami pozytywnymi i negatywnymi jest to, że po nadejściu krotki t na wyjście generowana jest tylko krotka aktualizująca agregat grupy, do której należy. W przypadku operatora agregacji omówionego w punkcie 1, generowane jest co najmniej tyle krotek, ile jest grup w strukturze H.



Rys. 2.5. Wyznaczanie agregatów dla modelu z krotkami temporalnymi i negatywnymi

Przykładowy przebieg czasowy operatora agregacji dla strumienia typu mieszanego ilustruje rys. 2.5 gdzie wejściem jest strumień *In* a wyjściem Out_1. Operator agregacji posiada lokalną kolekcję krotek *H*, których czas życia nie wygasł. Zgodnie z definicją strumienia, każda krotka ma ustalony czas życia (w przypadku skrajnym krotka ma nieskończony czas życia). W oparciu o niego definiujemy listę *LT*, która zawiera uporządkowane rosnąco znaczniki t_s i t_e dla zbioru *H*. Odstępy pomiędzy kolejnymi znacznikami w liście *LT* tworzą interwały dla których wartości agregatów nie ulegają zmianie. Gdy operator odczyta na wejściu krotkę *t* dodaje ją do kolekcji *H* oraz aktualizuje *LT*. Potem wysyła na wyjście krotkę negatywną, która dezaktualizuje poprzednią wartość agregatu. W kolejnym kroku podobnie jak dla punktu 1, na wyjście wymiatane są agregaty dla przedziałów, których czas życia nie przekracza progu $t.t_s$. Na koniec w przeciwieństwie do algorytmu z punktu 1 wysyłana jest bieżąca wartość agregatu. Cechą zaproponowanego rozwiązania jest jeden niezmienny algorytm interpretacji strumienia. Dzięki temu operator agregacji nie narzuca sposobu interpretacji strumienia wynikowego, tak jak miało to miejsce w przypadku operatora agregacji dla punktu 2.

Jak widzimy na rys. 2.5, po przybyciu krotki *([0,3), +, 1)* na wyjście jest generowana bieżąca wartość agregatu *([0,3), +, 1)*. Gdy przetworzona zostanie krotka *([1,8), +, 4)*, operator wpierw usuwa poprzednią wartość agregatu *([1,1), -, 1)*. Następnie generuje nową wartość *([1,3), +, 5)*.

Zastąpienie krotek pozytywnych temporalnymi oferuje także bogatszą funkcjonalność, ponieważ operator dostarcza informacji o ograniczeniach czasowych wyznaczanych agregatów. Parametr ten może być pomocny w systemach wspomagających podejmowanie decyzji, tam gdzie ważne jest ustalenie czy zmiany mają charakter krótkoterminowy. Przykładowo wyliczamy agregaty na wartościach indeksów giełdowych dla okna czasowego przesuwnego. Znając czas życia cząstkowych pomiarów potrafimy udzielić odpowiedzi jak długo bieżąca wartość nie ulegnie zmianie. Można również zdefiniować agregat, który będzie aproksymował wartość oczekiwaną korzystając z wiedzy o czasie trwania aktualnie zebranych pomiarów cząstkowych.

Podsumowując, operator agregacji dla zaproponowanego modelu wyróżnia się jeszcze dwiema cechami. Operator ten generuje mniej krotek, gdy lista atrybutów grupowania jest niepusta w porównaniu do operatora omówionym w punkcie 1. Operator agregacji dla strumieni temporalnych po przetworzeniu krotki na wejściu, wysyłał na wyjście co najmniej tyle krotek, ile w bieżącej chwili zawartych było grup w kolekcji H. Operator agregacji dla modelu mieszanego na wyjście generuje tylko krotki dla grup, w których nastąpiła zmiana wartości agregatu. Ponadto zastosowanie krotek negatywnych pozwala na wysyłanie na wyjście bieżących wartości agregatów, natychmiast po przetworzeniu krotki wprowadzającej zmiany. Podczas gdy dla algorytmu w punkcie 1, aktualizacja następuje dopiero po przetworzeniu kolejnej krotki z wejścia.

W jednowątkowym systemie tylko jeden operator działa w danym momencie czasu, a decyzję o jego uruchomieniu podejmuje scheduler. Uruchomiony operator przetwarza dane do momentu ich wyczerpania. Czym system jest mocniej obciążony, tym więcej krotek zalega w strumieniach. W wyniku tego więcej krotek jest przetworzonych po uruchomieniu operatora. Przyjmijmy, że operator agregacji analizowany na rys. 2.5 został uruchomiony, gdy w strumieniu zalegały krotki ze znacznikami obejmującymi przedział od 0 do 9. Dla wersji podstawowej na wyjście wysyłane są nowe wartości agregatów zaraz po przetworzeniu krotki z wejścia, ilustruje to wyjście Out_1. Rozważmy przypadek, w którym operator agregacji wstawia wynikowe krotki temporalne do bufora. Jeżeli wstawiana jest krotka negatywna,

wtedy wyszukiwana jest w buforze krotka temporalna o identycznym kluczu głównym. Gdy wyszukiwanie zakończy się pozytywnie, wtedy aktualizowany jest znacznik t_e krotki temporalnej zgodnie z wartością krotki negatywnej. Jeżeli w buforze brak odpowiadającej krotki temporalnej, wtedy krotka negatywna jest wstawiana do bufora. Gdy operator agregacji przetworzy wszystkie krotki w danym cyklu schedulera, zawartość bufora jest przekazywana na wyjście. Po zastosowaniu tego algorytmu strumień wynikowy zawiera mniej krotek negatywnych, co ilustruje strumień Out_2 na rys. 2.5. Dla rozpatrywanego przykładu redukcja krotek negatywnych zmniejsza rozmiar strumienia wynikowego o 28%. Zauważmy, że wraz ze wzrostem obciążenia systemu strumieniowego, liczba buforowanych krotek wzrasta, w konsekwencji więcej krotek zostanie zredukowanych. Dokładniejsza analiza wydajności tego rozwiązania zostanie omówiona w sekcji 2.9.

Pełną definicję operatora fizycznego agregacji przedstawia alg. 2.2. Do jego zdefiniowania użyto czterech struktur danych:

H –tabela mapująca wartości atrybutów grupujących G na stan grupy. Stan grupy to struktura zawierająca zbiór krotek o tych samych wartościach atrybutów grupujących ze strumienia wejściowego. Dodatkowo struktura ta przechowuje: bieżącą wartość agregatów w zmiennej v, minimalną wartość t_e dla krotek zbioru.

H_{PK} – tabela haszująca krotki wejściowe ze względu na klucz główny PK.

L_{Exp} – kolekcja krotek uporządkowana zgodnie z czasem ich wygasania.

DL – bufor krotek wynikowych uporządkowany leksykograficznie względem t_s, i t_e.

Przeprowadzona zostanie teraz analiza poprawności implementacji operatora agregacji. Zgodnie z def. 2.8 operator fizyczny implementuje operator logiczny, wtedy i tylko wtedy, gdy $o\llbracket \mathcal{S}\llbracket S\rrbracket\rrbracket = \mathcal{S}\llbracket o\llbracket S\rrbracket\rrbracket$.

Zauważmy, że struktura H jest tabelą historii grupującą krotki ze względu na atrybuty G. Funkcja *Process* w wierszach: 6 i 10 obsługuje wstawianie krotek temporalnych do H. Z kolei wiersze 14 i 15 realizują obsługę krotek negatywnych. Gwarancję tego, że w strukturze H znajdują się wyłącznie krotki aktywne zapewnia wywołanie funkcji *RemoveExpiredTuples* w wierszu 1.

Algorytm 2.2. Algorytm operatora agregacji.

```
Process(Tuple t)
1)RemoveExpiredTuples(t)
2)If t jest krotką temporalną
3)      If t nie występuje w H_PK
4)            return
5)      dodaj t to H_PK
6)      If H nie zawiera grupy G_t
7)            utwórz grupę G_t dla H
8)      Else
9)            dodaj krotkę negatywną do DL która usuwa wcześniejszy
                    agregat, aż do czasu t.t_s
10)     dodaj t do G_t
11)     wylicz v stosując Agg na krotkach należących do G_t
12)     dodaj krotkę temporalną ([G_t.min, G_t.nextMin), <G.g,G.v>) do DL
13)If t jest krotką negatywną i nie występuje w H_PK
14)     dodaj krotkę negatywną do DL która usuwa wcześniejszy
              agregat, aż do czasu t.t_s
15)     RemoveTuple(t)
16)If t jest krotką Boundary
17)     dodaj krotkę Boundary do DL

RemoveExpiredTuples(Tuple t)
1)While L_Exp nie jest pusty
2)      pobierz krotkę r z L_Exp
3)      If r.te > t.ts
4)            return
5)      RemoveTuple(r)

RemoveTuple(Tuple t)
1)pobierz G_t dla t z H
2)usuń t z L_EXP
3)usuń t z G_t
4)usuń t z H_PK
5)If G_t jest puste
6)      usuń G_t z H
7)else
8)      wylicz v stosując Agg na krotkach należących do G_t
9)      dodaj krotkę temporalną ([G_t.min, G_t.nextMin), <G.g,G.v>) do DL
```

Produkcja agregatów cząstkowych jest podzielona na dwa etapy. Wpierw budowane są agregaty w trakcie wymiatania krotek, których czas życia upłynął w strukturze *H*. W drugim etapie wyliczany jest agregat dla pobranej z wejścia krotki *t*. Pierwszy etap realizuje funkcja *RemoveExpiredTuples*, która identyfikuje krotki ze znacznikami t_e mniejszymi lub równymi $t.t_s$, a następnie wywołuje funkcję *RemoveTuple*. Funkcja ta korzystając z krotki negatywnej dezaktualizuje poprzednią wartość agregatu oraz ustawia aktualną wartość poprzez krotkę temporalną. Aby przyspieszyć proces identyfikacji krotek, których czas życia upłynął zdefiniowano strukturę L_{Exp} zawierającą krotki uporządkowane zgodnie z czasem ich wygasania. Jej zawartość jest aktualizowana w trakcie obsługi struktury *H*. W drugim etapie tworzona jest krotka reprezentująca bieżącą wartość agregatu. Wpierw na wyjście

wysyłana jest krotka negatywna finalizująca czas życia poprzedniego agregatu. Operację tą implementują wiersze 9 i 14. Następnie, jeżeli na wejściu pojawiła się krotka temporalna, bieżącą wartość agregatu jest aktualizowana w wierszu 12 funkcji *Process*. Jeżeli na wejściu pojawiła się krotka negatywna, bieżący agregat jest wyznaczany w wierszu 9 funkcji *RemoveTuple*. Dzięki temu, że krotki zawarte w strukturze H są przetwarzane zgodnie z porządkiem rosnącym znaczników t_e bufor DL zawiera uporządkowaną leksykograficznie listę wszystkich krotek wynikowych. Wyróżniamy dwie implementacje bufora DL. W wersji podstawowej przekazuje on krotki bezpośrednio na wyjście. W wersji rozszerzonej krotki w buforze są przechowywane przez okres jednego cyklu schedulera, w celu redukcji krotek negatywnych zgodnie z def. 2.2. Omówione elementy składowe algorytmu operatora agregacji wskazują, że implementacja operatora agregacji spełnia def. 2.8.

Poprawna implementacja operatora agregacji wymaga dodatkowo wykazania zachodzenia niezmienników. Omówiony algorytm spełnia niezmiennik przejścia, ponieważ do przetworzenia krotki t potrzebny jest tylko prefiks strumienia dla krotki t. Po przetworzeniu krotki t wszystkie agregaty z czasami życia mniejszymi od $t.t_s$ są wstawione do bufora, a ich wartości nie są modyfikowane przez napływające na wejście krotki. Na podstawie tej obserwacji krotki źródłowe ze znacznikiem t_e mniejszym od $t.t_s$. mogą zostać usunięte z struktury H. Jest to realizowane przez metodę *RemoveExpiredTuples*. Dzięki temu, operator agregacji przechowuje tylko ten fragment strumienia, który odpowiada aktywnym krotkom w tabeli historii. Własność ta wskazuje, że zachowany jest niezmiennik stanu. Niezmiennik propagacji jest spełniony, ponieważ krotki graniczne są przekazywane bezpośrednio na wyjście po uprzednim oczyszczeniu struktury H z krotek wygasłych. Z kolei poprawność obsługi krotek negatywnych została wykazana podczas omawiania zgodności operatora fizycznego z definicją operatora logicznego.

Operator agregacji generuje zawsze strumień mocno nie monotoniczny. Jest to wynikiem tego, że po nadejściu nowej krotki należy wysłać krotkę negatywną dezaktualizującą wartość poprzedniego agregatu.

2.8.5 **Eliminacja duplikatów**

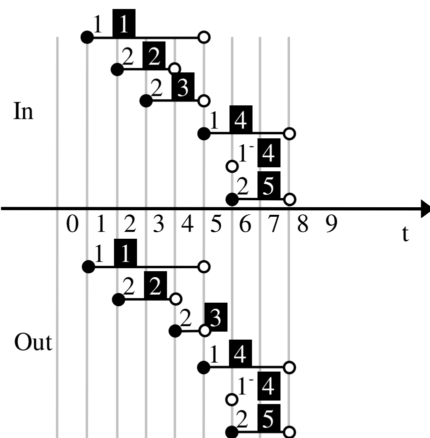
Operator logiczny definiujemy:

$$\delta_{DAttr}(H) = \{t|k \cdot t \in H\} \tag{2.15}$$

gdzie k jest liczbą całkowita większa od zera. Notacja $k \cdot t$ oznacza k wystąpień krotki t w tabeli historii. Dodatkowo krotki są do siebie podobne, jeżeli posiadają te same wartości atrybutów na liście *DAttr*.

Operator fizyczny eliminujący duplikaty porównuje krotkę temporalną na wejściu z krotkami wcześniej przetworzonymi. Jeżeli przybyła krotka temporalna jest unikalna ze względu na wartości atrybutów *DAttr*, wtedy jest ona przekazywana na wyjście. Jeżeli czas życia krotki temporalnej upłynie albo zostanie ona unieważniona przy użyciu krotki negatywnej, wtedy operator sprawdza czy w lokalnej kolekcji nie występuje inna krotka o identycznych wartościach *DAttr*, jeżeli tak to wysyła ją na wyjście.

Rysunek 2.6 ilustruje przebieg czasowy dla operatora zasilanego strumieniem mocno niemonotonicznym. Liczba na białym tle oznacza wartość *DAttr*, liczba na czarnym tle oznacza wartość klucza głównego *PK*. Przyjmijmy, że dla powyższego przykładu pojawiła się krotka 2^- dla *T=4*. Bez znajomości *PK* istnieje niejednoznaczność czy usunąć krotkę *[2, 4)* albo *[3, 5)*. Przypadek ten jest charakterystyczny dla strumieniowej bazy danych korzystającej z krotek temporalnych, ponieważ w zależności od wybranej krotki zmienia się czas życia wyniku. Jeżeli zostanie usunięta krotka o czasie życia *[2, 4)*, wtedy wynik ma czas życia dochodzący do *T=5*. Jeżeli wybrana zostałaby krotka o czasie *[3, 5)*, wtedy czas życia wyniku kończy się dla *T=4*. Jednoznaczne zarządzanie czasem życia krotek jest kluczowe w systemach strumieniowych, dlatego konieczne jest skorzystanie z klucza głównego *PK*. W przeciwieństwie do operatora obsługującego strumienie typu mieszanego, operator eliminujący duplikaty dla strumieni z krotkami pozytywnymi i negatywnymi [36] nie korzysta z *PK*. Jeżeli w danym momencie t istnieje kilka duplikatów o tej samej wartości np. *DAttr=2*, po nadejściu krotki negatywnej usuwana jest dowolna krotka z *DAttr=2*. Rozwiązanie takie jest dopuszczalne, ponieważ wszystkie krotki pozytywne reprezentują ten sam czas życia (nieskończony).



Rys. 2.6. Przetwarzanie strumienia wejściowego dla operatora eliminującego duplikaty

Algorytm 2.3 definiuje implementację operatora eliminacji duplikatów. Do jego budowy użyto poniższe struktury danych:

H_{PK} –tabela haszująca krotki wejściowe ze względu na *PK*.

H_{DST} –kolekcja przechowująca kubełki z krotkami wejściowymi o identycznych wartościach *DAttr*.

L_{EXP} –kolekcja krotek uporządkowana zgodnie z czasem ich wygasania.

DL –kolekcja krotek wynikowych uporządkowana leksykograficznie względem t_s, t_e.

Analiza poprawności operatora eliminacji duplikatów jest podobna do analizy poprawności operatora agregacji. Należy wykazać, że operator fizyczny dokonuje transformacji strumienia w tabelę historii, na której wykonywane są operacje zgodnie z definicją operatora logicznego. Następnie na wyjście wysyłane są krotki zgodnie z definicją strumienia. Podczas analizy implementacji operatora przyjmiemy, że w danym momencie na wejściu przetwarzana jest krotka *t*.

Odpowiednikiem tabeli historii dla omawianego operatora jest struktura H_{DST}. Kolekcja ta grupuje krotki ze względu na listę atrybutów *DAttr*. Obsługa struktury H_{DST} obejmuje wstawianie krotek temporalnych, co jest realizowane przez warunek w wierszu 6 funkcji *Process*. Z kolei obsługa krotek negatywnych jest zaszyta w wierszach 16-22 funkcji *Process*. W strukturze H_{DST} znajdują się wyłącznie krotki aktywne, co jest zagwarantowane przez funkcję *RemoveExpiredTuples* wywoływaną na początku obsługi dowolnej krotki na wejściu. Powyższe komponenty dowodzą, że struktura H_{DST} implementuje tabelę historii.

Algorytm 2.3. Algorytm operatora eliminacji duplikatów.

```
Process(Tuple t)
1)RemoveExpiredTuples(t)
2)If t jest krotką temporalną
3)      If t nie występuje w H_PK
4)            dodaj t do L_Exp
5)            dodaj t do H_PK
6)            If H_DST nie zawiera kubełka dla t
7)                  utwórz kubełek B i dodaj go do H_DST
8)                  dodaj t do kubełka B
9)                  dodaj t do DL
10)           Else
11)                 dodaj t do kubełka
12)If t jest krotką negatywną
13)     If t nie występuje w H_PK
14)           usuń t z L_Exp
15)           usuń t z H_PK
16)           pobierz kubełek B zawierający krotkę t
17)           usuń t z B
18)           dodaj negatywną krotkę t do DL
19)           If kubełek B zawiera inne krotki
20)                 dodaj następną krotkę z B do DL
21)           Else
22)                 odłącz B od H_DST
23)If t jest krotką Boundary
24)     dodaj krotkę Boundary do DL

RemoveExpiredTuples(Tuple t)
1)pobierz następną krotkę t_Rxp do wycofania z L_Exp
2)While t_Exp <> null i t_Exp poprzedza lub współistnieje wraz z t
3)      pobierz kubełek B zawierający krotkę t_Exp
4)      usuń t z L_Exp
5)      usuń t z H_PK
6)      usuń t_Exp z B
7)      /*w tym przypadku krotki negatywne nie są generowane*/
8)      If kubełek B zawiera inne krotki
9)            dodaj następną w porządku leksykograficznym krotkę z B do DL
10)     Else
11)           odłącz B z H_DST
12)     pobierz następną krotkę t_Exp do wycofania z L_Exp
```

Przed przystąpieniem do przetworzenia krotki t z kolekcji H_{DST} usuwane są krotki z znacznikami t_e mniejszymi lub równymi $t.t_s$. Obsługa usuwania krotek jest podobna do obsługi krotki negatywnej. Różnica tkwi w tym, że nie jest wstawiana do *DL* krotka negatywna, ponieważ nie zmienił się czas życia żadnej z uprzednio wygenerowanych krotek wynikowych. Przetworzenie krotki t składa się z trzech wariantów. Jeżeli krotka t jest temporalna i w kolekcji H_{DST} nie istnieje grupa przez nią reprezentowana, krotka t jest wstawiona do bufora *DL*. Jeżeli krotka t jest negatywna a w H_{DST} istnieje jej odpowiednik temporalny, wtedy jest ona wstawiona do *DL* a jej odpowiednik temporalny usunięty. Dodatkowo, jeżeli istnieje co najmniej jedna krotka o tej samej wartości atrybutów *DAttr*, wtedy do *DL* jest wstawiona

pierwsza krotka należąca do grupy zgodnie z porządkiem leksykograficznym. Jeżeli krotka *t* jest graniczną, algorytm wstawia ją bezpośrednio do *DL*.

Zdefiniowane trzy źródła krotek wynikowych tworzą poprawny strumień wynikowy, jeżeli jego elementy są uporządkowane leksykograficznie. Warunek ten jest spełniony, ponieważ przed przetworzeniem krotki *t* z kolekcji H_{DST} wymiatane są krotki wygasłe zgodnie z porządkiem leksykograficznym, przez co bufor *DL* zawiera uporządkowany zbiór krotek wynikowych.

Warto zasygnalizować, że jeżeli operator nie jest zasilany strumieniem mocno niemonotonicznym obsługa struktury K_{PK} jest zbyteczna. Tworząc wersję operatora bez tej struktury możemy poprawić wydajność czasową oraz zmniejszyć zapotrzebowanie na zasoby.

Przejdziemy teraz do weryfikacji czy spełnione są niezmienniki. Niezmiennik przejścia jest zachowany, ponieważ omawiany algorytm wymaga wyłącznie znajomości prefiksu strumienia dla krotki *t*. Zauważmy, że metoda *RemoveExpiredTuples* usuwa krotki, których znaczniki t_e są mniejsze lub równe $t.t_s$. Oznacza to, że struktura H_{DST} zawiera tylko aktywne krotki w tabeli historii, czyli stan operatora jest ograniczony a w konwencji niezmiennik stanu spełniony. Niezmiennik propagacji zachodzi, ponieważ algorytm implementuje generowanie krotek negatywnych i granicznych zgodnie z definicją operatora logicznego oraz do ich wygenerowania wymagana jest tylko znajomość krotek zapisanych w H_{DST}.

Przystąpimy teraz do analizy monotoniczności operatora. Krotki negatywne na wejściu prowadzą do wygenerowania krotek negatywnych na wyjściu, w konsekwencji strumień wynikowy jest mocno nie monotoniczny, gdy strumień wejściowy jest typu mocno nie monotonicznego. Zauważmy, że operator nie zachowuje porządku, w którym krotki rozpoczynają swój czas życia. Jeżeli w strumieniu występują duplikaty to po wygaśnięciu krotki wynikowej na wyjście jest wysyłana kolejna krotka należąca do tej samej grupy z zmienionym znacznikiem t_s. W konsekwencji kolejność krotek wynikowych nie odpowiada kolejności krotek na wejściu. Z drugiej strony operator nie zmienia czasów końca życia krotek. Powyższe elementy wskazują, że dla strumieni wejściowych o typach słabo monotoniczny i najsłabiej nie monotoniczny wyjście operatora jest słabo monotoniczne.

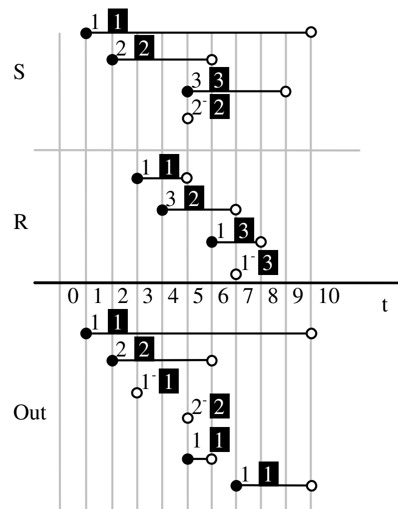
2.8.6 Różnica

Operator logiczny definiujemy:

$$S -_{Attr} R = \{(k-j) \cdot t | k \cdot t \in S \wedge j \cdot t \in R \wedge k > j\} \quad (2.16)$$

gdzie k i j są liczbami całkowitymi większymi od zera. Notacja $k \cdot t$ oznacza, że w tabeli historii dla strumienia S istnieje k wystąpień krotki t. Dodatkowo krotki są do siebie podobne, jeżeli posiadają te same wartości atrybutów $Attr$. Operator ten umieszcza w strumieniu wynikowym te krotki, które pojawiły się na wejściu S a które nie występują w strumieniu R. Definicja tego operatora uwzględnia krotność krotek. Jeżeli na wejściu S pojawi się 5 krotek, a na wejście R dotrą 3 krotki o tej samej wartości atrybutów $Attr$. Na wyjście zostaną przekazane 2 pierwsze krotki ze strumienia S zgodnie z porządkiem leksykograficznym.

Przykładowy przebieg czasowy operatora dla strumieni typu mocno nie monotonicznego przedstawia rys. 2.7.



Rys. 2.7. Przetwarzanie strumienia wejściowego dla operatora różnicy

Operator fizyczny minus jest zdefiniowany przez algorytmy 2.4 i 2.5, które korzystają dodatkowo ze struktur danych:

H_{Att} –tabela haszująca składająca się z kubełków B_{Att} z krotkami o identycznych wartościach $Attr$. Kubełek składa się z list $B.S$ i $B.R$ przechowujących krotki odpowiednio z wejść S i R. Dla obu list zdefiniowano metodę *size()* zliczającą liczbę elementów.

H_{SPK}, H_{RPK} –tabele mapujące klucze główne krotek ze strumieni S i R na referencje do struktur w tabeli H_{ATT}.

L_{Exp} –kolekcja zawierająca krotki zgodnie z porządkiem ich wygasania dla strumieni S i R.

OUT –kolekcja zawierająca krotki wynikowe uporządkowane leksykograficznie względem znaczników t_s i t_e.

Analiza poprawności operatora fizycznego różnicy podobnie jak do poprzednich operatorów składa się z trzech etapów. Wpierw wykazane zostanie, że operator fizyczny dokonuje transformacji strumienia w tabelę historii dla strumieni S i R. W oparciu o tą strukturę danych wyliczany jest wynik zgodnie z definicją operatora logicznego. Następnie na wyjście generowane są krotki zgodnie z definicja strumienia.

Odpowiednikiem tabeli historii dla omawianego operatora jest struktura H_{ATT}, składa się ona z kubełków, które zawierają krotki o tych samych wartościach atrybutów *Attr*. Aby przyspieszyć działanie operatora, kubełek posiada oddzielną listę dla krotek pochodzących ze źródła S i R. Implementacja konwersji ze strumienia do tabeli historii jest podzielona na cztery części. Wstawianie i usuwanie krotek pochodzących ze strumienia S jest realizowane odpowiednio w wierszach: 25-27 i 4-7, podczas gdy wstawianie i usuwanie dla strumienia R jest realizowane w wierszach: 34-38 i 14-17.

Generowanie wyniku po odczytaniu z wejścia krotki t jest podzielone na dwie fazy. Wpierw ze struktury H_{Attr} są wymiatane krotki ze znacznikami t_e mniejszymi lub równymi $t.t_s$, odpowiedzialna za to jest metoda *RemoveExpiredTuples*. Krotki są usuwane zgodnie z porządkiem znaczników t_e, co gwarantuje że wyniki są uporządkowane w kolejności leksykograficznej. Gdy krotka t_{exp} należy do strumienia R wtedy na wyjście wysyłana jest kolejna krotka ze strumienia S jeżeli *B.S.size() > B.R.size()*. Druga faza związana jest z obsługą krotki t. Kubełek B_t reprezentuje strukturę danych zawierającą krotki o tych samych wartościach atrybutów *Attr* co krotka t.

Algorytm 2.4. Algorytm operatora różnicy.

```
Process(Tuple t)
1)RemoveExpiredTuples(t)
2)If t jest krotką negatywną z strumienia S
3)      If t występuje w H_SPK
4)            pobierz kubełek B zawierający t
5)            usuń t z kolekcji S kubełka B
6)            usuń t z L_Exp
7)            usuń t z H_SPK
8)            If B.S.size() >= B.R.size()
9)                  pobierz krotkę t_next z B.S która: została
                          wytransmitowana na wyjście, jest najstarsza
                          w porządku leksykograficznym i nie została
                          jeszcze wycofana; stwórz negatywną krotkę
                          dla t_next i przekaż na OUT
10)           If B.S.size() == 0 and B.R.size() == 0
11)                 usuń kubełek B z H_ATT
12)Else if t jest krotką negatywną z strumienia R
13)     If t występuje w H_RPK
14)           pobierz kubełek B zawierający t
15)           usuń t z kolekcji R kubełka B
16)           usuń t z L_Exp
17)           usuń t z H_RPK
18)           If B.S.size() > B.R.size()
19)                 pobierz krotkę t_next z B.S która: nie została
                          wytransmitowana na wyjście i nie wygasła
20)                 dodaj krotkę t_next do OUT
21)           If B.S.size() == 0 and B.R.size() == 0
22)                 usuń kubełek B z H_ATT
23)Else if t jest krotką temporalną w strumieniu S
24)     If t nie występuje w H_SPK
25)           dodaj t do L_Exp
26)           dodaj t do H_SPK
27)           If H_ATT nie zawiera kubełka B dla t
28)                 stwórz kubełek B i dodaj go do H_ATT
29)           dodaj t do kolekcji S kubełka B
30)           if B.S.size() > B.R.size()
31)                 pobierz pierwszą krotkę t z B.S która: nie została
                    wytransmitowana na wyjście i nie wygasła; dodaj ją
                    na OUT

32)Else if t jest krotką temporalną w strumieniu R
33)     If t nie występuje w H_RPK
34)           dodaj t do L_Exp
35)           dodaj t do H_RPK
36)           If H_ATT nie zawiera kubełka B dla t
37)                 stwórz kubełek B i dodaj go do H_ATT
38)           dodaj t do kolekcji R kubełka B
39)           if B.S.size() >= B.R.size()
40)                 pobierz pierwszą krotkę t z B.S która: została
                          wytransmitowana na wyjście i nie wygasła
41)                 dodaj krotkę t na OUT
42)Else if t jest krotką boundary
43)     dodaj krotkę t na OUT
```

Zapisane tam krotki są uporządkowane w kolejności leksykograficznej i posiadają pole bitowe identyfikujące czy krotka została wysłana na wyjście. Po aktualizacji kubełka do którego należy krotka, sprawdzane jest czy należy wysłać na wyjście kolejną krotkę należącą do kubełka B_t lub usunąć wcześniej wysłaną wartość. Jeżeli

po przetworzeniu krotki t należy wysłać na wyjście krotkę z kubełka B_t, wybierana jest pierwsza krotka z listy S zgodnie z porządkiem leksykograficznym i która nie została do tej pory przekazana na wyjście. Jeżeli po przetworzeniu krotki t należy usunąć krotkę z kubełka B_t, wybrana zostaje krotka która wcześniej została wysłana na wyjściu i o najmniejszej wartości znaczników czasu zgodnie z porządkiem leksykograficznym. Powyższy algorytm jest konieczny, aby krotki wynikowe tworzyły ciąg uporządkowany leksykograficznie, co jest warunkiem wymaganym przez definicję strumienia. Warto zauważyć, że jeżeli strumień zasilający nie jest typu mocno nie monotonicznego, wtedy można wyłączyć z algorytmu obsługę kolekcji H_{SPK} i H_{RPK} przyspieszając tym pracę operatora.

Algorytm 2.5. Algorytm wymiatania wygasłych krotek dla operatora różnicy.

```
RemoveExpiredTuples(Tuple t)
1)Pobierz następną krotkę t_exp do wycofania z L_Exp
2)While t_exp <> null and t_exp.t_e < t.t_s
3)      Pobierz kubełek B zawierający krotkę t_exp
4)      Usuń t_exp z L_Exp
5)      If B.S zawiera t_exp
6)            Usuń t_exp z kolekcji B.S
7)      Else
8)            Usuń t_exp z kolekcji R.B
9)            If B.S.size() > B.R.size()
10)                 Pobierz aktywną krotkę t_next z B.S która nie została
                          wysłana na wyjście.
11)                 Dodaj krotkę t_next do OUT
12)           If B.S.size() == 0 and B.R.size() == 0
13)                 Usuń kubełek B z H_ATT
14)     Pobierz następną krotkę t_exp do wycofania z L_Exp
```

Druga faza algorytmu jest związana z obsługą krotki t. W trakcie działania algorytmu kubełek B_t zawiera listę krotek o identycznych wartościach atrybutów *Attr*. Zapisane tam krotki są uporządkowane w kolejności leksykograficznej i posiadają pole bitowe identyfikujące czy krotka została wysłana na wyjście. Po aktualizacji kubełka do którego należy krotka, sprawdzane jest czy należy wysłać na wyjście kolejną krotkę z kubełka B_t lub usunąć wcześniej wysłaną wartość. Jeżeli po przetworzeniu krotki t należy wysłać na wyjście krotkę z kubełka B_t, wybierana jest pierwsza krotka z listy S zgodnie z porządkiem leksykograficznym i która nie została do tej pory przekazana na wyjście. Jeżeli po przetworzeniu krotki t należy usunąć krotkę z kubełka B_t, wybrana zostaje najwcześniej wysłana na wyjście

i o najmniejszej wartości znaczników czasu zgodnie z porządkiem leksykograficznym. Powyższy algorytm jest konieczny, aby krotki wynikowe tworzyły ciąg uporządkowany leksykograficznie. Operacja zapisu wyniku do strumienia jest osiągana dzięki zastosowaniu listy L_{Exp}, która gwarantuje uporządkowanie leksykograficzne strumienia wynikowego. Warto zauważyć, że jeżeli strumień zasilający nie jest typu mocno niemonotonicznego, wtedy nie zawiera krotek negatywnych, co pozwala wyłączyć z algorytmu obsługę kolekcji H_{SPK} i H_{RPK} przyspieszając tym pracę operatora.

Analiza poprawności niezmienników dla operatora różnicy jest analogiczna jak dla poprzednich operatorów stanowych. Niezmiennik przejścia jest zachowany, ponieważ zgodnie z algorytmem po przetworzeniu krotki t operator fizyczny wyznacza wynik różnicy tylko w oparciu o zgromadzoną wcześniej historię strumieni S i R. Niezmiennik stanu jest spełniony, ponieważ operator przechowuje w strukturach danych wyłącznie aktywne krotki dla tabel historii zbudowanych na strumieniach S i R. Analiza algorytmu wskazuje na poprawną obsługę krotek negatywnych. Krotki graniczne są przekazywane z wejścia na wyjście po uprzednim wymieceniu z operatora krotek wygasłych, co zrealizowano w wierszach 1 i 43 funkcji *Process*. Dodatkowo przeprowadzona analiza algorytmu wskazuje na poprawną obsługę krotek negatywnych. Powyższe składniki dowodzą, że zachodzi niezmiennik propagacji.

Na zakończenie omówiona zostanie monotoniczność operatora. Zauważmy, że jeżeli na wejście R napłynie krotka, która wystąpiła wcześniej w strumieniu S, wtedy jest generowana na wyjściu krotka negatywna. A zatem nawet wtedy, gdy operator jest zasilany krotkami temporalnymi strumień wynikowy będzie zawierał krotki negatywne. Oznacza to, że wynikiem operatora minus jest zawsze strumień mocno niemonotoniczny.

2.8.7 Unia bez usuwania duplikatów

Definicja operatora logicznego:

$$H_1 \cup H_2 = \{(k+j) \cdot t | k \cdot t \in H_1 \vee j \cdot t \in H_2\} \quad (2.17)$$

gdzie k i j jest liczbą całkowitą większa od zera. Operator ten łączy dwa strumienie wejściowe S_1 i S_2 w pojedynczy strumień wynikowy w taki sposób, że strumień wynikowy jest uporządkowany ze względu na znaczniki t_s i t_e. Dodatkowo oba strumienie muszą posiadać identyczne schematy.

Implementacja operatora fizycznego polega na odczycie krotek wejściowych zgodnie z porządkiem leksykograficznym i przekazywaniem ich bezpośrednio na wyjście.

Operator ten jest bezstanowy, dzięki temu zachowany jest niezmiennik stanu. Krotki temporalne, negatywne i graniczne są przekazywane bezpośrednio na wyjście z zachowaniem porządku leksykograficznego, co skutkuje spełnieniem niezmiennika przejścia i propagacji.

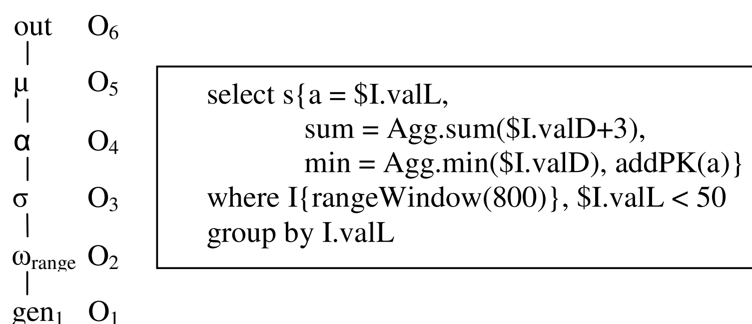
Analiza monotoniczności składa się z trzech przypadków. Operator przekazuje na wyjście sumę strumieni wejściowych, oznacza to, że jeżeli jeden ze strumieni zasilających jest mocno nie monotoniczny, wtedy wynik jest również mocno nie monotoniczny. Operator ten nie modyfikuje znaczników t_e, a zatem uporządkowanie krotek względem znacznika t_s dla słabo monotonicznych strumieni zasilających nie gwarantuje, że porządek względem t_s oraz t_e jest identyczny. Czyli jeżeli monotoniczność strumieni zasilających jest inna niż mocno nie monotoniczna, wtedy wynik jest słabo monotoniczny. Ostatni przypadek zakłada dwa strumienie monotoniczne, wtedy strumień wynikowy jest również monotoniczny.

2.9 Test kompresji strumieni dla agregacji

Przedstawiona analiza operatorów agregacji w punkcie 2.8.4 pokazuje, że ich realizacja znacząco różni się w zależności od przyjętego typu strumienia zasilającego. Jedną z zalet strumienia typu mieszanego jest możliwość jego kompresji poprzez redukcję krotek negatywnych. Porównanie liczności krotek w strumieniach Out_1 i Out_2 na rys. 2.5 pokazuje, że kompresja może dochodzić do 28%, co stanowi znaczącą poprawę. Dlatego w tym punkcie wydajność kompresji zostanie zbadana doświadczalnie.

Przeprowadzone testy mają na celu zweryfikować jak zmienia się wydajność kompresji w zależności od obciążenia systemu. Jako środowisko testowe posłuży

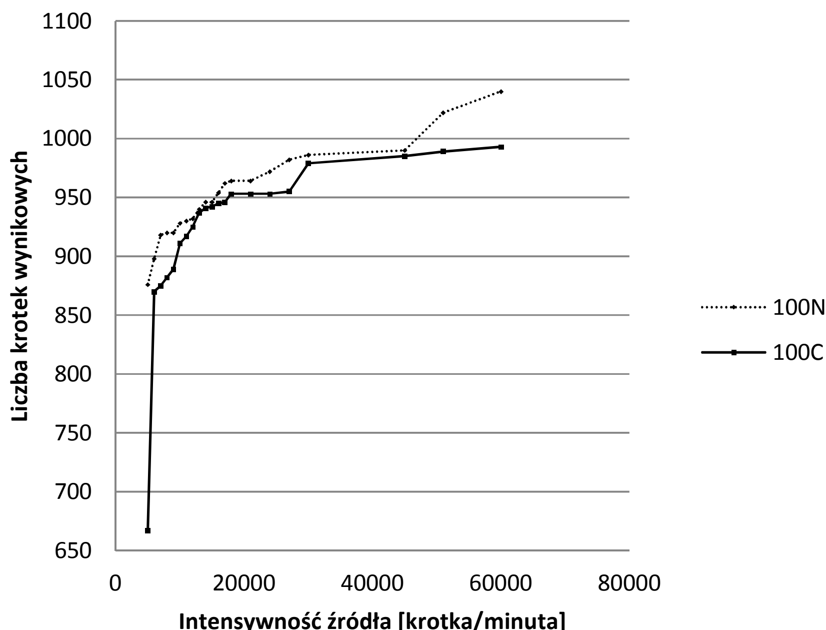
zapytanie zilustrowane na rys. 2.8. Składa się ono z ułożonych w szereg operatorów: okna liczebnościowego, operatora selekcji, operatora agregacji, operatora mapowania i operatora ujścia. Pełna definicja zapytania została opisana przy użyciu języka StreamAPAS i umieszczona po prawej stronie grafu zapytania. Aby sprawdzić algorytm kompresji w warunkach zbliżonych do rzeczywistych, dołączono grupowanie agregatów. Skutkuje to zwiększeniem odległości pomiędzy odpowiadającym sobie krotką pozytywnym i negatywnym, przez co obsługa kompresji zajmuje więcej czasu. Dodatkowo w zapytaniu umieszczono okno liczebnościowe obejmujące 800 elementów, aby realizacja operatora agregacji wymagała zastosowanie pełnego algorytmu przedstawionego w sekcji 2.8.4. Do eksperymentów użyto źródło generujące krotki ze stałym obciążeniem wyrażonym w krotkach na minutę. Schemat strumienia składa się z numeru grupy reprezentowanego przez atrybut *valL* i atrybutu *valD* zawierającego agregowaną wartość. Ponadto, numer grupy *valL* należy do przedziału[0,100) i był generowany przy użyciu rozkładu równomiernego. Pojedynczy eksperyment polegał na przetworzeniu 1000 krotek dla konfiguracji z załączoną i włączoną kompresją. Eksperyment ten powtórzono dla różnych intensywności strumienia wejściowego rozpoczynając od 5000 krotek na minutę kończąc na 60000.



Rys. 2.8. Zapytanie testujące kompresje strumienia agregacji

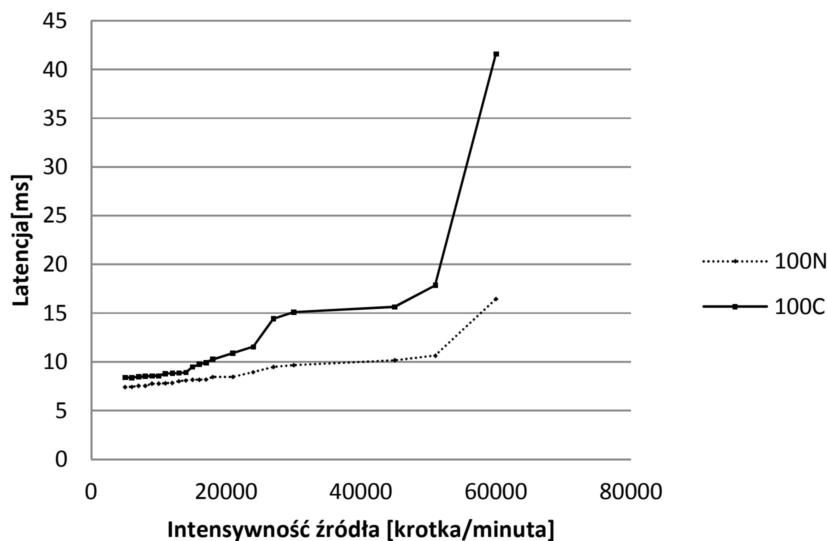
Wykres nazwany *100N* oznacza konfigurację z wyłączoną kompresją, a *100C* z załączoną. Rysunek 2.9 pokazuje, że wydajność kompresji jest znacznie niższa niż uzyskany wynik dla przykładu motywującego omówionego w punkcie 2.8.4. Dla przypomnienia w buforze kompresowane są krotki wynikowe wygenerowane w trakcie pojedynczego cyklu. Uruchomienie operatora agregacji rozpoczyna cykl. Następnie operator przetwarza dane do momentu opróżnienia strumienia wejściowego. W kolejnym kroku realizowana jest kompresja strumienia wynikowego,

po czym sterowanie jest oddane schedulerowi. Skuteczność kompresji poprawia się wraz z wzrostem rozmiaru bufora, gdy system jest mocniej obciążony. Można to zaobserwować na wykresie, gdzie wraz ze wzrostem intensywności źródła liczba skompresowanych krotek wzrasta. Rozmiar tego bufora nie jest jednak na tyle duży, aby efektywność kompresji równoważyła koszt związany z jej przeprowadzeniem. Wykres na rys. 2.10 jednoznacznie wskazuje, że czas odpowiedzi jest większy, gdy kompresja jest załączona.



Rys. 2.9. Wydajność kompresji

Przeprowadzone testy wskazują, że realna wydajność kompresji jest niska. Zbudowane środowisko testowe wyróżnia się tym, że operator jest zasilany strumieniem mocno niemonotonicznym, a zatem strumień wynikowy zawiera wiele krotek negatywnych. Własność ta powinna skutkować wysoką wydajnością kompresji, a tak nie jest. Przyczyna tego tkwi w rozmiarze bufora. System działa płynnie, przez co rozmiar bufora na cykl jest na tyle mały, że liczba redukowanych krotek negatywnych jest niska.



Rys. 2.10. Wpływ kompresji na opóźnienia

Podobny mechanizm kompresji danych został zaproponowany w systemach PIPES i CEDR. W systemie PIPES fakt, że strumień zawiera wyłącznie krotki temporalne zmusza do dekompozycji pojedynczej krotki temporalnej na szereg krotek o krótszym czasie życia w celu zmniejszenia czasów opóźnień. To zwielokrotnienie danych w strumieniu zmusza do stosowania operatorów kompresji w celu zminimalizowania zapotrzebowania pamięciowego oraz opóźnień. Przeprowadzone badania pokazują, że kompresja wiąże się z znaczącym wzrostem opóźnień. Z tej perspektywy zaproponowane rozwiązanie autorskie strumieni typu mieszanego jest atrakcyjne, ponieważ usuwa potrzebę dekompozycji krotek temporalnych a zatem konieczności ich późniejszej kompresji.

System CEDR wyróżnia się rozszerzeniem funkcjonalności o generowanie wyników tymczasowych, które są korygowane w chwili, kiedy do systemu docierają brakujące dane. Zaproponowana kompresja polega na zmniejszeniu liczby krotek korygujących poprzednie wartości. Zatem również w tym przypadku istnieje mechanizm zwielokratniający liczbę krotek w strumieniu, który uzasadnia późniejsze korzystanie z kompresji. Powyższe obserwacje wskazują, że w systemach konkurencyjnych kompresja jest realizowana po to, aby zminimalizować uboczne skutki dekompozycji krotek temporalnych lub korzystania z wyników tymczasowych.

2.10 Optymalizacja zapytania w oparciu o monotoniczność strumieni

W relacyjnych bazach danych zbiór danych, na których są realizowane operacje jest znany. W strumieniowych bazach danych nie znamy całego zbioru danych w chwili uruchomienia zapytania. Sprawia to, że statystyki opisujące zapytani są obarczone większym błędem. W przeciwieństwie do statystyk, niezmienniki gwarantują występowanie pewnych własności dla wszystkich krotek strumienia. Wyróżniamy dwa sposoby użycia monotoniczności w optymalizacji.

Pierwsza podejście jest skoncentrowane na pojedynczych operatorach. Znajomość typu monotoniczności pozwala przeprowadzić optymalizację operatorów stanowych, która jest niezależna od dynamicznie zmieniających się parametrów takich jak intensywność i selektywność. Przedmiotem optymalizacji jest tutaj algorytm przekształcający strumień w tabelę historii. Jeżeli strumień jest najsłabiej nie monotoniczny, wówczas kolejność krotek względem początku czasu życia odpowiada kolejności ich wygasania. Należy wtedy przechowywać krotki w kolejce FIFO, aby przyspieszyć proces wymiatania z tabeli historii krotki wygasłe. Implementacja wymiatania sprowadza się do przeglądania elementów listy, zgodnie z porządkiem zapisu. Jeżeli strumień zasilający jest słabo monotoniczny, wtedy o strumieniu wiemy tylko, że nie zawiera krotek negatywnych. Wydajna realizacja tabeli historii wymaga wówczas zbudowania kolejki priorytetowej zawierającej krotki uporządkowane względem czasu wygasania. Dzięki niej sprawdzane są tylko kolejne elementy kolejki do momentu napotkania krotki ze znacznikiem t_e większym od zadanego. Jeżeli strumień zasilający jest mocno nie monotoniczny, wtedy w strumieniu dodatkowo mogą pojawiać się krotki negatywne. Do ich obsługi konieczne jest utworzenie struktury indeksującej ze względu na klucz główny. Należy dodatkowo wprowadzić optymalizację zaproponowaną dla strumieni typu słabo nie monotonicznych, ponieważ strumień zawiera krotki temporalne oraz kolejności ich uporządkowania względem znaczników t_s i t_e są różne,. Konkludując, czym strumień jest słabiej monotoniczny tym bardziej złożona jest jego obsługa.

Drugi obszar zastosowania monotoniczności polega na przekształceniu zapytania strumieniowego tak, aby zawierało najwięcej operatorów przetwarzających strumienie o najmniejszym stopniu nie monotoniczności.

Zbudowany system jest oparty na operatorach logicznych, dlatego potrafimy zdefiniować przekształcenia tożsamościowe. Dodatkowo przy opisie każdego z operatorów, przeanalizowano jego typ monotoniczności, jako funkcja typów monotoniczności strumieni zasilających. Powyższe dwa elementy pozwalają na skonstruowanie prostego optymalizatora regułowego, który poprzez przestawienie operatorów w planie produkcji redukuje liczbę strumieni o wysokim stopniu nie monotoniczności.

Skonstruowany optymalizator pobiera wstępny plan produkcji zapytania opisany przy użyciu DAG, a jako wynik zwraca jego poprawioną wersję. Działanie optymalizatora opisuje alg. 2.6. Jego parametrami wejściowymi są zapytanie *dag* oraz lista reguł *reduceRules*. Reguła optymalizatora jest obiektem składającym się z dwóch metod. Pierwszą jest `check(op, dag)`, która sprawdza czy dla badanego operatora *op* zachodzi aktywacja reguły. Drugą jest `reduce(op, dag, ops)`, która modyfikuje plan produkcji. Jej argumentami są analizowany operator *op*, zapytanie *dag* i lista operatorów *ops* pozostałych do przetworzenia. Cykl optymalizatora polega na pobraniu z listy *reduceRules* kolejnej reguły. Następnie, jest ona wywoływana dla każdego operatora należącego do planu *dag*, do momentu gdy wszystkie operatory ją aktywujące zostaną przekształcone. Na koniec cyklu z listy *reduceRules* pobierana jest kolejna reguła.

Do skonstruowania listy reguł skorzystano z następujących własności:

1) Kolejność realizacji operatorów bezstanowych i okien czasowych w planie produkcji nie zmienia wyniku zapytania, jeżeli operatory bezstanowe nie korzystają z wartości znacznika t_e [56]:

$$\sigma_p(\omega_w(S)) = \omega_w\left(\sigma_p(S)\right)$$

$$\sigma_p\left(map_p(S)\right) = map_p\left(\sigma_p(S)\right)$$

Obie własności pozwalają przenieść operator selekcji na początek planu produkcji, co odciąża operatory okna lub mapy przed obsługą krotek, które później zostaną odfiltrowana. Zbiór operatorów okien jest zawężony do okien czasowych, ponieważ okna fizyczne zliczają liczbę wystąpień krotek, w konsekwencji przestawienie kolejności realizacji tego operatora z operatorem selekcji prowadzi do zmiany wyniku.

2) Łączność produktu kartezjańskiego:

$$(S_1 \times S_2) \times S_3 = S_1 \times (S_2 \times S_3)$$

Własność ta pozwala na zmianę kolejności realizacji operatorów produktu kartezjańskiego, aby na końcu obsługiwać strumienie z wysokim stopniem niemonotoniczności wymagające stosowania złożonych struktur danych.

3) Umieszczenie selekcji na strumieniu zasilającym produkt kartezjański albo przeniesienie go na wyjście produktu kartezjańskiego nie prowadzi do zmiany zawartości strumienia wynikowego.

4) Operator Θ złączenia jest złożeniem operatora produktu kartezjańskiego i selekcji zdefiniowanej na atrybutach należących do obu strumieni zasilających produkt kartezjański.

$$\times_\Theta (S_1, S_2) = \sigma_p(S_1 \times S_2)$$

Algorytm 2.6. Algorytm optymalizatora regułowego.

```
Optimize(DAG dag, List reduceRules)
1)  for(Rule r: reduceRules){
2)      boolean reduce = true;
3)      while(reduce){
4)          ops = stwórz listę operatorów dla DAG;
5)          reduce = false;
6)          while(!ops.isEmpty()){
7)              Operator op = ops.removeFirst();
8)              if(r.check(op, DAG)){
9)                  r.reduce(op, DAG, ops);
10)                 reduce = true;
11)             }
12)         }
13)     }
14) }
```

Przyjęto założenie, że zapytanie wejściowe posiada operator Θ złączenia zdekomponowany na operator produktu kartezjańskiego oraz operator selekcji. Pozwala to swobodnie przemieszczać operator produktu kartezjańskiego w trakcie optymalizacji. Dopiero, gdy przeprowadzona zostanie analiza monotoniczności, operator produktu kartezjańskiego oraz wybrane predykaty są zastępowane operatorami Θ złączenia. Wymienione własności zapytania strumieniowego użyto do skonstruowania następującej listy reguł:

1) Reguła przeniesienia operatora produktu kartezjańskiego na początek planu produkcji. Warunek aktywacji reguły zachodzi, gdy jeden ze strumieni

wejściowych jest generowany przez operator selekcji oraz operator ten zasila tylko operator produktu kartezjańskiego. W wyniku uruchomienia reguły, operator selekcji jest przeniesiony na wyjście operatora produktu kartezjańskiego.

2) Reguła przeniesienia operatora produktu kartezjańskiego mocno nie monotonicznego na koniec planu produkcji. Warunek aktywacji zachodzi, jeżeli operator kartezjański *op1* ma jeden strumień wejściowy co najwyżej słabo monotoniczny, a drugi mocno niemonotoniczny. Dodatkowo strumień ten jest zasilany przez operator kartezjański *op2*, którego jeden strumień wejściowy jest również co najwyżej słabo monotoniczny, a drugi mocno niemonotoniczny. Reguła ta korzystając z własności łączności produktu kartezjańskiego, przenosi produkt kartezjański *op2* na koniec planu produkcji, tak że tylko jeden operator przetwarza strumień mocno nie monotoniczny.
3) Reguła wczesnego wyznaczania selekcji. Warunek aktywacji zachodzi, jeżeli operator selekcji jest poprzedzony operatorem mapy albo oknem czasowym. Dodatkowo, gdy operator selekcji jest poprzedzony operatorem produktu kartezjańskiego oraz warunek predykatu jest zbudowany tylko na atrybutach jednego ze strumieni zasilających. W wyniku uruchomienia reguły operator selekcji jest zastępowany miejscem z operatorem poprzedzającym. Kiedy operatorem poprzedzającym jest operator produktu kartezjańskiego, operator selekcji jest umieszczany na wejściu, do którego dołączony był strumień zawierający operandy predykatu.
4) Reguła tworzenia operatora Θ złączenia. Warunek aktywacji zachodzi, jeżeli operator selekcji jest poprzedzony operatorem produktu kartezjańskiego i predykat selekcji jest zbudowany na obu strumieniach zasilających. W wyniku uruchomienia reguły powstaje operator Θ złączenia.

Analizując działanie optymalizatora, po przetworzeniu pierwszej i drugiej reguły operatory produktu kartezjańskiego są względem siebie tak ułożone, aby zminimalizować liczbę strumieni mocno nie monotonicznych. Reguła trzecia przenosi operatory selekcji na pozycje leżące najbliżej źródeł danych. Po jej realizacji każdy operator produktu kartezjańskiego ma po sobie szereg operatorów selekcji, przy czym pierwszy z nich zostanie wybrany do skonstruowania operatora Θ złączenia. Finalnie reguła czwarta definiuje operator Θ złączenia.

Przedstawione algorytmy ilustrują jak można użyć monotoniczność do optymalizacji pojedynczych operatorów, jak i grafu ich połączeń. Element które je wyróżnia to brak podatności na zmieniające się parametry dynamiczne strumieni. O ile optymalizacja bazująca na statystykach zawiera czynnik losowy, o tyle optymalizacja oparta na monotoniczności jest stabilna. Chcąc rozbudować moduł optymalizacji zapytań strumieniowych należałoby dodatkowo dołączyć indeks dla szeregu następujących po sobie operatorów selekcji, jak również wybór predykatu dla operatora Θ złączenia oprzeć na analizie złożoności obliczeniowej i selektywności operatorów selekcji następujących po operatorze produktu kartezjańskiego.

2.11 Wnioski i uwagi

Prostota obsługi i intuicyjność logiki przetwarzania strumieniowego decyduje o jej akceptacji użycia w przemyśle. Konstrukcja takiej logiki musi oferować zarówno łatwą interpretację funkcjonalności pojedynczych operatorów jak i zapytań. Na początku w strumieniowych bazach danych [14] operator był traktowany jako czarna skrzynka, która posiada *n* wejść strumieniowych i *m* wyjść strumieniowych. Wadą takiego podejścia jest konieczność poznania szczegółów implementacyjnych operatorów w celu interpretacji działania całego zapytania. Inną konsekwencją tego podejścia jest powstanie wielu wersji strumieni. Podczas omawiania operatora agregacji zasygnalizowano, że niektóre implementacje generują krotki negatywne przed wstawieniem nowej wartości agregatu. Inne systemy przyjmują, że nowa krotka usuwa poprzednią. W skrajnym przypadku w systemie STREAM [64,4] wprowadzono trzy typy strumieni. Chcąc przetestować istniejące zapytanie na innym źródle danych dla tego systemu, może zajść konieczność przebudowy zapytania. Oznacza to, że system uniemożliwia swobodne wielokrotne korzystanie z wcześniej zbudowanego zapytania strumieniowego. Do tej sytuacji łatwo doprowadzić, gdy implementacja poszczególnych operatorów jest przedkładana nad logikę przetwarzania strumieniowego.

W tym rozdziale zaproponowałem nowe podejście do konstruowania modelu przetwarzania strumieniowego. Prace badawcze zostały skierowane na utworzenie listy reguł oraz cech, które opisują operatory i strumienie. Rdzeniem

zaproponowanego modelu przetwarzania strumieniowego jest zwięzła i niezmienna metoda interpretacji strumienia. Obecnie w literaturze możemy wyróżnić strumienie składające się wyłącznie z krotek pozytywnych, strumienie oparte na krotkach pozytywnych i negatywnych, strumienie z krotkami temporalnymi lub z krotkami tri-temporalnymi. W tej pracy zaproponowano model strumienia danych składający się z krotek temporalnych i negatywnych. Taka kompozycja pozwala na zwięzłą reprezentację wydarzenia, którego czas życia jest znany w chwili jego wystąpienia, jak i nieznany. Istnienie dwóch mechanizmów definiujących czas życia krotki sprawia, że strumienie o różnej zawartości mogą reprezentować obserwację tego samego wydarzenia. Powyższa sytuacja zmusza do zdefiniowania pojęcia podobieństwa strumieni. W przedstawionym rozwiązaniu przyjęto, że dwa strumienie są podobne, jeżeli tworzą tą samą tabele historii. Podsumowując, wprowadzona definicja strumienia łączy zalety oraz usuwa wady reprezentacji bazujących na krotkach temporalnych, pozytywnych i negatywnych, dzięki czemu nie ma potrzeby korzystać z wielu typów lub wersji strumieni w ramach jednego systemu. Własność ta ułatwia interpretację zapytań, co jest kluczowe z punktu widzenia użytkownika strumieniowej bazy danych.

Drugi obszar badawczy przedstawiony w tym rozdziale objął zagadnienie sposobu opisu operatorów strumieniowych. Problem polega na zdefiniowaniu operatora na takim poziomie abstrakcji, aby w najmniejszym stopniu ograniczyć jego sposób implementacji. Zagadnienie to można wyjaśnić następująco. Tworząc strumieniową bazę danych jesteśmy zobligowani do udostępnienia użytkownikowi operator złączenia. Gdy pojawi się nowa wydajniejsza jego realizacja, dołączenie operatora złączenia do strumieniowej bazy danych powinno wiązać się z minimalną liczbą zmian. Aby osiągnąć postawiony cel, operatory strumieniowe zostały skonstruowane uwzględniając:

- regułę przekształcania operatora logicznego w operator fizyczny,
- niezmienniki stanu, przejścia i propagacji oraz
- definicję monotoniczności strumieni.

Analizując stan literatury, uwzględnienie tych trzech składników jednocześnie jest podejściem nowatorskim. Do obecnych rozwiązań konkurencyjnych można zaliczyć model stosowany w systemie CEDR [16]. Słabością niego jest jak

dotąd słabo rozpoznana problematyka monotoniczności oraz potrzeba korzystania z bardzo rozbudowanej logiki do rozwiązywania prostych zapytań strumieniowych.

Kluczowym elementem zaproponowanego modelu przewarzania strumieniowego jest podział na logikę opisaną przez operatory logiczne oraz algorytmy opisane przez operatory fizyczne. Relację pomiędzy tymi poziomami abstrakcji określa def. 2.8. Cechuje się ona tym, że definicja operatora logicznego nie narzuca bezpośrednio algorytmu operatora fizycznego. Taka definicja operatorów logicznych pozwala w czytelny sposób przenieść definicję operatorów relacyjnych baz danych do środowiska strumieniowego. Wyjątek stanowią operatory, których definicja jest ściśle związana z własnościami strumieni, przykładowo do grupy tej należy okno liczebnościowe.

Drugie kryterium uwzględniane przy budowie operatorów to niezmienniki [88]. Formalizują one własności, jakie musi spełnić każdy operator strumieniowy. Dotąd przy budowie operatorów strumieniowych kierowano się zasadą, aby operator był nie blokowalny oraz przechowywał ograniczony stan. Systematyka niezmienników definiuje dodatkowo własności, jakimi muszą charakteryzować się operatory, aby można było je swobodnie łączyć w złożone zapytania.

Ostatnim elementem uwzględnionym przy budowie operatorów jest monotoniczność strumieni. Zwróćmy uwagę na dwie podstawowe różnice pomiędzy strumieniem a relacją. Strumień jest kolejką krotek, z kolei relacja jest zbiorem krotek bez ustalonego uporządkowania. Ponadto w wielu systemach przyjmuje się, że strumień zawiera krotki uporządkowane względem znacznika czasowego. Powyższe założenia wskazują, że konstrukcja strumieniowej bazy danych jest bliższa temporalnej bazie danych aniżeli relacyjnej bazie danych. W szczególności metody optymalizacji uwzględniające uporządkowanie danych zostały szerzej zbadane dla optymalizatorów temporalnych baz danych [67,78,77]. Rezultatem tych obserwacji jest klasyfikacja monotoniczności strumieni [37]. Monotoniczność jest to cecha, która zawęża możliwe wartości krotek występujące w strumieniu. O ile przepustowość i selektywność operatora mogą ulegać zmianą w trakcie działania systemu, monotoniczność strumienia jest cechą stałą. Własność ta mówi, jakiego typu dane

na wejściu się nigdy nie pojawią, bazując na tej przesłance można skorzystać z odpowiednio uproszczonej i wydajniejszej wersji operatora.

Rozdział ten stanowi fundament dla badań mających na celu zweryfikowanie tez pracy. Teza pierwsza dotyczy rozwoju języka deklaratywnego obsługującego zapytania strumieniowe. Aby móc skonstruować język deklaratywny, konieczny jest model logiczny operatorów. Problemem otwartym jest opis operatorów strumieniowych na takim poziomie abstrakcji, aby można było skonstruować elastyczny deklaratywny język zapytań. Teza druga dotyczy zagadnienia synchronizacji, która w znaczący sposób wpływa na wydajność systemów wielowątkowych. Tutaj opis modelu logicznego operatorów odgrywa kluczową rolę. Czym definicja operatora logicznego w mniejszym stopniu determinuje algorytm operatora fizycznego, tym większy istnieje wachlarz algorytmów jego realizacji. Co w konsekwencji pozwala zastosować większą liczbę wersji synchronizacji oraz metod zarządzania zasobami. Powyższa sytuacja sprawiła, że przed przystąpieniem do właściwych badań, została zbudowana algebra operatorów strumieniowych, której nowatorskie elementy opisano powyżej.

Aby potwierdzić poprawność i kompletność omówionej logiki operatorów, każdy z omówionych operatorów zaimplementowano i przeprowadzono testy w prototypowym systemie StreamAPAS. Operatory stanowe zostały dodatkowo przeanalizowane pod kątem uproszczeń wynikających ze znajomości monotoniczności strumieni zasilających. Uproszczenia te są wykorzystywane przez optymalizator warunkowy opisany w sekcji 2.10. Opis każdego z operatorów zawiera definicje algorytmu podstawowego. Wydajność poszczególnych operatorów nie jest przedmiotem tych badań. W chwili, kiedy istnieje wiele prac poświęconych algorytmom operatorów agregacji, łączenia, eliminacji duplikatów i innych. Problem stanowi sposób definiowania operatorów strumieniowych oraz weryfikacja ich poprawności, tak aby dysponując kilkoma operatorami fizycznymi $X_1, \ldots, X_n$ móc odpowiedzieć na pytanie czy są to te same operatory logiczne.

Rozdział 3. Język zapytań

Przetwarzanie strumieniowe jest w IT zagadnieniem znanym. Istnieją aplikacje przetwarzające dane audio i video, radiostacje internetowe, systemy wspomagające podejmowanie decyzji na giełdzie. Zauważmy, że każda z tych aplikacji posiada nie tylko inny interfejs użytkownika, ale również występują tam operacje na strumieniach danych charakterystyczne dla danej dziedziny. Ponadto często tylko kontrola całego procesu przetwarzania gwarantuje stabilność systemu. W konsekwencji bieżące systemy strumieniowe to najczęściej aplikacje skonstruowane dla konkretnego odbiorcy, które nie korzystają z strumieniowych baz danych. Coraz niższa cena sensorów sprawia, że w coraz większej liczbie dziedzin spotykamy się z problemami, które wydajniej rozwiązać korzystając z przetwarzania strumieniowego. Przykładem jest tutaj koordynowanie ruchem drogowym, naliczanie opłat na obszarze miast lub autostrad. Aby obniżyć koszty realizacji takich projektów, konieczne jest wprowadzenie platformy przetwarzania strumieni. Wiąże się z tym powstanie języka zapytań strumieniowych.

Do grona wyróżniających się projektów badawczych nad strumieniowymi bazami danych zaliczamy systemy: STREAM [8], ATLAS[92], Telegraph [74,60], TelegraphCQ [23] i Aurora-Borealis [1]. W tych badaniach tematem pierwszoplanowym jest realizacja operatorów strumieniowych. Słabo rozwijana pozostaje tematyka języków zapytań. Języki te można podzielić na trzy kategorie. Najpopularniejsze podejście polega na tym, że użytkownik bezpośrednio definiuje każdy z operatorów oraz połączenia strumieniowe w pliku tekstowym albo poprzez interfejs graficzny [1]. Drugie podejście opiera się na zastosowaniu języków proceduralnych, gdzie operacje strumieniowe są prezentowane przez pętle programowe zasilane krotkami. Najbardziej obiecujące jest zastosowanie języków deklaratywnych [8,92], ponieważ taka prezentacja zwalnia użytkowników od znajomości realizacji fizycznych operatorów. Z kolei na kompilator i optymalizator przenosi się odpowiedzialność za dobór algorytmów realizujących zapytanie.

Przed przystąpieniem do projektowania języka należy zrozumieć jego zakres użycia. Dlatego na wstępie porównajmy strumieniową bazę danych z bazami relacyjnymi poprzez analogię do rzeki i kubełków wody. Tak jak w rzece nie mamy kontroli nad jej intensywnością, podobnie w strumieniowej bazie danych nie ma kontroli nad intensywnością strumieni. Przeciwieństwem są relacyjne bazy danych, dla których analogiem są kubełki wody. Operacje realizowane w relacyjnej bazie danych można zilustrować przez proces przelewania wody pomiędzy naczyniami. Operacje te cechują się tym, że to baza danych decyduje o szybkości przetwarzania, innymi słowy to od szybkości przelewania wody zależy jak szybko zostanie rozwiązane zapytanie. W przypadku systemów strumieniowych, to strumień narzuca tempo przetwarzania. Zauważmy, że nie możemy wstrzymać rzeki, musimy zarządzać przepływem wody na bieżąco w przeciwnym wypadku dojdzie do jej wylania. To ryzyko jest analogią do przepełnienia buforów wynikającego z braku mocy obliczeniowej. Aby zapewnić stabilność systemu, ważna jest koordynacja całego procesu przetwarzania, mająca na celu dopasowanie mocy obliczeniowej i zasobów systemowych do bieżących wymagań. Czynnik ten jest znacznie ważniejszy dla strumieniowych baz danych aniżeli dla relacyjnych, ponieważ zmiany zachodzące w systemach strumieniowych mają większą dynamikę.

Obecnie w systemach informatycznych w celu przechowywania danych najczęściej korzysta się z relacyjnych baz danych, które implementują język SQL. Liczność tych aplikacji sprawia, że SQL stał się standardem znanym dla wielu osób pracujących w IT. Pojawia się pytanie, jaka powinna być składnia języka strumieniowych baz danych oraz czy rozbudowa składni języka SQL jest trafnym podejściem do problemu.

Uruchamianie zapytania strumieniowego można podzielić na trzy etapy. Na początku zapytanie strumieniowe jest analizowane pod względem poprawności przez kompilator, po czym jest zapisane do postaci przejściowej. Następnie użytkownik zleca uruchomienie zapytania, które od tego momentu zaczyna przetwarzać dane do chwili, kiedy użytkownik zleci jego wycofanie. W relacyjnych bazach danych po uruchomieniu zapytania oczekujemy na wynik, po czym jest ono wycofywane, a zatem z punktu widzenia użytkownika jest ono jedno etapowe. Oznacza to, że język zapytań powinien umożliwić zarządzanie oczekującymi na uruchomienie i uruchomionymi zapytaniami. W strumieniowych bazach danych

należy również szerzej rozumieć pojęcie zapytanie. W relacyjnych bazach danych zapytanie utożsamimy z frazą *select* w SQL. W systemie strumieniowym zapytanie jest reprezentowane przez graf podzadań[6]. Zastosowanie składni SQL zmuszałoby użytkownika do zarządzania wieloma zapytaniami oddzielnie, co byłoby czasochłonne i trudne. Dlatego język dla zapytań strumieniowych baz danych powinien udostępniać mechanizm grupowania zapytań w większe jednostki.

Istnieją projekty bazodanowe, w których znacząca część logiki biznesowej jest realizowana po stronie serwera. Do ich zdefiniowania powstało sporo rozszerzeń i całe języki takie jak T-SQL czy PL/SQL, które umożliwiają przeprowadzenie dodatkowych obliczeń po stronie bazy danych. Mimo to wzorce projektowe zalecają, aby baza danych wyłącznie realizowała operacje przechowywania danych. W opozycji stoją systemy strumieniowe, które mają służyć realizacji logiki biznesowej. Tylko w ten sposób można zagwarantować jakość przetwarzania rozumianą jako: zachowanie rygów opóźnień, ograniczeń pamięciowych, zapewnienie bezpieczeństwa na wypadek awarii jednego z węzłów w środowisku rozproszonym i ograniczanie skutków przeciążeń. Z tej perspektywy język zapytań strumieniowych powinien być lepiej przygotowany na dodawanie nowej funkcjonalności niż SQL.

Do czynników determinujących postać języka zapytań należy również zaliczyć typ strumieni i składniki jego schematu. Typ mieszany przedstawiony w rozdziale 2 jest opisany poprzez trzy składniki:

- schemat krotki definiujący zbiór atrybutów transportowanych przez krotkę,
- atrybuty tworzące klucz główny, które są konieczne do obsługi krotek negatywnych,
- monotoniczność.

Czym lepiej znamy własności strumienia tym skuteczniej jesteśmy w stanie zoptymalizować jego przetwarzanie. Należy mieć na uwadze, że badania nad strumieniowymi bazami danych są w toku, dlatego składnia języka powinna uwzględniać pojawienie się nowych elementów opisu strumieni.

Powyższą analizę streszcza poniższa lista wymagań stawianych językowi zapytań strumieniowych baz danych:

1. Zmiany w przetwarzaniu strumieniowym są bardziej dynamiczne niż w relacyjnych bazach danych, dlatego język zapytań powinien definiować pełne środowisko przetwarzania, co później pozwoli kompleksowo kontrolować jakość przetwarzania.
2. Aplikacje strumieniowe zazwyczaj wymagają specyficznych dla swojej dziedziny operatorów strumieniowych. Kluczowa jest swoboda w dodawaniu nowych operatorów/funkcji i ich automatycznego udostępniania z poziomu języka. Wymaganie to sugeruje budowę języka dynamicznego pozwalającego dołączać nowe konstrukcje.
3. Język musi efektywnie wspierać zarządzaniem grupami zapytań, ich: tworzeniem, uruchamianiem i wycofywaniem. Jako zapytanie rozumiemy tutaj nie tylko definicję przekształceń, ale również składniki określające QoS (Quality of service).
4. Z punktu widzenia użytkownika sposób reprezentacji danych ma wpływ na prostotę zapisu zapytań i jego zrozumiałość. Czynniki te mają duże znaczenie, jeżeli chcemy skrócić czas tworzenia i pielęgnacji tekstu zapytań. Aby to osiągnąć, składnia języka powinna udostępniać złożone typy danych.
5. Biorąc pod uwagę intensywne badania nad strumieniowymi bazami danych, pojawienie się rozszerzeń schematu strumieni jest wysoko prawdopodobne, dlatego składnia języka powinna usystematyzować kwestię zarządzania nimi w sposób kompleksowy.

Gdy odniesiemy składnię języka SQL do powyższych wymagań okazuje się, że główny problem leży w tym, że nie systematyzuje ona mechanizmów rozszerzania funkcjonalności. Wprowadzenie nowej składni języka jest zazwyczaj jedynym sposobem dodana nowej funkcjonalności. Poniżej podano istotne słabości języka SQL, jako wzorzec języka dla strumieniowych baz danych:

1. Kiedy powstawały relacyjne bazy danych ich główną rolą było trwałe przechowywanie danych. W konsekwencji język SQL nie był konstruowany z myślą o wyrażaniu złożonych obliczeń. Tą rolę przejęły obecnie rozszerzenia SQL.
2. Dodanie nowej funkcjonalności do języka SQL ogranicza się do definiowania własnych funkcji. Wprowadzenie szerszych zmian w funkcjonalności wymaga przebudowy składni języka.

3. Liczba sposobów przechowywania danych w relacyjnych bazach danych jest na tyle wysoka, że do obsługi tego zadania skonstruowano odrębny język DDL. Składnia jego zakłada, że schemat tabeli relacyjnej bazy danych składa się z listy atrybutów oraz atrybutów tworzących klucz główny. Chcąc rozszerzyć tą definicję albo dodać nową strukturę przechowywania danych należy skonstruować nową wersję składni. Jeżeli chcemy, aby język pozwalał dodawać nowe źródła danych oraz dopuszczał rozszerzanie opisu schematu strumieni powyższe rozwiązanie jest zbyt statyczne.
4. W relacyjnej bazie danych, użytkownik oczekuje na wynik po zadaniu zapytania. W strumieniowej bazie danych zapytanie jest rejestrowane a następnie przetwarzane w sposób ciągły. Ten model wymaga dodatkowego zarządzania zbiorem uruchomionych zapytań, z czym nie mamy do czynienia w SQL. Język SQL nie zawiera również dopracowanej i elastycznej składni, która precyzowałaby wymagania QoS. Przykładem jest składnia SQL dla bazy ORACLE, która do tego celu zaadaptowała frazę komentarzu przy zapytaniach SQL.
5. W języku SQL przyjęto prostą prezentację schematu rekordu, jako listy atrybutów. Rozwiązanie to staje się tym bardziej uciążliwe, czym bardziej złożone funkcje są wywoływane, ponieważ nie można pogrupować atrybutów tematycznie tak jak oferują to języki obiektowe.

Podsumowując, dopasowanie języka SQL do nowych potrzeb wiązało się dotąd z koniecznością tworzenia nowych wersji jego składni. Rzeczywistość wymaga nowych funkcjonalności, a czas potrzebny na zdefiniowanie oficjalnej nowej wersji języka SQL jest długi, dlatego producenci baz: DB2, Oracle, MySql, PostgreSQL i SQLServer tworzą własne rozszerzenia składni takie jak: PL/SQL, PL/pgSQL, pureXML i wiele innych. Idąc tym tropem można postawić tezę, że język SQL można tak rozbudować, aby obsługiwał zapytania strumieniowe. Idąc tym tropem istnieje duże ryzyko, że po kilku cyklach wzbogacania składni języka SQL powstanie język z niekonsekwentną i złożoną składnią.

Poszukując wzorców, z których można skorzystać podczas budowy języka zapytań wytypowano elementy składni: języków obiektowych, języka MDX, języka SQL i wyrażeń lambda.

Obecnie najpopularniejsze języki programowania są zorientowane na programowanie obiektowe. Istotą tego podejścia jest wydzielenie obiektów, które łączą stan i zachowanie. Obiektowy program jest definiowany przez zbiór komunikujących się ze sobą takich obiektów. Zapis programów w tej postaci jest czytelniejszy od programowania proceduralnego, ponieważ ułatwia stosowanie: hermetyzacji i abstrakcji. Jeżeli weźmiemy pod uwagę wymagania stawiane językowi zapytań strumieniowych, dla uzyskania lepszej czytelności, część funkcjonalności można przedstawić przy użyciu obiektów. Zgodnie z wymaganiem 3) język powinien umożliwiać zarządzanie zapytaniami oraz pracę na większych jednostkach niż pojedyncze zapytania. Stosując podejście obiektowe można zaprezentować pojedyncze zapytanie, jako obiekt typu *Query*, z kolei grupę zapytań objąć obiektem typu *Unit*. Dodatkowo parametry niefunkcjonalne takie jak QoS definiować poprzez metody obiektu *Unit*. Programowanie obiektowe pozwala zaprezentować typ *Query*, jako interfejs implementowany poprzez kilka klas. Jeżeli język zapytań udostępniałby takie elementy obiektowe, wtedy dodanie nowej funkcjonalności nie będzie wiązało się z potrzebą zmiany składni języka. Rozwiązaniem alternatywnym dla interfejsów jest zastosowanie AOP(Aspect Oreitned Programing), wtedy metody udostępniane z poziomu języka zapytań są dodatkowo adnotowane. Sam proces dołączania funkcjonalności do strumieniowej bazy danych można zrealizować poprzez mechanizm podobny do pakietów w języku Java. Pakiet zawiera grupę definicji klas, które implementują zadane interfejsy. Przyjmijmy, że tworzony jest system strumieniowy monitorujący stan poziomów rzek i konieczne jest stworzenie zapytań, które będą pobierały dane z sensorów. Aby to uzyskać należy zbudować klasy implementujące interfejs *Query*, które będą pobierały dane ze źródeł i transformowały je do strumieni danych. Następnie klasy te należy wstawić do pakietu. Chcąc z nich skorzystać wystarczy załadować do systemu pakiet i w języku zapytań stają się dostępne nowe obiekty typu *Query*, które rozszerzą język o brakujące elementy. Podsumowując, jeżeli podstawowe elementy zapytania są obiektami, rozszerzenie funkcjonalne systemu można przedstawić przy użyciu względnie prostej składni języka.

Istniejące języki zapytań strumieniowych przedstawiają użytkownikowi strumień, jako sekwencje krotek. Zauważmy, że podczas definiowania logiki operatorów strumieniowych posługiwaliśmy się głównie pojęciem tabeli historii.

Dzięki niej definicja operatora jest precyzyjna, z drugiej strony nie narzuca konkretnej realizacji fizycznej operatora. Idąc tym tropem język zapytań może operować na pojęciu tabeli historii, podczas gdy w rzeczywistości obliczenia są prowadzone na strumieniach. Tabela historii jest czytelniejsza dla użytkownika, ponieważ abstrahuje od typów krotek przesyłanych w strumieniach. Aby osiągnąć taką abstrakcję, nowatorskie jest spojrzenie na strumienie jak na obiekty wielowymiarowe, gdzie wyróżniamy operacje na wymiarach oraz operacje na wartościach. Obecnie z wielowymiarowością spotykamy się w hurtowniach danych. Analiza dużych wolumenów ma kluczowe znaczenie dla kierowania działalnością gospodarczą. Operacje te mogą służyć wspieraniu podejmowania decyzji (DSS), zarządzanie relacjami z klientami (CRM) i innymi procesami mającymi na celu analizę efektywności. W tych obszarach zastosowań, osoby korzystające z systemów informatycznych nie posiadają dużej wiedzy z zakresu IT, dlatego ważny jest prosty w obsłudze interfejs użytkownika. Z drugiej strony ze względu na liczbę danych, hurtownia danych musi być rozwiązaniem wydajnym. Z punktu widzenia użytkownika hurtownie danych wyróżniają się wielowymiarową reprezentacją danych. Przyjmijmy, że chcemy przeprowadzić analizę sprzedaży. Aby zbudować hurtownię należy wyróżnić fakty, które będą przedmiotem analizy oraz wymiary względem których dokonywane będzie drążenie danych. Dla podanego przykładu faktami są wartości sprzedaży produktów z uwzględnieniem daty, sprzedającego i kupującego. Wymiarami są czas, sprzedający i kupujący. Aby ułatwić posługiwanie się wymiarami są one zorganizowane w hierarchie. Przykładowo dziedzina czasu może zostać podzielona na dni, następnie na kwartały i lata. Użytkownik definiując zapytanie precyzuje, jakie przedziały dla poszczególnych wymiarów go interesują i na jakim poziomie szczegółowości. Następnie precyzuje operacje analityczne na ustalonym wycinku danych. Gdyby podobne operacje chcieć zrealizować na relacyjnej bazie danych, wtedy użytkownik musiałby być bardzo dobrze zaznajomiony z modelem bazy danych, ponieważ zapytanie będzie wymagało złączenia wielu tabel. Kiedy dane są zaprezentowane w postaci hiper-kostki, użytkownik koncentruje się wyłącznie na procesie analizy. Spośród istniejących języków zapytań dla hurtowni danych wiele z nich to rozszerzenia SQL, w konsekwencji kostka danych jest prezentowana przez tablicę.

Nowatorskie podejście prezentuje język MDX wywodzący się z systemu Hypernion, który przedstawia kostkę danych, jako strukturą hierarchiczną. Jej korzeń odpowiada wszystkim elementom kostki, następnie każdy z wymiarów tworzy pierwszy poziom drzewa. Kolejne poziomy wymiarów stanowią rozwinięcie węzłów na pierwszym poziomie. Korzystając ze składni zaproponowanej w MDX strumień również można zaprezentować poprzez obiekt wielowymiarowy. Przy takiej konwencji, do operacji na wymiarach zaliczamy wszelkie modyfikacje czasu życia obserwowanych wydarzeń. Podsumowując użytkownik przy użyciu wymiaru czasu definiuje czas aktywności krotek w tabeli historii. Abstrakcja w postaci obiektu z wieloma wymiarami może także posłużyć do reprezentacji operacji na indeksach strumieniowych, jako transformacji do postaci tabeli historii. Warto podkreślić, że obecne badania nad indeksami strumieniowymi nie idą w parze z rozwojem języków zapytań. Zaproponowany kierunek zastosowania struktur wielowymiarowych w języku zapytań strumieniowych jest podejściem obiecującym przy obsłudze złożonych indeksów, ponadto taka reprezentacja przybliża strumieniowe bazy danych do platformy zaawansowanych analiz prowadzonych na bieżąco, którą można by nazwać strumieniową hurtownią danych.

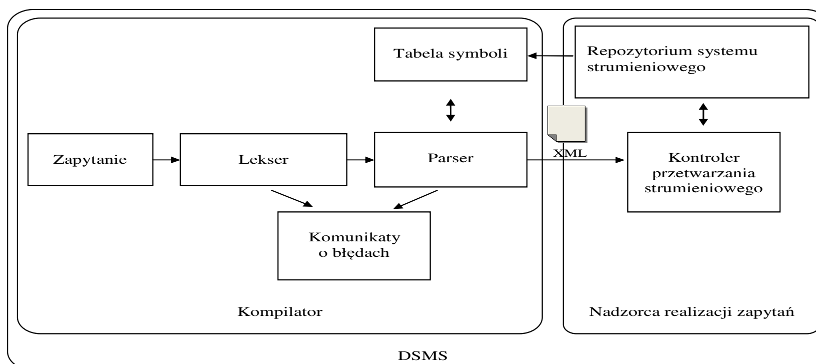
Na zakończenie, nie mniej istotną słabością języka SQL i języków dla zapytań strumieniowych jest schemat danych ograniczający się do prostej listy atrybutów. To ograniczenie sprawia, że bardziej rozbudowane funkcje muszą posiadać długą listę parametrów, co czyni kod mało czytelny. Alternatywą jest przechowywanie danych w formacie XML, lecz to rozwiązanie ogranicza się do danych tekstowych. Interesującą propozycją języka zapytań dla danych w formacie XML jest TQL [20,21,25,24]. Wprowadza on pojęcie drzewa informacji do zdefiniowania sposobu wydobywania i przetwarzania danych zawartych w dokumencie XML. Warty uwagi jest sposób prezentacji danych hierarchicznych. W takich językach jak pureXML lub XPath pobranie danych z węzła wymaga zdefiniowania ścieżek przy użyciu wyrażeń regularnych. To rozwiązanie jest wygodne, jeżeli wykonujemy obliczenia na jednym typie węzłów. Jeżeli zapytanie korzysta z wielu typów węzłów, wtedy wielokrotne definiowanie pełnych ścieżek do węzłów dokumentu XML czyni zapytanie coraz mniej czytelnym. W TQL, szereg pojedynczych ścieżek jest zastąpiony drzewem informacji, dzięki czemu zapytanie jest zwarte i łatwiejsze do zrozumienia.

Podejście to zaadaptowałem w języku zapytań strumieniowych, gdzie posłużyło do obsługi złożonych struktur danych.

Popularność i łatwość korzystania z języka SQL sprawia, że zastosowanie fragmentów jego składni do budowy języka zapytań dla strumieniowych baz danych jest istotne z punktu widzenia użytkowników. Z tego powodu w rozwijanym języku skorzystano ze znanej składni SELECT FROM. Została ona jednak mocno zmieniona poprzez zastosowanie nowego modelu danych, elementów obiektowości oraz rozszerzeń składni wyrażeń arytmetycznych. Zebrane wyniki prac zostały opublikowane w pracach [40,42]. W tym szeroki opis nowatorskiego języka zapytań został wydany w międzynarodowym czasopiśmie Complex Intelligent Systems and Their Applications [43].

3.1 Kompilator języka StreamAPAS

Przed przystąpieniem do omówienia składni języka oraz szczegółów implementacyjnych kompilatora, na rys. 3.1 przedstawiono przepływ sterowania podczas uruchamiania zapytania zastosowany w autorskim systemie StreamAPAS. Na modelu wyróżniamy moduł kompilatora oraz moduł zarządzający strumieniową bazą danych. Rolą modułu zarządzającego jest uruchamianie, zatrzymywanie, optymalizacja i koordynacja przetwarzania strumieniowego. Kompilator jest oddzielnym modułem, który analizuje tekst zapytania i generuje plan produkcji w postaci przejściowej zapisanej do formatu XML. Postać przejściowa składa się z dwóch części. W pierwszej zdefiniowane są schematy struktur danych. Druga część zdefiniuje operacje z uwzględnieniem typów danych oraz ograniczeń systemu strumieniowego. Uruchomienie zapytania polega na utworzeniu operatorów oraz strumieni przez DSMS w oparciu o plan produkcji zapisany w formacie XML. Wprowadzenie postaci przejściowej sprawia, że kompilator oraz DSMS nie są ze sobą ściśle powiązane strukturami danych, co ułatwia równoczesny rozwój obu elementów systemu. Jeżeli w DSMS jest rozwijana nowa funkcjonalność, zapis przejściowy umożliwia jej swobodne testowanie niezależnie od zaawansowania rozwoju prac nad językiem zapytań.



Rys. 3.1. Przebieg kompilacji zapytań

3.2 Typy danych

System StreamAPAS jest napisany w języku Java. Aby przy jego użyciu można było uruchamiać różne systemy strumieniowe przyjęto, że atrybuty krotek mogą być dowolnego typu obiektowego zdefiniowanego przy użyciu języka Java. Przy czym zbiór typów wspieranych przez operatory rozwijanego języka zawęża się do:

- String,
- Double,
- Float,
- Long,
- Integer,
- Boolean.

Wyjątek stanowią operatory odczytu oraz przypisania wartości, które obsługują dowolny typ obiektowy. Taka konstrukcja pozwala przekazywać z wejścia na wyjście dane dowolnego typu, o ile nie są one modyfikowane przez zapytanie strumieniowe. Jeżeli użytkownik dołączy do systemu odpowiednio przygotowany pakiet funkcji, istnieje dodatkowo możliwość realizacji funkcji na typach zdefiniowanych przez użytkownika. Podstawowy zbiór operatorów wylistowano w tabeli 3.1.

Tabela 3.1. Zbiór operatorów arytmetycznych i logicznych dostępnych w StreamAPAS

Etykieta operatora	Opis operatora
ASSIGN	zapis wartości do zmiennej
VAL	odczyt wartości zmiennej dowolnego typu
EQ	==
GE|EQ	>=
GE	>
LO|EQ	<=
LO	<
LO|GE	!=
SUM	+
SUB	-
MUL	*
DIV	/
AND	iloczyn logiczny
OR	suma logiczna
XOR	alternatywa wykluczająca
NOT	negacja logiczna
NEG	wartość przeciwna

Kompilacja wyrażenia matematycznego składa się z analizy składniowej, w wyniku której powstaje drzewo rozbioru. Następnie przeprowadzana jest analiza semantyczna mająca na celu sprawdzenie poprawności zastosowanych operatorów oraz uzupełnienie informacji o typach operandów. Analiza semantyczna sprawdza czy argumenty operatorów są zgodne z definicją oraz czy jest konieczna konwersja typów. Kompilator języka StramAPAS automatycznie dodaje rzutowanie typów na typy ogólniejsze. Posiadając wyrażenie matematyczne w postaci drzewa rozbioru, analiza semantyczna składa się z następujących etapów:

- Odczyt typów operandów,
- Ustalenie typu operatora, tak aby istniały konwersje typów operandów do typu na którym operator wykonuje obliczenia,

- Wyznaczenie typu wynikowego operatora,
- Wstawienie do drzewa rozbioru brakujących operacji konwersji operandów.

Dopuszczalne typy operandów dla podstawowych operatorów zdefiniowano w tabeli 3.4. Przykładowo nie istnieje operator sumowania z operandami typu Double i Integer, tak jak w wyrażeniu: 2 + 2,1. Aby wyliczyć to wyrażenie analizator semantyczny konwertuje wartość Integer na Double, następnie korzysta z operacji sumowania zdefiniowanej dla typu Double. Promocja na typy szersze jest zbudowana w oparciu o tabelę 3.2. W odróżnieniu do języka Java w języku StreamAPAS udostępniono konwersje typu Boolean na Integer realizowaną zgodnie z tabelą 3.3.

Tabela 3.2. Domyślne rozszerzanie typów

Operator	Integer	Long	Float	Double	Boolean	STRING
Integer		•	•	•		•
Long				•		•
Float				•		•
Double						•
Boolean	•	•	•	•	•	•

Tabela 3.3. Konwersja typu Boolean do Integer

Boolean	Integer
true	1
false	0

Tabela 3.4. Typy na których są zdefiniowane operatory w StreamAPAS

Operator	Integer	Long	Float	Double	Boolean	STRING
EQ	•	•	•	•	•	•
GE|EQ	•	•	•	•		•
GE	•	•	•	•		•
LO|EQ	•	•	•	•		•
LO	•	•	•	•		•
LO|GE	•	•	•	•	•	•
SUM	•	•	•	•		•
SUB	•	•	•	•		
MUL	•	•	•	•		
DIV	•	•	•	•		
AND	•	•			•	
OR	•	•			•	
XOR	•	•			•	
NOT					•	
NEG	•	•	•	•		

3.3 Drzewo atrybutów

Do opisu złożonych struktur takich jak dane przestrzenne, schemat krotki zdefiniowany przy użyciu hierarchi jest łatwiejszy zarówno do zrozumienia, jak i definiowania na nim przekształceń. Często wśród schematów krotek możemy wyróżnić fragmenty podobne. Przykładowo, jeżeli pewna grupa strumieni przekazuje informacje o lokalizacji w układzie kartezjańskim, należy spodziewać się, że do opisu położenia użyto atrybutów x, y, (z). Jeżeli krotka przekazuje komunikaty o alarmach w systemie, prawdopodobnie jej schemat zawiera atrybuty: *nazwa* i *opis*. Aby umożliwić tworzenie podtypów w języku StreamAPAS wprowadzono drzewo atrybutów przedstawiające schemat krotki, jako hierarchię z nazwanymi węzłami. W odróżnieniu do XML, drzewo atrybutów definiuje typ danych przechowywany w węzłach.

Drzewo atrybutów jest zdefiniowane, jako zagnieżdżona lista węzłów posiadających nazwy i wartości obiektowe. Węzły nieprzechowujące wartości, są

typu *NONE*, służą one definiowaniu grup atrybutów, gdzie sam węzeł grupy nie ma ze sobą skojarzonej wartości. Z kolei etykieta korzenia drzewa atrybutów oznacza nazwę kolekcji danych. Poniżej przedstawiono składnię wyrażeń z użyciem drzew atrybutów:

η ::= etykieta wyrażenia

$*name* etykieta do danych znajdującego się w przestrzeni nazw kolekcji

name etykieta węzła

A, B ::= wyrażenie arytmetyczne

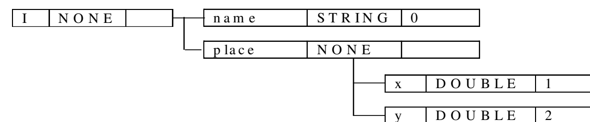
true pod-drzewo dla bieżącego węzła

$\eta[A]$ lokalizacja bieżącego węzła

A, B kompozycja wyrażeń

name = exp przypisanie do węzła o nazwie *name* wartości wyrażenia arytmetycznego *exp*

Od strony implementacyjnej krotka zawiera wartości atrybutów zapisane w liście indeksowanej od 0, na której zdefiniowano operacje zapisu i odczytu. Drzewo atrybutów mapuje wartości pomiędzy węzłami a komórkami listy. Aby zrozumieć działanie drzewa atrybutów, na rys. 3.2 przedstawiono przykładowy schemat krotki zawierający informacje o nazwie oraz położeniu alarmu wygenerowanego przez system monitorujący. Każdy z węzłów jest opisany przez trzy wartości: pierwsza oznacza nazwę węzła, druga jego typ, trzecia jest indeksem wskazującym na komórkę listy, gdzie przechowywana jest wartość atrybutu. Innymi słowy, jeżeli zapytanie korzysta z wartości węzła *I.place.x*. Kompilator pobiera wartość indeksu dla tego węzła, następnie dla każdej krotki odczytywana jest wartość elementu listy o indeksie 1.



Rys. 3.2. Przykładowe drzewo atrybutów

Z każdym węzłem jest skojarzona przestrzeń nazw zawierająca etykiety węzłów do niej należących. W trakcie kompilacji przestrzenie nazw są odkładane

na stosie. W danym momencie są dostępne tylko etykiety należące do bieżącej przestrzeni nazw. U spodu leży bazowa przestrzeń, która zawiera etykiety do kolekcji danych np. strumieni. Jeżeli wybierzemy kolekcję danych o nazwie *I* wtedy na stos zostaje odłożona przestrzeń nazw zawierająca etykiety, które opisują elementy kolekcji danych *I*, ponadto przestrzeń ta staje się bieżącą przestrzenią nazw. Jeżeli etykieta jest poprzedzona znakiem '$', wtedy następuje skok do bazowej przestrzeni nazw.

Powróćmy do przykładu z rys. 3.2. Zarówno węzeł jak i całe poddrzewo może być operandem wyrażeń. Ścieżka do węzła składa się z etykiet obiektów oddzielonych kropką. Składnię tą zaczerpnięto z języków obiektowych, gdzie istnieje pod nazwą 'dot-sytax'. Jeżeli chcemy pobrać wartość węzła *x* należy zbudować ścieżkę *I.place.x*. Wyrażenie *I.place.x* jest poprawne pod warunkiem, że bieżącą przestrzenią nazw jest przestrzeń bazowa. Aby uniezależnić się od bieżącej przestrzeni nazw należy podać ścieżkę: *$I.place.x*, ponieważ znak '$' determinuje przeglądanie bazowej przestrzeni nazw. Jeżeli operandem ma być drzewo należy użyć składni *[true]*. Przykładowo zapis *$I.place[true]* oznacza drzewo o korzeniu w węźle *place*.

Podstawową czynnością podczas budowania drzewa atrybutów jest operacja zapisu wartości do węzła, do jej realizacji służy składnia *name = exp*. Jeżeli w bieżącej przestrzeni nazw nie istnieje węzeł o nazwie *name*, wtedy jest tworzony nowy węzeł o typie identycznym jak typ wyrażenia *exp*. Następnie do węzła jest zapisana wartość wyrażenia. Jeżeli w bieżącej przestrzeni nazw istnieje węzeł *name*, wtedy weryfikowane jest czy typ zapisany w węźle jest zgodny z typem zwracanym przez wyrażenie *exp*. Jeżeli istnieje niezgodność, kompilacja jest przerwana z odpowiednim komunikatem błędu. Aby w bieżącej przestrzeni nazw zbudować węzeł *x* o wartości 1 należy zapisać *x = 1*. Wyrażenia przypisania można łączyć ze sobą w większe ciągi przy użyciu składni kompozycji. Stworzenie węzłów *x* i *y* z wartościami odpowiednio *1* i *0* sprowadza się do zapisu *x = 1,y = 0*. Na zakończenie chcąc zbudować węzeł *place* zawierający wcześniej zdefiniowane węzły x i y, należy skorzystać ze składni lokalizacji: *place[x=1,y=0]*. Jeżeli do lokalizacji zastosujemy składnię 'dot-sytax', wtedy otrzymamy zapis: *place.x=1, place.y=0*. Ze względu na złożoność zapisu pierwsza wersja jest czytelniejsza. Warto zauważyć, że po zdefiniowaniu węzła można użyć go w następujących później wyrażeniach. Jako przykład zbudujmy drzewo atrybutów *d*,

które zawiera węzły *x=2*, *y=1* oraz wartość średnią zapisaną w węźle *avg*: *d[x=2,y=1, avg=(x+y)/2]*.

W językach programowania typy danych dzielimy na proste oraz złożone, czyli struktury zbudowane z typów podstawowych. Podczas kompilacji każda etykieta zmiennej jest uzupełniana o typ. Następnie po naniesieniu typów na drzewo rozbioru składniowego sprawdzana jest poprawność semantyczna programu. W językach programowania szeroko stosowane są typy nazwane. Rozwiązanie to polega na tym, że każda struktura złożona posiada swoją nazwę jednoznacznie ją identyfikującą. Takie podejście upraszcza konstrukcję kompilatora, ponieważ weryfikacja zgodności typów jest realizowana poprzez zgodność nazw typów. Z drugiej strony może zaistnieć sytuacja, że w programie będą istniały dwie identyczne struktury z nazwami *A* i *B*. Pomimo to funkcja posiadająca argument typu A, nie może zostać wywołana z argumentem typu B. Aby wyliczyć wartość funkcji dla argumentu typu B, należy go wpierw skonwertować do typu A. Postać definicji typów złożonych ma wpływ na organizację i czytelność kodu źródłowego oraz narzuca sposób korzystania ze wzorców projektowe. W językach programowania korzystamy z typów nazwanych, dlatego ogromne znaczenie ma zdefiniowanie współdzielonych typów podczas tworzenia aplikacji w zespole wieloosobowym. W przeciwnym wypadku stworzone oddzielnie fragmenty programu będą musiały intensywnie korzystać z konwersji typów. Aby przyspieszyć tworzenie aplikacji konieczne jest korzystanie z gotowych komponentów takich jak: biblioteki, technologie i platformy. W tym momencie program może posiadać wiele typów oznaczających podobne struktury, co skutkuje mniejszą czytelnością kodu oraz wieloma konwersjami typów. Relacyjne bazy danych są zorientowane na obsługę ograniczonej liczby typów. Inne struktury danych należy zapisać w postaci binarnej. Wiąże się to z ograniczeniem, że nie można na binarnym typie definiować warunków zapytania. Taka konstrukcja bazy danych sprawia, że tworzenie nowych typów danych przez użytkownika jest mało atrakcyjne w odniesieniu do rozwiązań oferowanych w językach programowania. Zgodnie z wymaganiem 1) strumieniową bazę danych należy postrzegać jako platformę, do której można dołączyć własne operatory w celu uzupełnienia jej funkcjonalności. Dodawanie nowych operatorów bez możliwości konstruowania własnych typów jest rozwiązaniem nie pełnym. W odróżnieniu do typów nazwanych, w języku StreamAPAS zastosowano podobieństwo strukturalne. Idea polega na tym,

że kompilator sprawdza czy w danych drzewie atrybutów znajdują się elementy wymagane przez funkcję. Przyjmijmy, że chcemy zdefiniować typ opisujący punkt w przestrzeni 2D. Korzystając z podobieństwa strukturalnego taki typ można zdefiniować na kilka sposobów. Przykładowo takim typem jest drzewo atrybutów, którego dwa pierwsze pod-węzły są typu *Integer*. Dopuszczalna jest również definicja, że jest to drzewo atrybutów z pod-węzłami typy *Integer* o nazwach: *x* i *y*. Warto zauważyć, że obie definicje dopuszczają istnienie drzewa atrybutów z większą liczbą węzłów. Dla tak stworzonej definicji, drzewo o korzeniu w węźle *place* na rys. 3.2 jest zgodne z definicją punktu 2D. Od strony implementacyjnej, podobieństwo semantyczne jest implementowane przez użytkownika w języku Java. Pozwala to na zdefiniowanie bardziej złożonych zasad. Przykładowo funkcja dopuszcza tylko struktury z dwoma danymi typu Integer, gdyż trzy typy Integer oznaczałyby punkt w trój-wymiarze.

Podsumowując, posługiwanie się drzewem atrybutów przez użytkownika jest niewiele trudniejsze niż definiowanie prostych wyrażeń w SQL. Składnia drzewa atrybutów umożliwia użytkownikowi definiowanie struktury danych oraz planu obliczeń w jednym miejscu. Sprawia to, że zapis jest zwięzły i czytelny. Dzięki temu, że operandem może być fragment drzewa atrybutów, można uniknąć potrzeby podawania długiej listy atrybutów, tak jak to ma miejsce w języku SQL. Dużym ograniczeniem w swobodnym korzystaniu z gotowych bibliotek programistycznych jest mnogość podobnych typów. W języku StreamAPAS zastosowano drzewo atrybutów oraz podobieństwo strukturalne. Takie połączenie sprawia, że pojedyncze drzewo atrybutów może być postrzegane, jako instancja wielu typów. To nowatorskie podejście ma na celu otworzyć język zapytań na dodawanie nowej funkcjonalności a ograniczyć trudności związane z zarządzanie wieloma typami. Inną korzyścią tego podejścia jest zmniejszenie liczby funkcji, które konwertują typy danych, ponieważ nie ma konieczności konwertować danych strukturalnie podobnych.

3.4 Język StreamAPAS

Plan przetwarzania jest zdefiniowany poprzez acykliczny graf skierowany (Directed Acyclic Graph). Aby zdefiniować DAG w języku StreamAPAS korzysta się

z obiektów *Task*. Pojedynczy *Task* definiuje kilka operatorów połączonych ze sobą strumieniami. Cegiełki te następnie są ze sobą łączone w celu zbudowania całego systemu przetwarzania. Do zdefiniowania Task można skorzystać z dwóch składni. Pierwsza jest oparta na składni Select-From z języka SQL. Druga reprezentacja tworzy obiekty *Task* bezpośrednio poprzez metody fabrykujące.

W odróżnieniu do języków CQL, SQL i MDX, w języku StreamAPAS wprowadzono pojęcia: *Unit* i *Task*. Zarządzanie przetwarzaniem odbywa się poprzez obiekty *Unit*, tylko one mogą być uruchamiane lub wycofywane. Obiekt ten reprezentuje instancje DAG oraz dodatkowe parametry konieczne do uruchomienia przetwarzania strumieniowego. Aby zbudować i uruchomić obiekt *Unit* korzystamy ze składni:

```
<nazwa unit> run
Begin
  <definicja DAG przy użyciu obiektów Task>
End;
```

Jeżeli chcemy tylko sprawdzić czy *Unit* się kompiluje bez jego uruchamiania zastępujemy parametr *run* na *compile*:

```
<nazwa unit> compile
Begin
  <definicja DAG przy użyciu obiektów Task>
End;
```

Po skompilowaniu stworzony *Unit* jest dostępny w systemie pod wskazaną nazwą. Aby go uruchomić należy wywołać polecenie:

```
<nazwa unit> run;
```

Aby zatrzymać i wycofać uruchomiony *Unit* korzystamy ze składni:

```
<nazwa unit> stop;
```

3.5 Fabryka danych

Na początku rozwoju strumieniowych baz danych, przyjęto że dane przesyłane są przez strumienie to proste kolejki zawierające krotki o tym samym schemacie i w taki sposób prezentowano je w językach zapytań. Jak opisano w rozdziale 2, znacznie wygodniej posłużyć się pojęciem tabeli historii w celu

zdefiniowania logiki zapytania, ponieważ opis na takim poziomie abstrakcji nie wymaga od użytkownika znajomości szczegółów implementacyjnych operatorów.

Do tej pory w literaturze nie zwrócono uwagi na to, że pojęcie tabela historii rozszerza definicję danych strumieniowych. Aby przyspieszyć działanie zaawansowanych operatorów strumieniowych takich jak strumieniowe wyliczanie histogramów lub operacje przestrzenne, korzysta się z dedykowanych indeksów. Po zdefiniowaniu warunków ekstrakcji z takiego indeksu otrzymujemy wynik będący tabelą historii. W przypadku skrajnym indeks którego jednym z wymiarów jest oś czasu to również przykład danych strumieniowych. Obecnie rozwój strumieniowych baz danych koncentruje się na danych zapisanych w strumieniach oraz tabelach. Moim zdaniem w przyszłości strumieniowe bazy danych przekształcą się w uniwersalne platformy przetwarzania strumieniowego. W takich systemach źródłem danych strumieniowych będą strumienie, tabele oraz bardziej złożone obiekty takie jak: indeksy lub serwisy. Aby język zapytań mógł swobodnie korzystać z takich źródeł wprowadzono pojęcie fabryki danych, która abstrahuje od sposobu przechowywania danych, z punktu widzenia użytkownika jest ona źródłem tabeli historii.

Fizyczną realizacją fabryki danych może być tabela, strumień lub inna struktura danych. Z poziomu języka StreamAPAS, fabryka danych jest tworzona poprzez obiekt *Task*. Obecnie zdefiniowane operatory obsługują tylko fabryki strumieni, dlatego składnia frazy Select-From zawęża się do tego typu danych. Tworzenie innych typów fabryk danych jest możliwe poprzez metody tworzące obiekty *Task*, co zostanie omówione na końcu rozdziału. Fabryka danych jest obiektem reprezentującym repozytorium danych. Sposób odczytu tych danych jest definiowany poprzez obiekt Proxy. Szczegóły korzystania z tego obiektu zostaną przedstawione podczas omawiania składni Select-From.

3.6 Składnia Select-From

Poniżej podano rdzeń frazy Select-From, który wzorowano na SQL, CQL i MDX:

```
 SELECT definicja_fabryki_danych
[ FROM definicja_źródła, definicja_źródła,.... ]
[ WHERE wyrażenie_where ]
[ GROUP BY identyfikatorAtrybutu, identyfikatorAtrybutu,.... ]
[ HAVING wyrażenie_having ]
```

Podczas konstruowania składni Select-From dla języka StreamAPAS wzięto pod uwagę fakt, że zestaw operacji służący do analizy danych jest bogaty i stale się powiększa. Istotne jest, aby projektant aplikacji mógł dołączyć brakujące funkcjonalności do systemu. Z myślą o osiągnięciu tego celu połączono elementy obiektowości oraz składnię języka SQL. Operatory strumieniowych baz danych oraz relacyjnych baz danych można podzielić na cztery grupy:

1) operatory na zbiorach,
2) operatory okien,
3) operatory agregacji,
4) i pozostałe.

W oparciu o powyższy podział operatorów prześledzimy kolejne fragmenty frazy Select-From w języku StreamAPAS.

3.6.1 Operacje na zbiorach

Analizując składnię SQL najmniej konsekwentny jest sposób wyrażania operatorów na zbiorach. Operator eliminujący duplikaty jest definiowany wewnątrz klauzuli `select`.

```
SELECT [ALL | DISTINCT | DISTINCTROW ]...
```

Operator sumy zbiorów jest definiowany przy użyciu składni, która dodatkowo udostępnia operator eliminacji duplikatów:

```
SELECT ...
UNION [ALL | DISTINCT] SELECT ...
```

Z kolei operatory różnicy zbiorów lub części wspólnej zbiorów są zdefiniowane przez składnię:

```
SELECT ...
MINUS SELECT ...

SELECT ...
INTERSECT SELECT ...
```

Podsumowując gramatyka SQL posiada nadmiarowość, jest niekonsekwentna, ponadto dodanie do systemu nowego operatora reprezentującego inną operację na zbiorach zawsze wiąże się z rozszerzeniem gramatyki. Takie cechy nie pozwalają na to, aby swobodnie budować własne operatory należące do tej grupy i automatycznie dołączać je do systemu.

W języku StreamAPAS zdecydowano się wyrażać takie operatory przy użyciu metod statycznych. Metody operują na fabrykach danych oraz dodatkowych parametrach oraz zwracają jako wynik nową fabrykę danych. Przykładowo chcąc zbudować strumień *Out* będący sumą dwóch strumieni o nazwach *I* i *II*, należy użyć wyrażenie:

```
Out{Set.union(I{}, II{})}
```

Korzystamy tutaj z metody statycznej `union` zdefiniowanej w klasie `Set`. Argumentami metody są dwie fabryki strumieni o identycznych schematach krotek. Wynikiem jest fabryka generująca strumień będący sumą zgodnie z definicją przedstawioną w punkcie 2.8.7. Zastosowana składnia języków obiektowych wdrożona do języka zapytań wyróżnia się dwiema zaletami. Metody statyczne umożliwiają wyrażanie dowolnej funkcji, co otwiera język zapytań na dodawanie nowych operatorów. Zastosowane elementów obiektowości powala grupować operatory ze względu na klasy. Korzystając z tej cechy można uporządkować operatory tematycznie, co ułatwia posługiwanie się bibliotekami operatorów oraz poprawia czytelność kodu.

Do grupy operatorów realizujących obliczenia na zbiorach zaliczamy także wyliczanie różnicy zbiorów oraz wyznaczanie elementów unikalnych. Podobnie te operatory są udostępniane poprzez metody statyczne. Przykładowo operator różnicy ma składnię:

```
Set.minus(DatasetI aS, DatasetI aR, String aKeyS, String aKeyR)
```

Argumentami *aS* i *aR* są fabryki strumieni. Argument *aKeyS* definiuje klucz składający się z węzłów krotki dla strumienia *aS*. Argument *aKeyR* definiuje klucz dla strumienia *aR*. Oba atrybuty to tekst będący listą identyfikatorów węzłów drzewa atrybutów odseparowanych znakiem ‘,’: **`<identyfikatorAtrybutu1>,<identyfikatorAtrybutu2>, …, <identyfikatorAtrybutuN>`**. Identyfikator to lista nazw węzłów na ścieżce od korzenia do wskazywanego węzła oddzielona znakiem ‘.’, przy czym pominięty jest pierwszy węzeł reprezentujący nazwę fabryki danych. Zgodnie z definicją operatora przedstawioną w punkcie 2.8.6, krotka *t* pobrana ze strumienia *aR* usuwa krotkę ze strumienia *aS*, jeżeli jej wartości węzłów *aKeyS* są identyczne do wartości węzłów *aKeyR* należących do krotki *t*. Przykładowe użycie tej metody demonstruje poniższy przykład:

```
Out{Set.minus(I{}, II{}, "val", "val")}
```

Operator wyliczający elementy unikalne jest udostępniany poprzez metodę:

```
Set.distinct(DatasetI aD, String aKey)
```

Argument *aD* to wejściowa fabryka strumieni. Argument *aKey* definiuje węzły drzewa atrybutów względem których krotki wynikowe mają być unikalne. Poniżej podano przykładowe wywołanie tej metody:

```
P0{Set.distinct(I{}, "val")}
```

3.6.2 Klauzula select

W relacyjnych bazach danych do utworzenia tabeli służy język DDL. Wraz z pojawieniem się nowych metod składowania danych, składnia DDL jest rozszerzana. Zastosowanie takiego podejścia do definiowania fabryk danych jest niewłaściwe. Nowy typ fabryki danych może utworzyć projektant i dołączyć go do systemu. W takim przypadku składnia języka zapytań musi być na tyle bogata, aby umożliwiała zarządzanie dowolnym typem obiektów wprowadzonym przez użytkownika. Dlatego instancja fabryki danych jest reprezentowana przez obiekt konfigurowany poprzez wywoływanie jego metod. Zaletą tego podejścia jest także czytelność kodu zapytania, ponieważ w jednym miejscu definiowana jest struktura danych oraz proces generacji danych. Składnia elementu **`definicja_fabryki_danych`** jest następująca:

```
<definicja_fabryki_danych> ::= <nazwa_kolekcji_danych>{ <element_definicji>,
<element_definicji>,… }
<element_definicji> ::=<drzewo_atrybutów> | <metoda(listaArgumentów)>
```

Utworzona fabryka danych jest dostępna poprzez etykietę: `nazwa_kolekcji_danych`. Wyrażenie objęte w klamrach {…} jest związane z kontekstem obiektu fabryki danych oraz reprezentuje korzeń struktury danych. Sprawia to, że wewnątrz tego kontekstu można swobodnie korzystać z metod udostępnionych przez fabrykę danych oraz definiować schemat strumienia. Schemat danych jest definiowany poprzez drzewo atrybutów, a parametry są ustawiane poprzez metody fabryki. Obecnie system StramAPAS udostępnia wyłącznie operatory strumieniowe, dlatego wynikiem zapytania zdefiniowanego przez frazę Select-From jest zawsze strumień. Gdy w kolejnej wersji dołączone zostaną tabele relacyjne, wtedy rozważane jest dołączenie etykiety typu fabryki danych zgodnie z poniższym wzorem:

```
<definicja_fabryki_danych> ::=
<nazwa_kolekcji_danych>:<typ_fabryki_danych>{…}
```

Obiek fabryki strumieni udostępnia dwie metody:

addPK(Node aNode) –do listy atrybutów tworzących klucz główny dodaje kolejny węzeł lub węzły drzewa atrybutów.

clearPK() – usuwa atrybuty tworzące klucz główny.

Poniżej przedstawiono przykładową definicję strumienia *tmp* z kluczem głównym na atrybucie *tmp.name* zbudowanym w oparciu o strumień *I* przedstawiony na rys. 3.2:

```
select tmp{$I.name, sum = $I.place.x + $I.place.y, $I.place[true],
           addPK(name)}
from I{}
```

W wyniku przetworzenia powyższego wyrażenia powstaje strumień, którego schemat jest opisany przez drzewo atrybutów o korzeniu *tmp*. Jego atrybutami na pierwszym poziomie są węzły: *name*, *sum* i *place*. Dodatkowo węzeł *place* posiada pod-węzły: *x* i *y*. Na koniec, przy użyciu metody `addPK` zdefiniowane są atrybuty tworzące klucz główny. Kolejność członów jest istotna, ponieważ wyrażenie zawiera człony definiujące schemat strumienia oraz metody, które korzystają z węzłów tego schematu. Poniższe wyrażenie jest niepoprawne, ponieważ metoda `addPK` korzysta z węzła *name*, który jest jeszcze niezdefiniowany:

```
select tmp{ addPK(name), $I.name, sum = $I.place.x + $I.place.y,
            $I.place[true]}
…
```

Przyjmijmy, że w przyszłości funkcjonalność fabryki danych strumieni ulegnie rozszerzeniu. Na poziomie języka zapytań objawi się to poprzez udostępnienie nowych metod. Ujawnia się tutaj zaleta zastosowania elementów obiektowości, ponieważ nie ma już konieczności rozszerzania składni języka tak jak ma to miejsce dla DDL.

3.6.3 Klauzula from

Zestaw źródeł danych potrzebny do wyznaczenia zapytania jest tworzony w oparciu o analizę klauzul: **`where`**, **`having`**, **`groupby`**, dlatego nie ma konieczności deklarowania listy źródeł, na których będą realizowane obliczenia w klauzuli: **`from`**. Klauzula **`from`** służy definiowaniu strumieni pomocniczych na potrzebę bieżącego zapytania. Poniżej podano składnię klauzuli **`from`**:

```
<definicja_źródła> ::= (<specyfikacja_zapytania>)
                     | <nazwa_kolekcji_danych>{}
```

Aby uniknąć niejednoznaczności w interpretacji, składnia języka nakazuje umieszczanie definicji pod-zapytania wewnątrz nawiasów ‘()’. Poniżej podano przykład, gdzie pod-zapytanie tworzy strumień *tmp2*.

```
select tmp1{$tmp2.id, $tmp2.valF}
from (select tmp2{$I.id, $I.place[true]}
 where $I.place.x < 1000)
```

Bez użycia nawiasów treść zapytania jest niejednoznaczna, ponieważ klauzula *where* mogłaby być interpretowana: jako fragment pod-zapytania lub jako fragment głównego zapytania co ilustrują odpowiednio poniższe przypadki 1) i 2):

1)

```
select tmp1{$tmp2.id, $tmp2.valF}
from (select tmp2{$I.id, $I.place[true]}
 where $I.place.x < 1000)
```

2)

```
select tmp1{$tmp2.id, $tmp2.valF}
from (select tmp2{$I.id, $I.place[true]})
where $I.place.x < 1000
```

3.6.4 Klauzula where

Klauzula `where` składa się z wyrażeń definiujących warunki filtracji, łączenia strumieni oraz z elementu charakterystycznego dla języka StreamAPAS, którym jest Proxy. Element ten stanowi mechanizm pośrednictwa do fabryk danych. Nazwa ta w sposób luźny nawiązuje do idei działania wzorca projektowego Proxy.

```
<wyrażenie_where>::= [<lista_proxy>][<warunek_where>]
```

Aby użyć w zapytaniu tabelę historii należy zdefiniować algorytm ekstrakcji danych z fabryki danych poprzez Proxy. Parametry tej ekstrakcji są ustawiane poprzez wywołanie metod obiektu Proxy. Zastosowanie tutaj elementów języków obiektowych sprawia, że składnia języka StreamAPAS umożliwia obsługę wielu typów Proxy. Dla projektanta nowej fabryki danych, zdefiniowanie Proxy ogranicza się do utworzenia klasy w języku Java wraz z odpowiednio adnotowanymi metodami służącymi do jej konfiguracji. Po dołączeniu fabryki danych do systemu, dzięki mechanizmu refleksji adnotowane metody zostają udostępnione na poziomie języka zapytań. Na początku zostanie przedstawiona uproszczona składnia służąca do definiowania Proxy, która wystarcza do zaprezentowania zalet nowego podejścia. Pod koniec rozdziału przedstawiona zostanie pełna składnia Proxy, która daje szerokie możliwości obsługi złożonych fabryk danych.

Uproszczona składnia Proxy sprowadza się do definicji:

```
<lista_proxy>::=<proxy>, <proxy>,…
<proxy> ::= <nazwa_kolekcji_danych>{ <metoda(listaArgumentów)>,
metoda(listaArgumentów), …}
```

Na początku wyrażenia pojawia się `nazwa_kolekcji_danych` identyfikująca fabrykę danych obsługiwaną przez Proxy. Następnie przy użyciu nawiasów {…} przechodzi się do kontekstu obiektu Proxy. Wyrażenie wewnątrz nawiasów {…} zawiera wywołania metod konfigurujących. Składnia zezwala wywołać kilka metod oddzielonych przecinkiem. Poprawność semantyczna tworzonej konfiguracji jest weryfikowana przez obiekt Proxy w chwili wywoływania metod, jeżeli zostanie wykryty błąd użycia lub wartości argumentów, wtedy zgłaszany jest błąd kompilacji.

W języku przyjęto, że tabela historii jest wynikiem ekstrakcji danych z fabryki danych, która może reprezentować dane wielowymiarowe. Zgodnie z tą koncepcją okna należą do operatorów ekstrakcji danych zdefiniowane na osi czasu.

Dlatego operatory okien zaliczamy do elementów Proxy. W przypadku fabryki strumieni, Proxy jest konfigurowany poprzez metody:

rangeWindow(Long aCount)	–	tworzy okno liczebnościowe o rozmiarze *aCount*.
fixedWindow(Long aFrom, Long aPeriod)	–	tworzy okno kroczące o szerokości *aPeriod*[ms] gdzie pierwszy krok jest stawiany w chwili *aFrom*. Czas *aFrom* jest naliczany w milisekundach względem daty początkowej przypadającej na północ w styczniu 1, 1970 UTC.
slideWindow(Long aPeriod)	–	tworzy okno przesuwne o szerokości *aPeriod*[ms]

Implementacja fabryki strumieni została tak zrealizowana, że jeżeli ktoś dla jednego źródła danych przypisze kilka okien czasowych wtedy brane jest pod uwagę ostatnie w wyrażeniu.

Składnia wrażenia `warunek_where` jest zbliżona do składni języka SQL. Poniżej podano zestaw reguł definiujących dopuszczalne wyrażenia dla tego symbolu:

```
<expr> ::= <expr> AND <expr>
 | <expr> OR <expr>
 | <expr> XOR <expr>
 | <expr> <comparison_operator> <expr>
 | <expr> '-' <expr>
 | <expr> '+' <expr>
 | <expr> '*' <expr>
 | <expr> '/' <expr>
 | <prefixExpr>
 ;
<prefixExpr>::= <atom>
 | '-' <prefixExpr>
 | '!' <prefixExpr>
 ;
<atom>::= literal
 | <identyfikatorAtrybutu>
 | <method_call>
 | '(' expr ')'
 ;
<comparison_operator>::= <|<=|>|>=|!=|==
```

3.6.5 Klauzula group by i having

Działanie klauzul `group by` i `having` jest takie samo jak w języku SQL. Jedyna różnica polega na zastosowaniu składni opisanej przez symbol `expr`.

3.6.6 Operatory agregacji

W języku StreamAPAS operatory agregacji mogą być użyte w klauzulach *select* oraz *having*. Ponadto wywołań operacji agregacji nie można zagnieżdżać. Innymi słowy operandem agregacji nie może być wartość wyliczona przez inny agregat. W relacyjnej bazie danych zestaw funkcji agregujących jest ubogi, aby go rozszerzyć producenci baz danych wprowadzili rozszerzenia PL/pgSQL i PL/SQL. Od strony składniowej te rozwiązania definiują nowe operatory przy użyciu języka proceduralnego. W systemie StreamAPAS przyjęto, że operatory implementowane są w języku Java, a na poziomie języka zapytań są udostępniane jako metody statyczne podobnie jak operatory na zbiorach. To rozwiązanie pozwala z jednej strony uczynić implementację operatora agregacji na tyle wydajną na ile pozwala maszyna wirtualna Java, z drugiej strony osiągana jest unifikacja interfejsu języka zapytań.

Zestaw podstawowych operatorów agregacji jest zdefiniowany w klasie Agg. Obecnie system StreamAPAS udostępnia poniższy zestaw agregatów dla typów Integer, Long, Float i Double:

Agg.min(arg)

Agg.max(arg)

Agg.sum(arg)

Agg.count()

3.7 Bezpośrednie tworzenie obiektów Task

Składnia frazy Select-From nie obsługuje definiowania operatorów źródeł danych oraz ujść. Podobnie nie może posłużyć do zdefiniowanie indeksów. Aby obsłużyć brakującą funkcjonalność w języku StreamAPAS udostępniono bezpośrednie tworzenie obiektu Task. Obiekt ten definiuje skąd pobiera dane oraz jakie tworzy

fabryki danych. Do jego utworzenia zastosowano wzorzec metody fabrykującej. Składnia metody statycznej jest wystarczająco bogata do tworzenia takich obiektów, ponieważ umożliwia użycie jako argumentów zarówno typów podstawowych jak również fabryk danych. Drugą zaletą tego rozwiązania jest unifikacja systemu, projektant po zapoznaniu się w jaki sposób konstruowany jest pojedynczy operator może zastosować zdobytą wiedzę do budowy operatora ujścia lub indeksu. Korzystając z tego mechanizmu w systemie StreamAPAS zdefiniowano dwie klasy `benchmark` i `gui`. Przy ich użyciu można stworzyć szereg mikro-testów. Aby przedstawić ideę dołączania nowej funkcjonalności oraz korzystania z niej, poniżej podano specyfikację tych klas.

3.7.1 Klasa `benchmark`

Klasa ta zawiera metody fabrykujące tworzące źródła strumieni do budowy testów. W skład niej wchodzą metody: `twoStreams`, `StreamUniformRandom` i `StreamUniformRandomPKRandom`. Poniżej przedstawiono ich definicje.

```
twoStreams(String aName1, String aName2,
Integer aValFrom, Integer aValTo, Integer aPerMinute)
```

Metoda ta tworzy *Task* z dwoma źródłami strumieni nazwanymi odpowiednio *aName1* i *aName2*. Każde źródło generuje krotki ze stałą intensywnością wynoszącą *aPerMinute* krotek na minutę. Czas życia krotek wynosi 1ms. Schemat źródła składa się z czterech węzłów: *Id*, *valD*, *valF* i *valL* o typach odpowiednio: Long, Double, Float i Long. Atrybut *id* zawiera wartości zmiennej losowej o rozkładzie równomierny z zakresu od 0 do 999. Źródło definiuje także drugą zmienną losową o rozkładzie równomiernym i zakresie [0.0, 1.0). Wartości tej zmiennej losowej są zapisane w zmiennych *valD* i *valF*. Atrybut *valL* zawiera wartości tej zmiennej przeskalowane do zakresu od *aValFrom* do *aValTo*.

```
StreamUniformRandom(String aName1,
Integer aValFrom, Integer aValTo, Integer aPerMinute, Long aLife)
```

Metoda ta tworzy *Task* z pojedynczym źródłem strumienia nazwanym *aName*. Specyfikacja źródła jest identyczna jak w metodzie *twoStreams*. Argument *aLife* definiuje czas generowania krotek mierzony w [ms]. Jeżeli wartość *aLiftime = -1,* wtedy czas pracy źródła nie jest ograniczony.

```
StreamUniformRandomPKRandom(String aName1,
Integer aValFrom, Integer aValTo, Integer aPerMinute, Long aMaxPK)
```

Metoda ta tworzy *Task* z pojedynczym źródłem strumienia nazwanym *aName*. Specyfikacja źródła jest podobna jak w metodzie *twoStreams*. Różnica polega na tym, że kolejne wartości całkowite rozpoczynając od zera są wstawiane do atrybutu *id*. Argument *aMaxPK* definiuje maksymalną wartość *id*. Jeżeli wartość tego argumentu jest równa -1 wtedy źródło nie ma ograniczeń na liczbę wygenerowanych krotek.

3.7.2 Klasa `gui`

Klasa ta definiuje metody fabrykujące tworzące operatory ujść, które wyświetlają użytkownikowi zawartość strumieni oraz archiwizują wyniki. W klasie `gui` zdefiniowano metody:

```
show(DatasetI aStream)
```

Metoda ta tworzy *Task* zawierający pojedyncze ujście, które wyświetla na konsoli użytkownika zawartość strumienia *aStream*.

```
showAndRegisterLatency(DatasetI aStream, String aPath)
```

Metoda ta tworzy *Task* zawierający pojedyncze ujście wyświetlające na konsoli użytkownika zawartość strumienia *aStream*. Dodatkowo do pliku wskazanego przez *aPath* zapisane są wartości opóźnień krotek. Opóźnienie krotki jest różnicą pomiędzy bieżącym czasem systemowym a znacznikiem t_s krotki.

3.8 Obiektowa reprezentacja zapytania

W wyniku analizy składniowej tekst zapytania jest przekształcany do drzewa rozbioru. Następnie korzystając z reguł semantycznych tworzony jest plan produkcji w postaci DAG. Dodanie nowej funkcjonalności wiąże się od strony implementacyjnej z utworzeniem nowych typów oraz reguł składniowych, które będą tworzyły drzewo rozbioru oraz go przekształcały. W systemie StreamAPAS ten cel jest osiągany poprzez wprowadzenie elementów obiektowości do języka zapytań. Idea polega na tym, aby elementy składni postrzegać jako obiekty tworzące drzewo rozbioru.

Do implementacji systemu StreamAPAS użyto język obiektowy Java, dzięki temu obiektowość na poziomie języka zapytań można zamodelować przy użyciu obiektów z których zbudowany jest kompilator. Istota rozwiązania tkwi w takim zdefiniowaniu obiektów w języku Java, aby były one automatycznie mapowane na obiekty na poziomie języka zapytań. Problem ten rozwiązano przez mechanizmy adnotacji i refleksji. Należy zaznaczyć, że takie rozwiązanie ogranicza definiowanie nowej funkcjonalności do napisania kodu programu w języku Java, co jest dużym ułatwieniem dla użytkownika, ponieważ nie musi on poznawać dodatkowych języków deklarujących interfejs, tak jak to ma miejsce dla obiektów COM lub bibliotek dll.

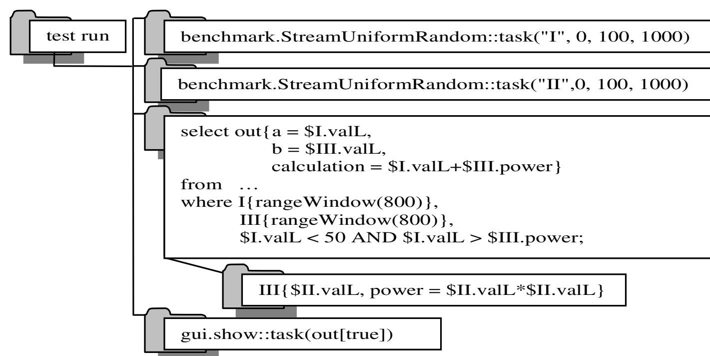
Przeanalizujmy poniższe zapytanie:

```
test run
begin
 benchmark.StreamUniformRandom::task("I", 0, 100, 1000)
 benchmark.StreamUniformRandom::task("II", 0, 100, 1000)

 select out{a = $I.valL,
 b = $III.valL,
 calculation = $I.valL+$III.power}
 from III{$II.valL, power = $II.valL*$II.valL}
 where I{rangeWindow(800)},
 III{rangeWindow(800)},
 $I.valL < 50 AND $I.valL > $III.power;

 gui.show::task(out[true])
end;
```

Na początku definiowany jest obiekt *Unit* o nazwie *test*. Następnie pojawia się lista definicji *Task*-ów które tworzą DAG. Wpierw budowane są generatory strumieni *I* oraz *II*. W tym celu wywołane są metody fabrykujące:

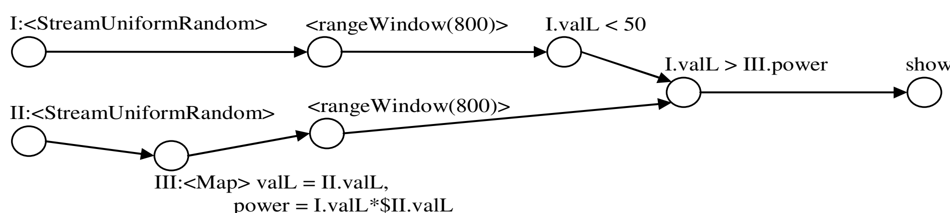
- *StreamUniformRandomPacket::task("I", 0, 100, 1000)*,
- *StreamUniformRandom::task("II", 0, 100, 1000)*.

Tworzą one dwa *Taski* składające się z źródeł danych o nazwach *I* i *II*. W następnym kroku, obiekt *Task* jest zdefiniowany przy użyciu składni Select-From. Wynik tego zapytania jest zapisany do fabryki strumieni o nazwie *out*. Na koniec zawartość strumienia *out* jest wyświetlona na konsoli przy użyciu operatora zawartego w *Task* zdefiniowanym funkcją fabrykującą *gui.show::task(out[true])*. Na rys. 3.3 przedstawiono uproszczoną postać powstałego drzewa rozbioru. Jego korzeniem jest obiekt typu *Unit*, pozostałe elementy to obiekty typu *Task*.



Rys. 3.3. Drzewo parsowania dla przykładowego zapytania

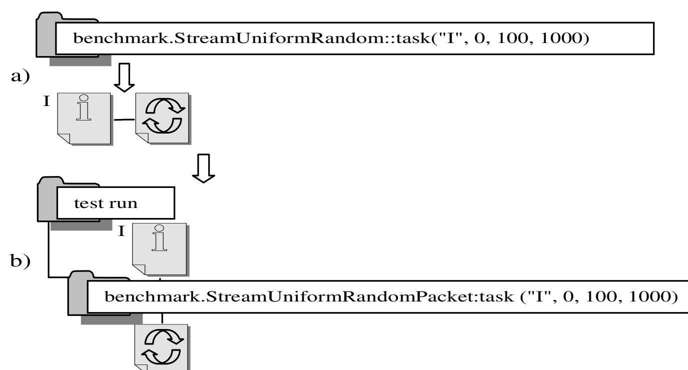
Po przeprowadzeniu analizy semantycznej dla rozważanego zapytania powstaje DAG zilustrowany na rys. 3.4.



Rys. 3.4. DAG dla przykładowego zapytania

W języku StreamAPAS obiekty *Unit* oraz *Task* mają ze sobą skojarzone przestrzenie nazw. Do zarządzania przestrzeniami nazw służy stos, u jego spodu leży przestrzeń globalna nazw. Przestrzeń globalna zawiera etykiety do obiektów *Unit* oraz do globalnie dostępnych fabryk danych. Gdy tworzony jest *Unit*, na stos odkładana jest jego przestrzeń nazw, którą nazywamy publiczną. Przestrzenie nazw dziedziczą wartości od swoich rodziców. Oznacza to, że jeżeli w bieżącej przestrzeni nazw brak poszukiwanej etykiety, wtedy przeglądana jest przestrzeń nazw leżąca niżej na stosie. Proces ten jest powtarzany do momentu zejścia na spód. Obiekty umieszczone w przestrzeni globalnej są ogólnodostępne dla wszystkich obiektów *Unit*. Do obiektów publicznych mają wyłącznie dostęp operacje zdefiniowane wewnątrz tego samego obiektu *Unit*. Aby przenieść obiekt z poziomu publicznego na poziom globalny należy użyć metody *setGlobal* obiektu *Unit*. Zastosowanie hierarchicznej przestrzeni nazw pozwala hermetycznie wydzielić samodzielne jednostki przetwarzania strumieniowego, odsuwa także problem związany z wyczerpaniem się popularnych nazw etykiet. To z kolei zmniejsza ryzyko popełniania błędów podczas tworzenia zapytań.

Implementacja obiektu typu *Task* definiuje jego zachowanie i reguły weryfikujące jego poprawność użycia oraz szczegółowe komunikaty błędów kompilacji. Realizacja tych funkcjonalności wiąże się z obsługą dwóch struktur. Pierwszą strukturą jest drzewo rozbioru, gdzie umieszczony jest węzeł reprezentujący użycie obiektu *Task* w tekście zapytania. W trakcie analizy semantycznej obiekt *Task* tworzy definicję procesu strumieniowego oraz schematy fabryk danych udostępniających wyniki, co ilustruje rys. 3.5 a). Następnie zbudowane elementy definiujące proces przetwarzania są umieszczane w DAG, z kolei schematy fabryk danych są wstawiane do drzewa rozbioru. W ten sposób użytkownik może rozszerzyć system o brakującą funkcjonalność jak i elementy analizy semantycznej.



Rys. 3.5. Drzewo parsowania dla przykładowego zapytania

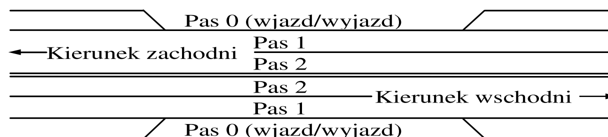
Przykładowo w wyniku analizy semantycznej metody fabrykującej *StreamUniformRandomPacket::task("I", 0, 100, 1000)*, w DAG powstaje fabryka strumieni o nazwie *I* oraz deklaracja jej schematu. Jeżeli wywołanie tej metody zawierałoby nie właściwe zakresy argumentów zgłoszony zostałby komunikat błędu. Po zweryfikowaniu poprawności semantycznej, deklaracje schematów są wstawiane do przestrzeni nazw, tak jak ilustruje to rys. 3.5. W wyniku tego strumienie *I*, *II* i *III* są umieszczone w publicznej przestrzeni nazw.

3.9 Zestaw testowy

Obecne popularnym rozszerzeniem strumieniowym SQL jest język CQL. Przy jego pomocy zdefiniowano kilka zestawów testowych takich jak: Linear Road Benchmark [6] i NEXMark [66,89]. Aby pokazać, że składnia języka StreamAPAS

pozwala wyrażać zapytania nie mniej złożone co produkty konkurencyjne, porównam go z językiem CQL. Do tego celu użyto test Linear Road Benchmark (LRB) zawierający wszystkie kluczowe operatory strumieniowej bazy danych.

Obecnie problem udrażniania ruchu ulicznego jest rozwiązywany poprzez rozbudowę infrastruktury oraz systemy sterujące sygnalizacją świetlną. Systemy te można nazwać biernymi, ponieważ nie mają one wpływu na wybór połączeń drogowych przez użytkownika. Nowatorskie podejście polega na kreowaniu zachowania użytkowników poprzez system naliczający mniejsze opłaty za korzystanie z dróg o słabszym obciążeniu. W tym rozwiązaniu konieczne jest analizowanie wielu danych napływających w sposób ciągły, dlatego problem ten stał się podstawą dla testu LRB, który służy porównywaniu wydajności strumieniowych baz danych. Test ten symuluje naliczanie opłat za korzystanie z autostrady. Ich wartość jest zwiększana, gdy samochód przemierza segment drogi zaklasyfikowany w danym momencie jako silnie obciążony. W konsekwencji ruch drogowy jest równoważony, ponieważ kierowcy są zachęcani do wyboru mniej zatłoczonych połączeń. System ten zakłada, że każdy samochód jest zaopatrzony w czujnik transmitujący szybkość oraz położenie. Informacje te są przekazywane poprzez system sensorowy do centralnego serwera aktualizującego współczynniki opłat oraz indywidualne opłaty. Następnie dane wynikowe są przekazywane z powrotem do każdego samochodu.



Rys. 3.6. Przykładowy segment autostrady w LRB

Na rysunku 3.6 przedstawiono elementy składowe autostrady w LRB. W systemie istnieje L pasów ruchu ponumerowanych od 0 do $L\text{-}1$. Każdy pas ruchu liczy 100mil i leży na kierunku wschód-zachód. Autostrada jest podzielona na 100 segmentów, ponadto na granicy segmentów znajdują się wjazdy i zjazdy. Każdy pojazd na autostradzie przekazuje swoje położenie oraz szybkość co 30 sekundy, przy czym położenie jest definiowane poprzez numer pasa ruchu, kierunek i dystans mierzony od lewego końca autostrady.

Pojazdy płacą opłatę, gdy przejeżdżają przez zatłoczony segment. Segment jest uznawany za zatłoczony, gdy średnia szybkość wszystkich pojazdów w przeciągu 5 minut jest mniejsza niż 40 MPH. Opłata jest naliczana zgodnie ze wzorem: $opłataBazowa * (liczbaPojazdów - 150)^2$.

System sensorowy po zebraniu pomiarów od samochodów tworzy strumień o schemacie *SegSpeedStr(vehiculeId, speed, segNo, dir, hwy)*, gdzie atrybut *vehiculeId* identyfikuje pojazd, *speed* jest jego szybkością naliczaną w MPH, *segNo* oznacza segment w którym się znajduje, *dir* wskazuje kierunek przejazdu i *hwy* oznacza numer pasa. Szczegółowy opis LRB jest dostępny w [6].

3.10 CQL

Konstruktorzy tego języka przyjęli, że będzie on obsługiwał operatory dostępne w relacyjnych bazach danych. Przetwarzanie strumieniowe osiągnięto poprzez dodanie operatorów konwertujących strumień danych w relację i na odwrót. W ten sposób CQL jest językiem SQL poszerzonym o składnię definiującą operatory konwersji strumienia w relację oraz operatory konwersji relacji w strumień. Podsumowując, język obsługuje trzy klasy operatorów: operatory relacyjne, operatory konwersji strumień w relację, operatory konwersji relacji w strumień. Konwersję strumienia w relację realizują trzy operatory:

- okno czasowe (range window), które zawiera wszystkie krotki z ostatnich *n* jednostek czasu,
- okno liczebnościowe (count window), które zawiera *n* ostatnich krotek; oraz
- okno partycjonujące będące połączeniem operatora grupowania oraz okna liczebnościowego. Jego działanie polega na przechowywaniu *n* ostatnich krotek dla każdej partycji, przy czym krotki należą do tej samej partycji, jeżeli wartości ich atrybutów partycji są identyczne.

Za każdym razem, gdy przybywa do strumienia nowa krotka wyliczana jest relacja przy użyciu operatorów okien, a następnie na tymczasowych relacjach są uruchamiane operatory z relacyjnej bazy danych. Aby wynik był strumieniem stosowane są operatory konwertujące relację w strumień. Wyróżniono trzy takie operatory:

- *Istream* –porównuje wynik bieżący oraz poprzedni i wstawia na wyjście krotki których nie było w wcześniejszej iteracji,
- *Dstream* –operator ten w przeciwieństwie do Istream wysyła na wyjście krotki usunięte względem wcześniejszej iteracji,
- *Rstream* –wysyła na wyjście wszystkie krotki należące do bieżącej relacji.

Zaletą składni CQL jest intuicyjność, co ułatwia wyrażanie zapytań osobom zaznajomionym z SQL. Składnia CQL zostanie zaprezentowana na przykładzie systemu LRB. Celem jest omówienie różnych elementów składniowych, dlatego prezentacja ograniczy się do czterech z sześciu zapytań wprowadzonych w LRB.

Zapytanie 1: Należy wyliczyć zbiór aktywnych pojazdów. Pojazd jest aktywny jeżeli przekazał informację o swoim położeniu w przeciągu ostatnich 30 sekund.

```
Select Istream(distinct vehiculeId)
From SeqSpeedStr[Range 30 Seconds]
```

Powyższe zapytanie ilustruje składnię operatorów należących do każdej z trzech klasy. Wpierw dane są pobierane ze strumienia `SeqSpeedStr` zdefiniowanego w punkcie 3.9. Następnie strumień jest konwertowany w relację przy użyciu okna czasowego obejmującego wszystkie krotki z przeciągu ostatnich 30 sekund. Później na otrzymanej relacji eliminowane są duplikaty aby wylistować zbiór samochodów bez duplikatów. Na koniec wynikowa relacja jest konwertowana do postaci strumienia przez operator *Istream*. W konsekwencji strumień wynikowy zawiera tylko krotki unikalne względem poprzedzającej iteracji.

Język CQL wprowadza kilka skrótów składniowych, aby zredukować potrzebę definiowania w każdym zapytaniu: operatora konwersji strumień-relacja, operacji na relacjach i operatora konwertującego relację w strumień. Jeżeli w zapytaniu zostanie pominięta deklaracja operatora konwersji strumień-relacja a semantyka zapytania wymaga relacji, kompilator wstawia do zapytania operator okna *S[range unbounded]*. Okno to ma nieskończony czas życia, w konsekwencji wynikowa relacja zawierająca wszystkie krotki strumienia. Kompilator wstawia również operator *Istream*, jeżeli w zapytaniu nie zdefiniowano operatora konwersji relacja-strumień a wynik zapytania jest monotoniczny. Stosując te reguły powyższe zapytanie redukuje się do postaci:

```
Select distinct vehiculeId
```

```
From SeqSpeedStr[Range 30 Seconds]
```

Inne podobieństwa pomiędzy CQL i SQL prezentują kolejne przykłady.

Zapytanie 2: Należy utworzyć relację, która zawiera bieżący segment dla każdego aktywnego pojazdu. Otrzymany strumień wynikowy jest dostępny pod nazwą *ActiveVehiculeSegRel*.

```
Select distinct L.vehiculeId, L.segNo, L.dir, L.hwy
From SegSpeedStr[Range 30 Seconds] as A,
 SegSeedStr[Partition by vehiculeId Row 1] as L
Where A.vehiculeId = L.vehiculeId
```

Okno czasowe w zapytaniu zostało użyte do zidentyfikowania aktywnych pojazdów w ostatnich 30 sekundach analogicznie jak w zapytaniu 1. Okno partycjonujące służy zidentyfikowaniu segmentu, w którym przebywał ostatnio każdy z pojazdów. Zaskakujące może być fakt, że nazwa strumienia wynikowego nie należy do składni frazy Select-From języka CQL. Informację tą podaje się jako parametr uruchomienia zapytania.

Ponadto złączenia [LEFT|RIGHT] JOIN oraz złączenia zewnętrzne nie są obsługiwane przez wiele logik strumieniowych. Przyczyną tego jest problem w definicji operatora negacji. W relacyjnych bazach danych przyjmuje się, że negacją zbioru A jest uzupełnienie tego zbioru w relacji. Strumień jest sekwencją krotek nieograniczoną w czasie. Z tej perspektywy negacja strumienia jest znana dopiero po zamknięciu strumienia. Sprawia to, że nie każda logika przetwarzania strumieni pozwala wprowadzić operator negacji, który podwójnie użyty prowadzi do otrzymania strumienia wejściowego.

Zapytanie 3: Należy utworzyć relację zawierająca wszystkie zatłoczone segmenty.

```
Select segNo, dir, hwy
From SegSpeedStr[Range 5 Minutes]
Group By segNo, dir, hwy
Having Avg(speed) < 40
```

Zgodnie z wprowadzeniem do LRB, segment autostrady jest zatłoczony, jeżeli średnia szybkość pojazdów w ostatnich 5 minutach w segmencie jest mniejsza niż 40 MPH. Aby wyznaczyć odpowiedź należy skorzystać z okna czasowego o rozmiarze

5 minut. Następnie przy użyciu frazy *having* przekazać na wyjście segmenty zatłoczone.

Zapytanie 4: Należy zliczyć pojazdy przebywające w segmentach.

```
Select segNo, dir, hwy, count(vehiculeId) as numVehicles
From ActiveVehiculeSegRel
Group by segNo, dir, hwy
```

Skorzystano tutaj ze strumienia *ActiveVehiculeSegRel* wyliczanego w zapytaniu 2. Zapytanie to ilustruje jak definiowany jest operator grupowania oraz agregacji.

3.11 Zapytania w StreamAPAS

W tej sekcji przedstawione zostaną powyższe zapytania zapisane w języku StreamAPAS.

Zapytanie 1:

```
select result{$SegSpeedStr.vehiculeId}
where SegSpeedStr{slideWindow(30 000)}

resultDist{Set.distinct(result{}, "vehiculeId")}
```

Na początku stworzono zapytanie realizujące okno przesuwne o rozmiarze 30 sekund na strumieniu *SegSpeedStr*. Otrzymany wynik cząstkowy jest podany na wejście operatora wyliczającego elementy unikalne. Finalnie strumień wynikowy zasila fabrykę strumieni *resulDist*. Przykład ten ilustruje, zastosowanie składni wywołania metody klasowej `Set.distinct` do zadeklarowania operatora wyliczającego elementy unikalne. Zauważmy, że tego typu wywołanie jest zbliżone do wzorca projektowego metody fabrykującej. Wywołanie metody `distinct` uruchamia pewną fabrykę, która po analizie zapytania wybiera implementację algorytmu realizującego zadeklarowane zadanie. W kolejnych zapytaniach ideę takiego wzorca metody fabrykującej użyto do deklaracji operatorów agregacji.

Zapytanie 2:

```
Select tmp{$L.vehiculeId, $L.segNo, $L.dir, $L.hwy}
From L{SegSpeedStr{}}
Where SegSpeedStr{slideWindow(30000)},L{partitionedWindow(1,"vehiculeId")},
 SegSpeedStr.vehiculeId == L.vehiculeId
```

```
ActiveVehicleSegRel{Set.distinct(tmp{}, "vehiculeId")}
```

Zauważmy, że operatory okien są deklarowane przy użyciu wywołania metod zdefiniowanych dla obiektu Proxy. Przykładowo metoda `slideWindow` jest zdefiniowana przez Proxy udostępniające strumień *SegSpeedStr*. Chcąc dodać do systemu inny operator okna, należy zdefiniować kolejną metodę dla obiektu Proxy. Z punktu widzenia składni, takie podejście standaryzuje mechanizm rozszerzania języka o nowe operatory okien. Ogólniej rzecz ujmując, takie rozwiązanie pozwala swobodnie opisać dowolną metodę ekstrakcji danych z fabryki danych. Kontynuując analizę powyższego zapytania. Zastosowano tutaj okno przesuwne o rozmiarze 30 sekund oraz okno partycjonujące o rozmiarze 1 na atrybucie *vehiculeId*. Na koniec, eliminowane są duplikaty w strumieniu *tmp* identyfikowane poprzez wartość atrybutu *vehiculeId*.

Poniższe zapytania 3 i 4 przybliżą składnię służącej do deklaracji atrybutów grupowania oraz zastosowanie metod fabrykujących w celu definiowania operatorów agregacji i warunków selekcji na wartościach agregatów.

Zapytanie 3:

```
select CongestedSegRel{$SegSpeedStr.segNo.segNo, $SegSpeedStr.segNo.dir,
                            $SegSpeedStr.segNo.hwy}
where SegSpeedStr{slideWindow(300 000)}
group by SegSpeedStr.segNo, SegSpeedStr.dir, SegSpeedStr.hwy
having Agg.sum($SegSpeedStr.speed) < 40
```

Zapytanie 4:

```
Select SegVolRel($ActiveVehiculeSegRel.segNo, $ActiveVehiculeSegRel.dir,
 $ActiveVehiculeSegRel.hwy, numVehicles = Agg.count())
group by ActiveVehiculeSegRel.segNo, ActiveVehiculeSegRel.dir,
 ActiveVehiculeSegRel.hwy
```

Podsumowując, składnia StreamAPAS jest podobna do języka CQL i SQL, co jest zaletą dla osób zaznajomionych z SQL. W odróżnieniu do rozwiązań konkurencyjnych, zaproponowany język wyróżnia się: drzewami atrybutów, fabrykami danych oraz zastosowaniem wzorca projektowego *metody fabrykującej*.

Dzięki wprowadzeniu drzewa atrybutów, argumentami oraz wynikiem funkcji mogą być złożone struktury danych. Dodanie takiej struktury do języka umożliwia tworzenie operatorów, które zwracają jako wynik kilka wartości. Własność ta pozwala uczynić kod zapytania zwięzłym i czytelniejszym, ponieważ nie ma konieczności stosowania wielu funkcji. Ponadto wyliczenie kilku wartości przez

jedną funkcję pozwala zastosować wydajniejsze algorytmy, co ma duże znaczenie w przypadku operatorów agregacji.

W tej sekcji przedstawiono również zastosowanie fabryk danych. Wprowadzając ten mechanizm do języka zapytań usystematyzowano proces dołączania nowych okien do języka zapytań. Cecha ta jest istotna, ponieważ realizacja niektórych zapytań wymaga zastosowania nietypowych okien. Przykładem jest zapytanie 2, które wymagało zdefiniowania okna partycjonującego.

Metoda wytwórcza lub fabrykująca pozwala tworzyć delegatów, którzy będą obsługiwać poszczególne operacje, przykładowo operacja agregacji lub specjalizowane okno czasowe. W przypadku zapytań potrzebny jest czasem kontekst wywołania metody. Z tej perspektywy istnieje pewna swoboda w rozumieniu wzorca projektowego metody fabrykującej. Włączenie tego wzorca do języka StreamAPAS pozwoliło zunifikować dołączanie nowych operatorów na poziomie języka zapytań. Dzięki temu nie ma potrzeby zmiany jego składni w celu rozszerzenia funkcjonalności.

3.12 Budowa kompilatora

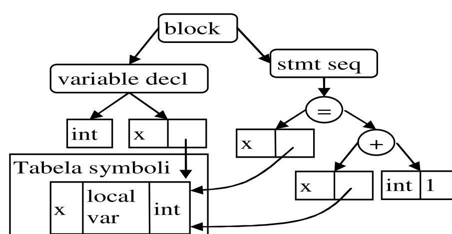
Kompilator języka StreamAPAS został zbudowany korzystając z generatora parserów BYAC/Java oraz analizatora leksykalnego JFlex. Generator parserów jest programem napisanym w Java, którego pierwowzorem był generator parserów YACC v1.8 stworzonym w Berkeley. Zbudowanie parsera oraz leksera polega na zdefiniowaniu plików analizy leksykalnej oraz pliku z gramatyką języka. Następnie przy użyciu JFlex i BYAC/Java generowany jest kod programu w Java będący lekserem i parserem. Zaletą wybranych narzędzi jest to, że składnia plików konfiguracyjnych jest zgodna ze składnią wprowadzoną w narzędziach YACC i flex, które wyznaczają standard w tej dziedzinie. Obecnie dużą popularność zyskuje generator ANTLR, który oferuje zestaw dodatkowych narzędzi służących do budowy i testowania poprawności gramatyk. W szczególności narzędzie to wspiera programistę podczas tworzenia systemu odpowiedzialnego za zgłaszanie oraz obsługę błędów kompilacji. Zaleta ta skłania do wykorzystania tego narzędzia w przyszłych pracach badawczych.

3.13 Tabela symboli

Wprowadzając do języka zapytań elementy składni języków obiektowych oraz stosując język Java do implementacji kompilatora stworzono nowatorskie rozwiązanie, które pozwala przy użyciu mechanizmu refleksji uczynić język zapytań otwartym na dodawanie nowej funkcjonalności. Kluczowy jest tutaj sposób definiowania typów. Korzystanie z mechanizmu refleksji wiąże się z przyjęciem do języka zapytań typów języka Java. Z drugiej strony zestaw typów języka zapytań jest bogatszy. W tej sekcji przedstawiona zostanie architektura tabeli symboli, która łączy oba systemy typów. Na początku zostanie przedstawiona podstawowa architektura symboli na przykładzie drzewa atrybutów. Następnie zdefiniowane zostaną zasady mapowania obiektowości języka zapytań StreamAPAS na język Java. Na koniec omówiona zostanie architektura przestrzeni nazw, która mapuje literały języka StreamAPAS na elementy tabeli symboli.

W kompilatorach języków programowania wyróżniamy: drzewo rozbioru składniowego oraz tabele symboli. W wyniku operacji parsowania powstaje drzewo rozbioru składniowego. Tabela symboli stanowi centralne repozytorium typów i zmiennych, które jest użytkowane w kolejnych fazach kompilacji. Taka konstrukcja kompilatorów wprowadza podział na struktury danych oraz operacje. Struktury danych są przechowywane w tabeli symboli z kolei operacje są zdefiniowane w drzewie rozbioru zapytania oraz strukturach utworzonych w trakcie analizy semantycznej. Przykładowo, po przeprowadzeniu analizy składniowej poniższego fragmentu programu w C, powstaje tabela symboli i drzewo rozbioru składniowego przedstawione na rys. 3.7.

```
…
int x;
x = x + 1;
…
```



Rys. 3.7. Tabela symboli i drzewo rozbioru składniowego dla fragmentu programu w C

Wyrażenie: `x = x + 1` zostało zaprezentowane poprzez drzewo rozbioru, którego węzły reprezentują operacje, a krawędzie oznaczają przepływ wartości wygenerowanych przez dołączone operatory. Zauważmy, że prawym operandem dla operatora przypisania „=” nie jest operator sumy ale wartość jaką zwraca ten operator. Etykieta `x` reprezentuje symbol definiujący zmienną lokalną o typie `int`. Aby zapisać informację, że zmienna `x` jest lokalna, symbol definiujący zmienną ma ustawiony modyfikator na wartość: `local var`. Jeżeli zestaw typów w języku jest zaszyty na sztywno, wtedy typ zmiennej jest zdefiniowany poprzez prostą listę wartości wyliczeniowych. W języku C zestaw typów jest rozszerzalny. Przykładowo użytkownik może zdefiniować strukturę. Aby wyrazić taki element tabela symboli przyjmuje, że typ jest również symbolem. Po wprowadzeniu takiej modyfikacji zmienna jest definiowana w tabeli symboli poprzez: nazwę, modyfikator oraz symbol reprezentujący typ. Przyjrzyjmy się teraz definicji typu. W językach proceduralnych stan oraz zachowanie są rozpatrywane oddzielnie, skutkuje to tym, że typ definiuje tylko sposób przechowywania stanu. W językach obiektowych stan oraz zachowanie są ze sobą połączone w strukturach nazwanych obiektami. Opis takiego obiektu jest realizowany poprzez typ zwany klasą. W językach programowania wprowadzono także pojęcie meta-typu. Tak jak zmienna jest realizacją pewnego typu, typ jest realizacją pewnego meta-typu. Przykładem meta-typów są typy generyczne w języku Java lub klasy szablonowe w C++. W współczesnych językach programowania do zestawu omówionych paradygmatów programowania dołączono mechanizm refleksji. Realizacja tego mechanizmu w języku Java umożliwia odczyt definicji typów jako elementy tabeli symboli w trakcie kompilacji. W przypadku języka Java, typy są zdefiniowane poprzez obiekty typu Class. Oznacza to, że definicja symbolu również może być argumentem funkcji wywołanej przy użyciu mechanizmu refleksji. Własność ta ma ogromne znaczenie, ponieważ pozwala na zdefiniowanie funkcji w docelowym języku zapytań reprezentujących operacje na różnych poziomach abstrakcji.

3.13.1 **Drzewo atrybutów**

Celem postawionym podczas projektowania tabeli symboli dla kompilatora języka StreamAPAS jest zdefiniowanie symboli w taki sposób, aby umożliwić

swobodne korzystanie z mechanizmu refleksji zaimplementowanego w języku Java. Problem polega na tym, że w języku Java architektura symboli jest zamknięta na modyfikacje; oznacza to że nie można rozszerzać jej o elementy konieczne do przeprowadzenia analizy semantycznej lub zdefiniowania drzewa atrybutów. Dlatego tabela symboli dla języka StreamAPAS łączy dwie wersje struktur opisujące ten sam typ w zależności czy analiza jest na poziomie języka StreamAPAS, czy na poziomie Java. Rozwiązanie takie jest konieczne, aby wszystkie symbole mogły być obsługiwane przy użyciu mechanizmu refleksji. Przyjęto, że każdy symbol implementuje interfejs `SymbolI`:

```
public interface SymbolI {
 public String getName();
 public Class getSymbolClass();
 public SymbolI getParent();
 public void toXML(TransformerHandler aTHd) throws SAXException;
}
```

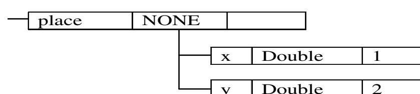
Interfejs ten definiuje symbol jako strukturę, która posiada nazwę udostępnianą przez metodę `getName` oraz typ symbolu dla mechanizmu refleksji udostępniany metodą `getSymbolClass`. Dodatkowo metoda `toXML` zapisuje bieżący symbol do postaci XML oraz metoda `getParent` zwraca symbol w którym bieżący symbol został zdefiniowany.

W języku StreamAPAS występują także typy złożone. Przykładem nich jest drzewo atrybutów, gdzie każdy z węzłów posiada podwęzły. Innym przykładem jest element *Unit*, który składa się z *Task*-ów. Aby zdefiniować takie struktury drzewiaste wprowadzono dodatkowy symbol pełniący rolę kontenera symboli. Symbol taki jest reprezentowany przez interfejs `ScopeI`:

```
public interface ScopeI extends SymbolI{
 public boolean isCatalogScope();
 public SymbolI findSymbol(IdentifierName aIdentifierName, int aDict);
 public SymbolI addSymbol(SymbolI aSym, int aDict) throws ParserException;
 public LinkedList<SymbolI> getSymbols(int aDict);
}
```

Powyższa definicja umożliwia dodawanie (`addSymbol`), wyszukiwanie (`findSymbol`) oraz listowanie wszystkich pod-symboli (`findSymbol`). Aby można było swobodnie przemieszczać się po definicji typów złożonych, interfejs `SymbolI` zawiera metodę `getParent`. Zwraca ona referencję do symbolu w którym został zdefiniowany bieżący symbol. Pozostająca metoda `isCatalogScope` zostanie omówiona przy opisie

przestrzeni nazw. Prześledźmy teraz konstrukcję typów złożonych na przykładzie fragmentu drzewa atrybutów z rys. 3.8.

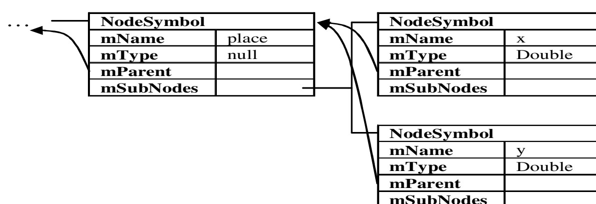


Rys. 3.8. Fragment drzewa atrybutów

Przyjmijmy, że do skonstruowania węzła drzewa atrybutów użyto interfejs `NodeSymbolI` zdefiniowany jako:

```
public interface NodeSymbolI extends ScopeI {
 …
}
```

Interfejs ten jest implementowany przez klasę `NodeSymbol`. Korzystając z tego symbolu rozważane drzewo jest opisane przez strukturę danych przedstawioną na rys. 3.9. Zwróćmy uwagę, że węzeł `NodeSymbol` może być interpretowany na dwa sposoby. Z jednej strony reprezentuje on wartość zapisaną w węźle drzewa atrybutów. Z drugiej strony reprezentuje on poddrzewo. Zgodnie z przyjętym założeniem typ symbolu dla mechanizmu refleksji jest zdefiniowany poprzez metodę `getSymbolClass`. Oznacza to, że zaproponowana architektura na rys. 3.9 nie posiada symbolu reprezentującego gałąź drzewa atrybutów. Jeżeli jako operandu użyjemy symbolu o nazwie *place*, to oznacza on tylko wartość węzła a nie gałąź. Reprezentacja taka nie pozwala również na przypisanie wartości stałej węzłowi drzewa atrybutów.



Rys. 3.9. Początkowa budowa symbolu

Problem ten rozwiązuje zastosowanie interfejsu `VarSymbolI`, który reprezentuje deklarację zmiennej.

```
public interface VarSymbolI extends SymbolI {
 public int DEFINITIONCLOSED = 1;
 public int FURTHERRESOLVE = 8;

 public void setSymbolClass(Class aType);

 public void setVal(Object aVal);
 public Object getVal();

 //---------------------------------------
 public boolean isDefinitionOpen();

 public void setModificator(int aModif);
 public void delModificator(int aModif);
 public int getModificator();
}
```

Interfejs `VarSymbolI` udostępnia informację o nazwie zmiennej, jej typ oraz modyfikator dostępności. Modyfikatory dostępności służą kompilatorowi do zapisania dodatkowych informacji koniecznych do przeprowadzenia analizy semantycznej. Wyróżniamy dwa modyfikatory:

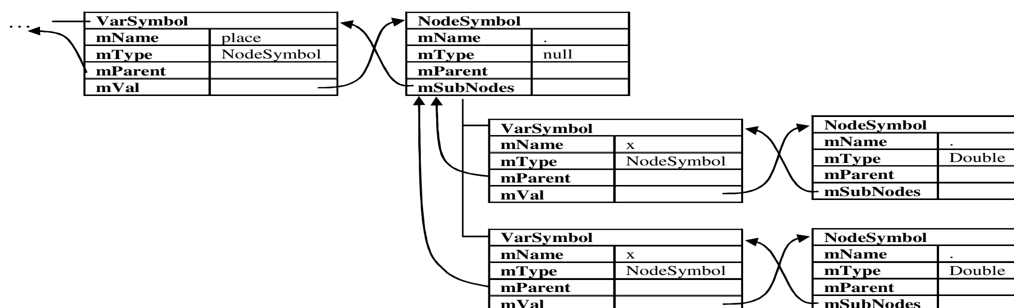
- FURTHERRESOLVE – oznacza on, że definicja typu w bieżącym momencie nie jest kompletna i wszelkie analizy semantyczne korzystające z tego typu muszą zostać odłożone do momentu skompletowania definicji.
- DEFINITIONCLOSED – informuje, że definicja typu jest zamknięta. Oznacza to, że modyfikuja typu z poziomu języka zapytań jest niedostępna, a wszelkie jej próby będą zgłaszane komunikatem błędu.

Interfejs ten jest zaimplementowany przez klasę `VarSymbol`.

Powróćmy teraz do rozważanego przykładu drzewa atrybutów. Aby usunąć zasygnalizowane wady, definicję `NodeSymbolI` rozszerzono o interfejs `VarSymbolI`.

```
public interface NodeSymbolI extends ScopeI, VarSymbolI {
 …
}
```

Korzystając z klasy `VarSymbol` oraz nowej definicji interfejsu `NodeSymbolI` drzewo atrybutów z rys. 3.8 zostało zdefiniowane przez strukturę przedstawioną na rys. 3.10.



Rys. 3.10. Budowa symbolu złożonego

Istotną słabością pierwszej struktury tabeli symboli był brak symbolu wskazującego na gałąź drzewa atrybutów. W nowej architekturze przyjęto zasadę, że każdy symbol jest deklarowany w tabeli symboli poprzez `VarSymbol`. W ten sposób `VarSymbol` reprezentuje gałąź drzewa atrybutów. Z kolei wartość węzła jest reprezentowana przez `NodeSymbol`. Istnieje także możliwość zapisania wartości statycznej dzięki rozszerzeniu interfejsu `NodeSymbolI` o interfejs `VarSymbolI`.

Znając składnię wyrażeń stosowaną do obsługi drzewa atrybutów, która została przedstawiona w sekcji 3.3 oraz dysponując wiedzą jak ta struktura jest zdefiniowana w tabeli symboli; zweryfikujmy działanie obu mechanizmów razem. Przyjmijmy, że dysponujemy strumieniem *s* zdefiniowanym:

```
…
select s{slot = 1, position[x=2, y=3.0],…}
…
```

Następnie strumień ten jest podany jako argument metody `Foo`, abstrahujemy w tym miejscu czy jest to wywołanie metody obiektowej czy klasowej:

```
Foo( s.position.x )
```

Pytanie brzmi jakiego typu jest argument funkcji. Metodę `getSymbolClass()` można wywołać zarówno na obiekcie reprezentującym zmienną (`ValueSymbol`) jak i na obiekcie reprezentującym wartość (`NodeSymbol`), w związku powyższym istnieją dwie potencjalne interpretacje:

- *s.position.x* oznacza zmienną o typie `Integer` zgodnie z definicją: *x = 2*. Zatem poszukujemy funkcji `Foo(Integer …)`,
- *s.position.x* oznacza obiekt klasy `NodeSymbol`. Zatem poszukujemy funkcji `Foo(NodeSymbol …)`.

Oba podejścia mają swoje odrębne zastosowanie:

- Pierwsza konwencja jest wygodna, gdy chcielibyśmy wywołać zwykłą funkcję arytmetyczną.
- Druga konwencja jest przydatna, gdy funkcja potrzebuje nie tylko informacji o typie zmiennej (dostępnym po wywołaniu metody getSymbolClass()) ale również jego dokładnym opisie (położeniu w systemie, strumień do którego należy). Taka interpretacja jest wymagana, gdy definiujemy operator przetwarzający strumień a nie tylko pojedynczy argument krotki. Przykładem są tutaj operatory agregacji lub operatory na zbiorach.

W języku StreamAPAS istnieją obie metody dostępu do zmiennych. Interpretacja etykiety zmiennej, jako wskaźnika do wartości zmiennej jest obsługiwany przez podstawową składnię. Obiekt typu `NodeSymbol` jest przekazywany do funkcji, gdy korzysta się z gałęzi drzewa atrybutów. Przykładowo zmienna `s.position[true]` reprezentuje pod-drzewo o korzeniu w węźle `position`.

3.13.2 Translacja obiektowości StreamAPAS na język Java

Przejdźmy teraz do zagadnienia, jakimi zasadami się kierować podczas zapisu zapytania do tabeli symboli. W tym celu przeanalizujmy składnię języka zapytań poprzez analogię do obiektowego języka programowania. W wyniku kompilacji zapytania otrzymujemy zestaw algorytmów, które wyliczają odpowiedź na zapytanie. Realizacją tych algorytmów jest proces przetwarzania, który generuje ciąg wyników. Posługując się zestawem pojęć języka obiektowego, sieć operatorów to obiekty zdefiniowane przez typy, które reprezentują zestawy algorytmów. Ten zestaw algorytmów może zostać uściślony przez frazę Select-From. Innymi słowy ta fraza definiuje pewien typ operacji przetwarzania. Przeprowadzone wnioskowanie prowadzi do stwierdzenia, że składnia języka jest meta-typem, przykładem tego jest fraza Select-From. Zapytanie zdefiniowane przy użyciu tej składni jest typem. Z kolei instancją tego typu jest obiekt realizujący przetwarzanie. Powyższa analiza była źródłem dla założenia, że meta-typ języka zapytań będzie reprezentowany bezpośrednio jako klasy języka Java. Wtedy typ w języku jest obiektem w języku Java. Następnie obiekt ten służy do zdefiniowania fabryki, która zbuduje proces przetwarzania danych, czyli obiekt w języku zapytań.

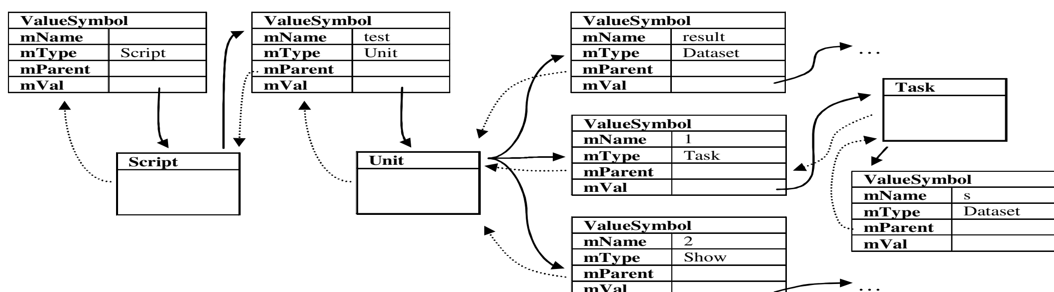
Zdefiniowane zasady mapowania typów języka zapytań na typy języka Java w sposób spójny opisują jakie elementy są odpowiedzialne za rozszerzenie składni języka, a jakie za implementację operatorów fizycznych. Prześledźmy zastosowanie tych zasad na poniższym zapytaniu.

```
test run
begin
 select result{id = $s.slot, $s.position[true]}
 from s{slot = 1, position[x=2, y=3.0]}
 where s.x < 4;

 gui.show::task(result[true])
end;
```

Omawiane zapytanie zawiera podstawowe elementy składni czyli: frazę Select-From, pod-zapytanie oraz obiekt `Task` zdefiniowany przy użyciu metody fabrykującej. Ze względu na rozmiar, analizę ograniczono do czterech meta-typów, są nimi:

- *Script* – reprezentuje tekst kompilowanego zapytania, które może składać się z kilku elementów *Unit*,
- *Unit* – definiuje pojedyncze zadanie uruchamiane w systemie,
- *Task* – reprezentuje jednostkę przy użyciu której budowany jest DAG,
- *Dataset* – definiuje schemat strumienia.

Przyjęte zasady mapowania mówią, że meta-typy są reprezentowane w tabeli symboli jako klasy. Meta-typy są również symbolami, co oznacza że implementują one interfejs `SymboI`. Jak również zachodzi zasada korzystania z symbolu `VarSymbol` w celu deklarowania typów. Stosują te reguły dla rozważanego przykładu otrzymujemy tabelę symboli na rys. 3.11.



Rys. 3.11. Tabela symboli dla analizowanego zapytania

Zauważmy, że w zaproponowanej architekturze symbolem jest zarówno schemat krotki, jak również zapytanie. Dzięki temu każdy symbol może być

operandem, w konsekwencji funkcje zdefiniowane w języku StreamAPAS pełnią szerokie spektrum ról, począwszy od zdefiniowania operacji arytmetycznych skończywszy na analizie semantycznej zapytania. Stosując metody fabrykujące użytkownik potrafi dołączyć własne symbole oraz funkcje do systemu, oznacza to że możliwe jest rozszerzenie funkcjonalności analizatora semantycznego a nie tylko zestawu operatorów. Takie rozwiązanie jest szczególnie atrakcyjne w systemach analitycznych, gdzie stale powstają nowe struktury indeksujące. Dzięki niemu nie ma konieczności czekania na stworzenie nowej wersji języka zapytań i analizatora semantycznego, wystarczy tylko zdefiniować meta-typy, które poprzez mechanizm refleksji są udostępniane z poziomu języka zapytań.

W tabeli symboli definicja typu jest reprezentowana jako zmienna. Cześć tych zmiennych jest generowana automatycznie w sposób niewidoczny dla użytkownika. Każda z tych zmiennych musi posiadać unikalną nazwę, dlatego analizator semantyczny z symbolem implementującym interfejs `ScopeI` kojarzy sekwencję zainicjalizowaną wstępnie na wartość 1. Jeżeli jest definiowana zmienna, która nie posiada nadanej nazwy przez użytkownika, pobierana jest kolejna wartość sekwencji z bieżącego kontekstu. Na rysunku 3.11 przykładami zastosowania takiego generatora nazw są nazwy zmiennych reprezentujące symbole `Task`.

3.13.3 **Przestrzenie nazw**

Ostatnim elementem związanym z obsługą tabeli symboli jest organizacja przestrzeni nazw. Każdy element kodu zapytania posiada swojego reprezentanta w postaci symbolu. W zbudowanym kompilatorze symbole są obiektami, czyli składają się z atrybutów oraz metod. Dodatkowo metody te można wywołać z poziomu języka zapytań przy użyciu mechanizmu refleksji. Przestrzeń nazw jest potrzebna kompilatorowi do powiązania literału użytego w języku zapytań z symbolem w którym go zdefiniowano. Wyróżniamy dwie polityki obsługi przestrzeni nazw. Pierwsza polega na przeładowywaniu przestrzeni nazw gdy przechodzimy z jednego symbolu do drugiego. Druga polityka wprowadza hierarchię pomiędzy przestrzenie nazw. Przyjmijmy że istnieją dwie przestrzenie nazw *A* i *B*, które posiadają zdefiniowaną metodę o nazwie *foo*. Jeżeli przestrzeń *B* jest zagnieżdżona w *A*, wtedy metoda *foo* dla przestrzeni *A* jest przesłonięta przez

implementację w przestrzeni *B*. Obsługa przestrzeni nazw korzysta ze stosu symboli. Jeżeli w kodzie programu odwołujemy się do symbolu, następuje otworzenie jego przestrzeni nazw oraz wstawienie go na szczyt stosu. Jeżeli wychodzimy z bieżącej przestrzeni nazw, ze szczytu stosu jest zdejmowany symbol. W konsekwencji stos zawiera symbole, które mają obecnie otwarte przestrzenie nazw. W kompilatorze języka StreamAPAS występują przestrzenie nazw, które dziedziczą oraz które przesłaniają. W celu rozróżnienia przestrzeni nazw, symbole implementujące interfejs `ScopeI` posiadają metodę `isCatalogScope`, która zwraca wartość prawda jeżeli symbol rozszerza hierarchię przestrzeni nazw. Ten typ symboli będzie nazywany katalogiem. Poniżej podano przykładową zawartość stosu. Nowe elementy odkładane są na prawą stronę listy, litera *S* reprezentuje zwykły symbol, a litera *C* katalog.

S_1, C_2, C_3, S_4, S_5, S_6, C_7, S_8, S_9

Jeżeli zadaniem jest wyszukanie zmiennej *foo*, analizator semantyczny na początku sprawdza czy nie jest ona zdefiniowana w symbolu S_9. Jeżeli jej brak, wtedy poszukiwany jest katalog leżący najbliżej szczytu stosu. Dla rozpatrywanego przykładu jest nim symbol C_7. Jeżeli brak jej w C_7, wtedy analizator przechodzi do kolejnego katalogu i powtarza poszukiwanie.

W zbudowanym kompilatorze katalogami są elementy: *Unit*, *Task* oraz korzeń tabeli symboli. Dzięki takiej organizacji, definicje fabryk danych można umieścić na różnych poziomach w tabeli symboli, co redukuje ryzyko wyczerpania krótkich nazw w bieżącej przestrzeni nazw. Przykładem zwykłych symboli są węzły drzewa atrybutów. Tutaj przejście z jednego węzła do następnego wiąże się z przezładowaniem dostępnych etykiet.

3.14 Funkcje

Wywołanie funkcji jest opisane składnią:

```
[<pełna kwalifikowana nazwa klasy>.]<nazwa metody>[::<modyfikator metody>]
 ([<lista argumentów>])
```

Gdzie:

`<pełna kwalifikowana nazwa klasy>` – jeżeli jest wywoływana metoda klasowa, wtedy konieczne jest podanie pełnej ścieżki do klasy zgodnie z konwencją zdefiniowaną w języku Java.

`<nazwa metody>` –jest identyfikatorem metody. Dopuszczalna składnia nazwy jest podana w dodatku A.

`<modyfikator metody>` – jest listą literałów oddzielonych przecinkiem. Modyfikatory wywołania metody informują o jej specjalnym użyciu. Obecnie zdefiniowano modyfikator `task` sygnalizujący, że w wyniku wywołania metody powstaje obiekt typu Task.

`<lista argumentów>` – jest listą operandów oddzielonych znakiem przecinka.

Wyróżniamy metody obiektowe oraz metody klasowe. Metody klasowe związane są z klasą w której są zdefiniowane. Wywołania tego typu metod zawierają pełną kwalifikowaną nazwę klasy. Przykładami wywołań dla tej grupy metod są: `gui.`**`show`**`::task(result[true])`, `Agg.`**`min`**`($I.valF)` i `Set.`**`distinct`**`(I{}, "val")`. Metody obiektowe są związane symbolami. Tutaj wyszukiwanie symboli jest realizowane przez algorytm obsługujący przestrzenie nazw. Przykładami użycia tych metod są wywołania: `select out{…,` **`addPK`**`(id)}`, `where II{`**`slideWindow`**`(800)}` i **`setGlobal`**`(out{})`.

Zauważmy, że wyszukiwanie metody obiektowej lub klasowej jest podobne do składni języka Java. Cecha ta sprawia, że w tym zakresie można w pełni skorzystać z implementacji refleksji języka Java. Niestety definicja metod w języku Java jest zbyt uboga aby analizator semantyczny języka StreamAPAS mógł tylko na niej bazować. Do brakujących elementów definicji zalicza się:

1) Przy użyciu mechanizmu refleksji mamy dostęp do wszystkich metod zdefiniowanych w klasie. Jednak nie wszystkie metody zdefiniowane dla symboli, powinny być udostępniane z poziomu języka zapytań. W definicji metody w języku Java brak informacji o tym czy jest ona przeznaczona do udostępniania z poziomu języka zapytań; istnieją jedynie modyfikatory: `private`, `public`, `protected`, `default` definiujące widoczność metod na poziomie języka programowania.
2) Tabela symboli zawiera meta-typy oraz typy. Przykładowo meta-typem jest interfejs `VarSymbolI` służący do deklaracji zmiennej. Z kolei typ zmiennej jest

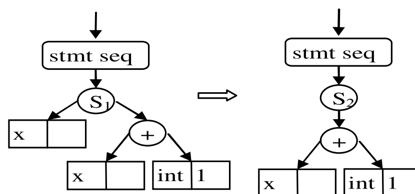
definiowany przez metodę `getSymbolClass`. Jeżeli metoda realizuje operację arytmetyczną, wtedy typem operandu jest typ zmiennej. Jeżeli metoda implementuje fragment analizy semantycznej, wtedy wymagany jest dostęp do deklaracji zmiennej, czyli symbolu definiującego zmienną. Metody w języku Java nie rozróżniają operacji na poziomie meta-typów i typów, w związku z tym konieczne jest rozszerzenie obsługi mechanizmu refleksji.

3) Przyjrzyjmy się teraz typowi zwracanemu przez metodę w języku StreamAPAS. W kompilatorze typ zwracany przez metodę jest opisany przez strukturę symboli podczas gdy typem wynikowym metody w języku Java jest klasa. Jeżeli zdefiniujemy w języku Java metodę arytmetyczną wyliczającą sinus dla argumentu typu double, jej deklaracja będzie miała postać:

```
double sin(double aVal) {….}
```

   Pytanie pojawia się jak zdefiniować w języku Java metodę, która zwraca jako wynik drzewo atrybutów.

4) Należy zauważyć, że w języku StreamAPAS metody realizują operacje na dwóch poziomach abstrakcji. Metody przekształcają analizę semantyczną, jak również definiują proces przetwarzania danych. W wyniku analizy składniowej powstaje drzewo rozbioru składniowego, składające się z węzłów reprezentujących operacje. Na rys. 3.12 przedstawiono przykładowe drzewo rozbioru składniowego składające się z listy zdań reprezentowanych przez węzeł `stmt seq`. Jednym ze zdań jest operacja zdefiniowana przez węzeł S_1. Jeżeli węzeł S_1 jest metodą przekształcającą analizę semantyczną, wtedy w wyniku jej uruchomienia powstaje nowy węzeł S_2 w miejscu S_1. Zauważmy, że drzewo rozbioru składniowego dołączone do węzła S_1 jest traktowane jako zmienna, którą można przekształcać. Jeżeli węzeł S_1 reprezentuje metodę przetwarzania, wtedy w wyniku jej uruchomienia powstaje wartość, która jest przekazywana operatorowi `stmt seq`. Metody w języku Java nie operują na poziomie analizy semantycznej, oznacza to że należy rozszerzyć definicję metody o informację na jakim poziomie abstrakcji działa metoda.



Rys. 3.12. Przekształcanie drzewa rozbioru składniowego

Aby usunąć powyższe ograniczenia zastosowano mechanizm adnotacji. Język Java pozwala rozszerzyć definicje klasy o dodatkowe komentarze dołączane do deklaracji klas, metod, zmiennych klasowych i obiektowych. Komentarz ten nazywano adnotacją, a odczyt jego wartości jest zbliżony do obsługi interfejsu. Istotny z punktu widzenia rozwijanego kompilatora jest fakt, że adnotacja pozwala zdefiniować własne modyfikatory oraz udekorować nimi metody oraz definicje klas. Adnotacja jest elementem definicji języka Java, oznacza to że poprawność wykorzystania tej techniki jest sprawdzana przez kompilator języka Java, co jest dodatkowym atutem. Ponadto korzystanie z tego rozwiązania jest poręczne, ponieważ nie ma konieczności tworzenia dodatkowych plików konfiguracyjnych.

Usunięcie ograniczenia pierwszego zrealizowano poprzez system adnotacji. Na poziomie klasy zdefiniowano adnotację blokującą dostęp do wszystkich metod składowych; a na poziomie metod zdefiniowano adnotacje oznaczającą ukrycie lub uwidocznienie metody. Użytkownik może zdefiniować dostępność metod na dwa sposoby. Pierwsze podejście polega na ukryciu wszystkich metod dodając adnotację `@CModificator(mode = CModificator.HIDEMEMBERS)` na poziomie klasy. Następnie metody, które mają zostać uwidocznione są oznaczone adnotacją `@MModificator(mode=MModificator.ALLOW)`. Drugie podejście polega na udostępnieniu na poziomie klasy wszystkich metod oraz ukrywaniu wybranych przy użyciu adnotacji `@MModificator(mode=MModificator.HIDEN)`. Gdy opisany system adnotacji jest pomijany, wtedy każda metoda jest dostępna z poziomu języka zapytań. Własność ta jest przydatna, ponieważ pakiety z operacjami arytmetycznymi, które nie były tworzone z myślą o zastosowaniu w języku StreamAPAS mogą być swobodnie obsługiwane. Przykładem jest pakiet `Math`.

Ograniczenie drugie rozwiązano wprowadzając adnotację `MModificator.CUSTOM_OP_BASE`. Informuje ona, że pierwszym argumentem metody jest tablica składająca się z symboli definiujących zmienne. Dostęp do symboli

reprezentujących definicję zmiennych istnieje tylko w trakcie analizy semantycznej. Adnotacja ta zatem przekazuje dodatkowo informację, że metoda ta rozszerza analizę semantyczną. Użycie adnotacji MModificator.CUSTOM_OP_BASE wiąże się również z inną interpretacją wyniku zwracanego przez metodę. Metoda adnotowana może nie zwracać wyniku lub zwracać obiekt reprezentujący węzeł drzewa rozbioru składniowego. Kompilator w trakcie analizy semantycznej uruchamia metodę adnotowaną MModificator.CUSTOM_OP_BASE i jeżeli ma miejsce przypadek drugi, wtedy węzeł reprezentujący wywołanie metody jest zastępowany węzłem zwróconym przez wywołaną metodę. Dzięki takiemu rozwiązaniu usunięte jest również ograniczenie czwarte. Zauważmy, że przy użyciu metody adnotowanej MModificator.CUSTOM_OP_BASE można skonstruować funkcję zwracającą złożoną strukturę symboli jako wynik. Oznacza to, że tego typu metody usuwają także ograniczenie trzecie.

Na zakończenie prześledźmy zastosowanie powyższych adnotacji na przykładzie metody setGlobal zdefiniowanej dla obiektu Unit. Metoda ta przenosi definicję fabryki danych z przestrzeni publicznej do przestrzeni globalnej. Jej deklaracja ma postać:

```
…
@MModificator(mode=MModificator.CUSTOM_OP_BASE | MModificator.ALLOW)
 public void setGlobal(OperatorBase[] aOperatorBase, DatasetI aDataset) {
…
```

Tylko metoda setGlobal dla obiektu Unit jest dostępna z poziomu języka zapytań, dlatego klasa Unit jest adnotowana przez @CModificator(mode = CModificator.HIDEMEMBERS), z kolei metoda setGlobal jest adnotowana przez MModificator.ALLOW. Dodatkowo zastosowano adnotację MModificator.CUSTOM_OP_BASE, ponieważ metoda ta wymaga dostępu do definicji zmiennej. Zauważmy, że z poziomu języka zapytań lista argumentów metody setGlobal składa się wyłącznie z aDataset. Pierwszy argument jest pomijany, ponieważ jego wartość jest dołączana automatycznie przez kompilator w trakcie analizy semantycznej. Jak widzimy połączenie mechanizmów adnotacji i reflekcji sprawia, że składnia języka zapytań automatycznie się aktualizuje wraz z dodaniem nowych funkcjonalności do obiektów rozbioru składniowego.

Podsumowując, przy użyciu adnotacji tworzących system udostępniania metod oraz adnotację powiadamiającą o uruchomieniu metody jako element analizy semantycznej, usunięto wszystkie ograniczenia mechanizmu refleksji.

3.15 Reprezentacja indeksów oraz operacji na hurtowniach danych

Obecnie strumieniowe bazy danych są głównie rozwiązaniami prototypowymi. Najczęściej w przemyśle istnieją dodatki do relacyjnych baz danych, umożliwiające przetwarzanie danych strumieniowych w wąskim zakresie tematycznym. Przykładem jest tutaj dodatek Oracle Stream dedykowany głównie do monitorowania zmian w tabelach i migracji danych pomiędzy relacyjnymi bazami danych. Coraz liczniejsza grupa algorytmów strumieniowych przeznaczonych do przeprowadzania złożonych analiz takich jak: analiza sekwencji, realizacja zapytań przestrzennych i zapytań o trajektorie wskazuje, że rozwój strumieniowych baz danych zmierza ku powstaniu nowych systemów analitycznych [35] tzw. strumieniowych hurtowni danych.

Warto zauważyć, że bogactwo składni języka StreamAPAS umożliwia wyrażać również tego typu zapytania. Przy użyciu metody statycznej użytkownik potrafi zdefiniować własny *Task* budujący dowolną strukturę indeksującą zasilaną poprzez fabrykę danych. Ilustracją użycia powyższego mechanizmu jest zapytanie 5.

Zapytanie 5: Należy zbudować indeks o nazwie *traffic*, który jest R-drzewem zasilanym przez strumień *I* zawierający informacje o bieżącym położeniu pojazdów. Poniżej przedstawiono fragment zapytania, który realizuje powyższą treść:

```
gis.Rtree::task(„traffic”, {$I.point[true], $I.name}, …)
```

Skorzystano tutaj z własności, że metoda statyczna w języku StreamAPAS może implementować brakujące elementy analizy semantycznej. Dzięki temu, że jest ona uruchomiona w trakcie kompilacji, istnieje dostęp do drzewa rozbioru składniowego i tabeli symboli, który pozwala zweryfikować poprawność argumentów i zrealizować inne operacje na tabeli symboli i drzewie rozbioru semantycznego. Warto tutaj zauważyć, że argumentami metody mogą być wartości stałe, strumienie jak i pod-zapytania oraz wyniki wywołania innych metod statycznych. Rozwiązanie

takie pozwala dołączyć się aktywnie użytkownikom lub innym firmom do tworzenia bibliotek rozszerzających standardową funkcjonalność. Przeanalizujmy teraz bogactwo składni języka StreamAPAS służącej do obsługi indeksów. W tym celu zaprezentowane zostaną zapytania 6 i 7.

Zapytanie 6: Chcemy użyć indeks *traffic* do wyszukania najbliższych pięciu pojazdów, które można wysłać do wypadku. Informacja o położeniu wypadków jest przekazywana przez strumień *help*.

```
select result{$traffic{}}
where traffic{kNN($help{})};
```

Do realizacji tego zapytania skorzystano z obiektu fabryki danych *traffic*. Obiekt ten udostępnia zasoby poprzez Proxy, a jego definicja z poziomu języka zapytań korzysta ze składni drzewa atrybutów. W języku StreamAPAS z zastosowaniem Proxy spotkaliśmy się przy definiowaniu okien na strumieniach. Ten przykład pokazuje, jak tą samą składnię języka użyto do zbudowania procesu ekstrakcji na innych typach fabryk danych.

Zapytanie 7: Przyjmijmy, że indeks *traffic* jest drzewem agregatów, a zadaniem zapytania jest zliczenie pojazdów przebywających na wyznaczonym obszarze.

```
select result{$traffic{}}
where traffic{contain ($areas{}), measures[sum()]};
```

Zauważmy, że drzewo agregatów jest strukturą wielowymiarową. Korzystając z terminologii hurtowni danych, w rozpatrywanym przykładzie do wymiarów zaliczamy czas oraz przestrzeń, po której poruszają się pojazdy. Do grupy faktów zaliczamy miary opisujące poszczególne pojazdy. W zależności od prowadzonych analiz część miar może występować równocześnie jako fakty i wymiary. Zaprezentowany przykład ilustruje zalety zastosowania składni drzewa atrybutów do zdefiniowania Proxy. To podejście pozwala w sposób zwarty przedstawić szereg operacji realizowanych na wymiarach oraz faktach. W omawianym przykładzie operator wyliczający sumę jest zdefiniowany na węźle `measures`, który reprezentuje fakty; a operację wyszukiwania pojazdów definiuje funkcja `contain`.

Przedstawione przykłady pokazują, że składnia języka StreamAPAS pozwala obsługiwać nie tylko strumieniowe bazy danych, ale także zadania związane

z działaniem strumieniowych hurtowni danych. Kluczową rolę pełni tutaj zastosowanie składni drzewa atrybutów do zdefiniowania Proxy. Dzięki temu operacje na strukturach danych można przedstawić w przestrzeni wielowymiarowej. Takie podejście, czyni sposób definiowania zapytania bardziej intuicyjnym. Źródłem inspiracji do takiej składni jest język MDX, który w podobny sposób udostępnia obsługę wymiarów w hurtowni danych zaimplementowanej w oparciu o MS SQL Server.

3.16 Wnioski i uwagi

Istniejące obecnie propozycje języków strumieniowych są silnie związane ze składnią SQL. Należy jednak pamiętać, że wymagania stawiane systemom baz danych kiedy tworzono język SQL były znacznie mniejsze niż wymagania stawiane obecnym systemom informatycznym. W szczególności w systemach strumieniowych ważne jest to, aby do języka można było łatwo dodawać brakującą funkcjonalność. Często ta funkcjonalność reprezentuje złożone procesy, co uzasadnia także zastosowanie bardziej rozbudowanego systemu typów, dzięki któremu operowanie na wielu argumentach jest prostsze i mniej narażające na popełnienie błędów.

Aby spełnić wymagania stawiane językom strumieniowym, w tym rozdziale zaproponowano autorski język StreamAPAS łączący język deklaratywny z elementami języka obiektowego. Głównym celem takiej modyfikacji jest otrzymanie składni minimalizującej nakłady związane z wprowadzeniem nowej funkcjonalności. Obecnie dodanie jej do baz danych wiąże się często z rozszerzaniem składni języka, co jest realizowane najczęściej przez producenta. W przypadku strumieniowych baz danych potrzeba dołączenia nowej funkcjonalności pojawia się znacznie częściej. Dlatego rozwiązanie minimalizujące konieczność zmiany składni języka jest kluczowym celem projektowym. Z tego punktu widzenia osiągnięta została nowa jakość obsługi strumieniowej bazy danych. W zaproponowanym rozwiązaniu wyróżniamy dwa elementy nowatorskie. Pierwszym elementem są zasady podziału języka na obiekty. Drugim jest drzewo atrybutów.

W językach programowania obiekt reprezentuje algorytm oraz strukturę danych. W zbudowanym języku zapytań rozszerzono postrzeganie obiektowości

wprowadzając dwa poziomy abstrakcji. Meta-typy służą do definiowania elementów analizy semantycznej, a typy definiują algorytmy przetwarzania danych. Przykładem meta-typu jest fraza Select-From, z kolei typem jest klasa implementująca metodę `Math.sin(…)`. Język zapytań umożliwia użytkownikowi dodawanie oraz posługiwanie się zarówno meta-typami jak i typami. Elementem charakterystycznym względem rozwiązań konkurencyjnych jest możliwość dodawania własnych meta-typów. Idea polega na tym, że użytkownik posługując się mata-typami nie precyzuje algorytmów przetwarzania, tylko definiuje wyrażenie jakie ma zostać wyliczone. Meta-typ implementuje wiedzę potrzebną do zrealizowania zadanego zadania. Dzięki temu użytkownik nie musi znać implementacji wszystkich wersji operatorów, ponieważ ich doboru dokonuje meta-typ. Stosując funkcjonalność meta-typów użytkownik może zdefiniować własną fabrykę danych. Meta-typy umożliwiają również bezpośrednią integrację strumieniowej bazy danych z elementami innych systemów poprzez definiowanie własnych typów źródeł oraz ujść strumieni. Spoglądając na nasz język StreamAPAS z perspektywy budowy kompilatora udało się zautomatyzować rozszerzanie funkcjonalności języka poprzez dołączanie pakietów klas do kompilatora. Do uzyskania tej automatyzacji wykorzystano mechanizm adnotacji oraz refleksji w języku Java. Warto tutaj zauważyć, że użytkownik definiuje nową funkcjonalność wyłącznie w języku Java. Brak dodatkowych plików konfiguracyjnych jest dużą zaletą, ponieważ nie ma wtedy konieczności uczenia się składni nowego języka. Ponadto unika się błędów wynikających z nie właściwej wersji pliku konfiguracyjnego dla zmienionej implementacji funkcjonalności.

Drugim elementem nowatorskim w języku StreamAPAS są drzewa atrybutów. Przedstawiają one zbiory atrybutów jako hierarchiczną strukturę. Takie rozwiązanie pozwala pogrupować tematycznie atrybuty krotek, co ułatwia zarządzanie danymi oraz pozwala skrócić listy argumentów funkcji. Z drzewami atrybutów związane są dodatkowo dwie własności. W przeciwieństwie do popularnych w językach programowania typów nazwanych, nasz język StreamAPAS implementuje strukturalne podobieństwo typów złożonych. Jeżeli przyjmiemy, że strumieniowa baza danych jest platformą rozszerzaną przez kilka grup programistów, pojawia się problem zarządzania wieloma typami. Z jednej strony istnieje ryzyko kolizji nazewnictwa, z drugiej połączenie rozwiązań rozwijanych niezależnie będzie wymagało licznych konwersji typów. Aby uniknąć

tych problemów, w języku StreamAPAS zastosowano podobieństwo strukturalne, które weryfikuje czy węzły dostępne w drzewie atrybutów tworzą strukturę zgodną z definicją argumentu funkcji. Drugą charakterystyczną własnością drzewa atrybutów jest składnia służąca do jego definiowania. Łączy ona rolę deklaracji schematu oraz definicji operacji na atrybutach. Rozwiązanie to jest wzorowane na języku TQL, który został stworzony do obsługi zapytań na danych zapisanych w formacie XML. Zaletą tego podejścia jest zwięzłość zapisu. Jeżeli deklaracja zmiennych oraz operacji na niej wykonywanych jest blisko siebie, wtedy minimalizuje się ryzyko popełniania błędów przez programistę. Podobny trend jest obserwowany w językach programowania. O ile w języku C++ spotykamy się z rozdzieleniem deklaracji typów oraz kodu programu na dwa pliki, co skutkuje to potrzebą napisania sporej liczby wierszy kodu stanowiących nadbudowę składni języka. W nowych językach takich jak C# lub Java to podejście już nie jest kontynuowane.

Podsumowując, w rozdziale tym dowiedziono poprawności pierwszej tezy rozprawy: „Możliwe jest rozszerzenie języka zapytań strumieniowych o drzewo atrybutów oraz elementy języka obiektowego, co ograniczy potrzebę zmiany jego składni oraz ułatwia budowę złożonych zapytań”.

Rozdział 4. Architektura

Skrócenie czasu opóźnień odpowiedzi oraz redukcja zapotrzebowania na pamięć są kluczowymi elementami kierującymi rozwojem przetwarzania strumieniowego. W tych systemach aktualizacja wyniku zapytania następuje z chwilą nadejścia kolejnej krotki na wejście. W DBMS operatory rozpoczynają działanie dopiero, gdy posiadają kompletne tabele danych. Podsumowując, synchronizacja dostępu do danych w tradycyjnych bazach danych jest na poziomie tabel rekordów z kolei w systemach strumieniowych jest na poziomie pojedynczych krotek (rekordów). Ponadto w przeciwieństwie do DBMS przetwarzanie zapytania ma charakter ciągły. Sprawia to, że znacznie więcej procesów jest realizowanych w trakcie przetwarzania zapytania. Zaliczamy tutaj zarządzanie metodami składowania danych [17], modyfikacje operatorów[73], zintegrowana optymalizacja wielu zapytań [57] i przywracanie zapytania do stanu stabilnego po wystąpieniu awarii[15,50]. Konsekwencją tak licznej grupy komponentów strumieniowej bazy danych jest szeroki problem optymalizacji. Badania w tej dziedzinie koncentrują się wokoło takich zagadnień jak: schedulery, architektura, algorytmy przywracania stabilności po awarii, optymalizatory i budowa operatorów. Powyższe algorytmy działają w jednym środowisku i wpływają na siebie nawzajem. Prowadzone badania upraszczają ten aspekt, koncentrując się nad wąskim zagadnieniem w izolacji od współistnienia innych elementów. Biorąc pod uwagę fakt, że strumieniowa baza danych równolegle prowadzi kilka intensywnych procesów obliczeniowych, skonstruowanie wydajnej architektury jest zadaniem trudnym. Przetwarzanie strumieniowe jest realizowane w środowisku wielowątkowym i rozproszonym, w konsekwencji głównym czynnikiem decydującym o skalowalności systemu jest udział synchronizacji w przetwarzaniu, o czym mówi prawo Amdahl-a [93]. Powyższa specyfika dziedziny skłoniła do postawienia tezy, że **możliwa jest redukcja czasów odpowiedzi lub obciążenia pamięciowego poprzez taką integrację komponentów strumieniowej bazy danych, która skutkuje redukcją obszarów synchronizowanych**. Zagadnienie to zostało przeanalizowane na trzech płaszczyznach.

Pierwsza płaszczyzna opisana w podrozdziale 4.1 obejmuje problem doboru architektury oraz interfejsów, poprzez które dołączane są komponenty tworzące strumieniową bazę danych. Decyzja o wyborze architektury przekłada się na udział czasu synchronizacji w trakcie przetwarzania danych jak i operacji sterujących. Pojedyncze zapytanie strumieniowe składa się z sieci operatorów fizycznych połączonych ze sobą strumieniami. DSMS jest aplikacją wielowątkową, która przetwarza tysiące krotek na sekundę, prowadzi to do sytuacji, że poprawa parametrów transmisji krotek przekłada się bezpośrednio na poprawę jakości przetwarzania. Zawężając listę elementów, które mają największy wpływ na wydajność przetwarzania strumieniowego otrzymujemy trójkę komponentów: schedulery, architektura operatorów i architektura systemu. Podstawy związane z budową schedulerów można odnaleźć w [9,10,53]. Aspekty implementacji interfejsów operatorów strumieniowych zostały omówione w pracy [18]. Tematyka architektur całego systemu była tematem badań nad prototypowymi systemami takimi jak: Aurora&Borealis [1], STREAM [11], PIPES [18], Eddy[60], Nile [2]. Powyższe badania abstrahują jednak od szczegółów związanych z wydajną implementacją powyższych komponentów w środowisku wielowątkowym. Dlatego przeprowadzono badania mające na celu wytypowanie wydajnych algorytmów buforujących strumienie oraz przeanalizowano dobór polityki zarządzania pulą wątków, ponieważ rozwiązania obecne bazują na uniwersalnych modelach, które nie korzystają w pełni z własności strumieniowych baz danych. Wyniki badań nad tą tematyką zostały opublikowane na międzynarodowej konferencji [45].

W podrozdziale 4.2 przeanalizowano przyczyny wzrostu czasu odpowiedzi w zapytaniach strumieniowych. Definicja strumieni zakłada, że nieznany jest czas pojawienia się na wejściu kolejnej krotki. Ponadto dopuszcza się wysoką dynamikę zmian intensywności źródeł, prowadzi to do dużych problemów z modelowaniem zapotrzebowania na pamięć oraz opóźnień. Biorąc dodatkowo pod uwagę wysoką intensywność źródeł, monitorowanie strumieni ogranicza się do pomiaru wartości średnich w zadanym przedziale czasu, ponieważ jest to najmniej złożone obliczeniowo zadanie. W oparciu o pomiary wartości średniej intensywności oraz średniej selektywności operatorów tworzone są modele służące schedulerom i innym optymalizatorom. Podczas badań nad architekturą strumieniowej bazy danych okazało się, że znaczącą rolę w opóźnieniach odgrywa wymóg przetwarzania

krotek zgodnie z porządkiem chronologicznym. Warunek ten jest pomijany w stosowanych obecnie modelach analitycznych czasu przetwarzania strumieniowego, chociaż przeprowadzone testy wskazują na duży wpływ tego czynnika. Zaistniała sytuacja skłoniła do zbudowania modelu analitycznego, który pozwala na dokładniejszą predykcję czasów odpowiedzi.

Trzecie ujęcie zagadnienia synchronizacji omówione w podrozdziale 4.3 dotyczy ograniczenia synchronizacji poprzez zastosowanie partycji operatorów. Zagadnienie to pojawiło się w trakcie prowadzenia badań nad architekturą oraz modelami analitycznymi strumieniowych baz danych. Partycje operatorów tworzone są po to, aby zastąpić połączenia strumieniowe bezpośrednimi połączeniami pomiędzy operatorami. Poprzez odpowiedni dobór, które strumienie zastąpić połączeniami bezpośrednimi można wprowadzić optymalizację pamięciową jak również czasową. Dotąd rozpatrywano algorytmy, które do zdefiniowania partycji wymagają dostępności selektywności i kosztu przetwarzania krotki przez operator. Ograniczenie się do tych miar sprawia, że algorytmy rozpatrują tylko część możliwych konfiguracji partycji. Przyczyna tego tkwi w stosowanym modelu analitycznym zapytania, który uniemożliwia przeanalizowanie bardziej złożonych partycji operatorów. Powyższe okoliczności, skłoniły do poszukiwań algorytmu pozwalającego ocenić jakość dowolnej konfiguracji partycji. Osiągnięte rezultaty w tym obszarze zostały zaprezentowane na konferencji międzynarodowej [39].

Badania obejmujące drugie i trzecie ujęcie problemu synchronizacji odnalazły poparcie ze strony komisji, która przyznała grant na ich przeprowadzenie[1].

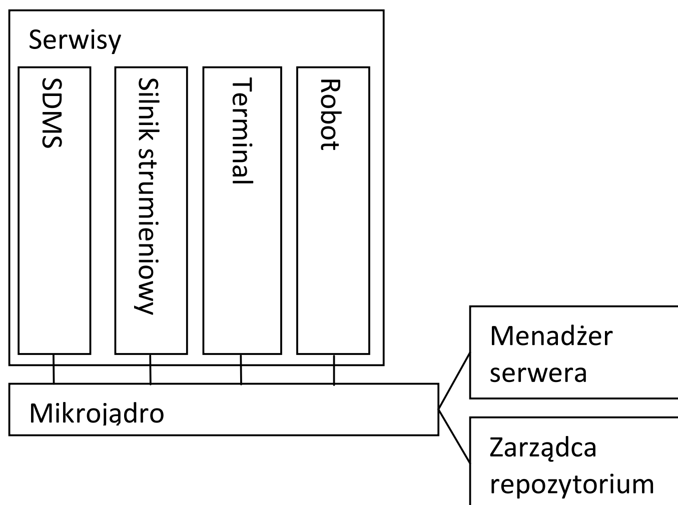
4.1 Dobór architektury

Architektura autorskiego systemu StreamAPAS została wzorowana na budowie serwerów aplikacji [19]. Głównym wymaganiem stawianym temu

---

1 Wsparcie dla współpracy sfery nauki i przedsiębiorstw Programu Operacyjnego Kapitał Ludzki, współfinansowanego przez Unię Europejską ze środków Europejskiego Funduszu Społecznego. POKL.08.02.01-24-019/08

rozwiązaniu była budowa komponentowa oraz wysoka adaptowalność systemu do uruchamiania w środowisku rozproszonym. Aby zrealizować te założenia wybrano architekturę opartą na mikro jądrze i serwisach. Ideą tej budowy jest podział systemu na węzły komunikujące się za pośrednictwem sieci.



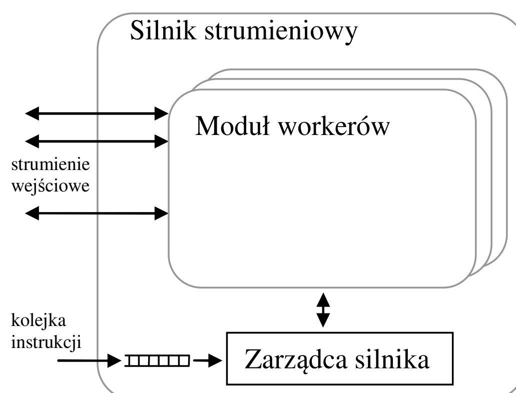
Rys. 4.1. Przebieg kompilacji zapytań

Każdy węzeł posiada architekturę przedstawioną na rys. 4.1. Jest on tak skonstruowany, aby każdy z serwisów korzystając z tych samych interfejsów mógł się komunikować zarówno z serwisami umieszczonymi lokalnie jak i zdalnie. W tym celu zdefiniowano mikro jądro, które udostępnia szereg interfejsów służących do komunikacji z serwisami, ponadto koordynuje współpracę pomiędzy nimi. Dzięki temu elementowi podstawowe funkcje są realizowane szybko oraz utworzony jest szkielet systemu wielomodułowego. Dodatkową funkcjonalność na poziomie serwera realizują moduły: menadżer serwera oraz zarządcę repozytorium. Zarządca repozytorium gromadzi lokalnie informacje o konfiguracji sytemu oraz o dostępnych węzłach. Rolą menadżera serwera jest koordynowanie cyklem życia serwera. Warto zauważyć, że przedstawione obecnie elementy systemu tworzą szkielet, który może posłużyć do budowy różnych systemów nie tylko strumieniowych baz danych. Realizacja strumieniowej bazy danych jest zaimplementowana poprzez dwa serwisy. Serwis SDMS pełni rolę menadżera strumieniowej bazy danych. Jest on odpowiedzialny za kompilację zapytań, optymalizację i koordynowanie przetwarzania. Szczegółowy jego opis przedstawiono w podrozdziale 3.1. Drugim serwisem jest silnik strumieniowy, który realizuje obliczenia. Dodatkowo w systemie

zdefiniowano serwis robot symulujący zachowanie użytkownika, co jest wykorzystywane do automatyzacji testów. Istnieje także serwis terminal funkcjonujący w dwóch trybach: tekstowym albo graficznym.

4.1.1 Pojęcia podstawowe

Przetwarzanie strumieni jest realizowane przez serwis silnika strumieniowego, jego zarys budowy przedstawia rys. 4.2. Aby umożliwić porównanie różnych realizacji przetwarzania strumieniowego w zależności od doboru optymalizatorów, wprowadzono budowę modułową. Każdy moduł jest w pełni funkcjonalnym pod-systemem uruchamiającym operatory strumieniowe. Przykładowo, w ten sposób można uruchomić przetwarzanie strumieniowe, gdzie każdy operator jest realizowany przez oddzielny wątek lub gdzie jeden wątek jest dystrybuowany pomiędzy wszystkie operatory.



Rys. 4.2. Architektura silnika strumieniowego

Aby usystematyzować opis budowy modułu workerów wprowadzono obiekt worker. Nadzoruje on uruchamianie operatorów fizycznych. Moduł może składać się z jednego lub większej ilości workerów. Wszystkie workery należące do modułu tworzą *pulę* workerów. W zależności od typu modułu workerów, workery mogą działać w środowisku jednowątkowym lub wielowątkowym. Przetwarzanie zapytania polega na cyklicznym uruchamianiu operatorów fizycznych, a dokładnie na uruchamianiu kolejnych operatorów fizycznych na wskazanych workerach. Za wybór operatora fizycznego, który ma zostać uruchomiony odpowiada scheduler.

Fragment zapytania realizowany przez dany worker jest definiowany przy użyciu grafu DAG oraz reprezentacji fizycznej. Dla przypomnienia *operator fizyczny* jest realizacją węzła DAG, z kolei *bufor strumienia* jest algorytmem reprezentującym krawędź DAG. Na potrzebę dalszej analizy definiujemy *ścieżką przetwarzania*.

Definicja 4.1. Ścieżka przetwarzania

Jest nią łańcuch operatorów rozpięty pomiędzy źródłem a operatorem ujścia z pominięciem operatora źródłowego. Dodatkowo operator leżący na końcu ścieżki przetwarzania nazywamy liściem.

Najprostszym typem schedulera jest Round-Robin [10]. Działanie jego polega na cyklicznym aktywowaniu operatorów posiadających krotki do przetworzenia. Aktywowany operator kończy swoje działanie z chwilą przetworzenia ostatniej krotki wejściowej lub po przetworzeniu ustalonego limitu krotek. Podstawową wadą tego algorytmu jest to, że nie kontroluje czasów odpowiedzi oraz zapotrzebowania na pamięć. Z drugiej strony algorytm ten gwarantuje, że nie wystąpi zjawisko zagłodzenia operatorów.

Strategia FIFO [10] polega na przetwarzaniu krotek zgodnie z czasem ich nadejścia do systemu. Algorytm taki w odróżnieniu do Round-Robin pozwala osiągnąć krótsze czasy opóźnień. Strategia ta jednak nadal nie uwzględnia własności operatorów, co ogranicza jakość optymalizacji.

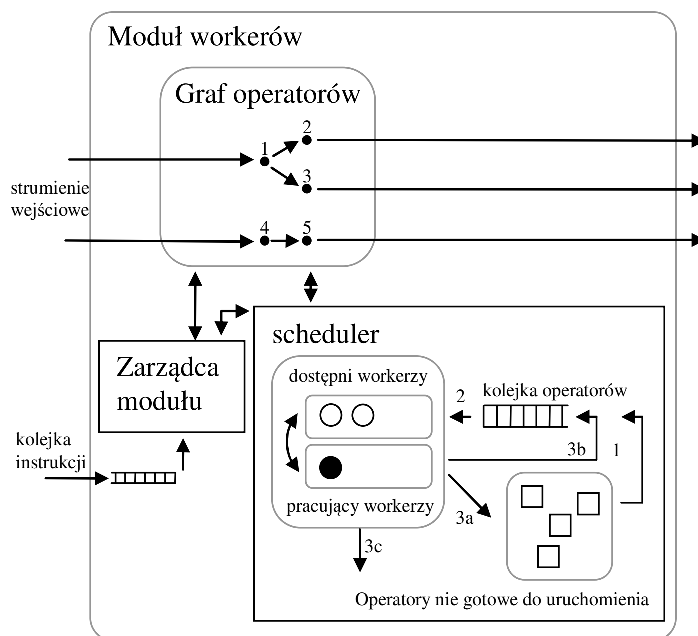
Strategia Chain [9] została utworzona w celu zmniejszenia rozmiaru strumieni w trakcie przetwarzania zapytania. Algorytm Chain korzysta z dwóch statystyk: średniego czasu przetwarzania krotki oraz selektywności operatora. Niestety algorytm ten nie gwarantuje uniknięcia tzw. zjawiska zagłodzenia operatorów. W [9] zaproponowano modyfikację Chain-flush wprowadzającą ograniczenie na czas pobytu krotek w systemie. Dzięki temu w przypadku chwilowych pik obciążenia systemu, nie dochodzi do wstrzymania generowania wyników, co ma miejsce dla wersji Chain. W Chain-flush dodatkowo ustalamy parametr wskazujący maksymalny czas opóźnienia krotek. W chwili, gdy kolejna do przetworzenia krotka przez operator przekroczony prób opóźnienia, następuje uruchomienie schedulera FIFO w celu minimalizacji czasów opóźnień. Szczegółowy opis tego algorytmu jest dostępny w [10].

Analiza literatury prowadzi do wniosku, że schedulery przeznaczone do strumieniowych baz danych zakładają pomijalnie niskie czasy transferu krotek oraz niskie opóźnienia rozpoczęcia transmisji. W ten sposób czas odpowiedzi jest zależny tylko od obciążenia CPU. Drugim uproszczeniem konstrukcyjnym schedulerów jest założenie, że wszystkie operatory są umieszczane w jednej kolejce. Realizacja takiej kolejki w systemach rozproszonych lub wielowątkowych jest nie akceptowalna. Podsumowując, istnieje szeroki obszar badawczy obejmujący wydajną realizację schedulerów w środowisku rozproszonym lub współbieżnym.

4.1.2 Podstawowy model puli wątków

Algorytmy schedulerów można zaimplementować korzystając z pojedynczej puli wątków i pojedynczej kolejki operatorów oczekujących na uruchomienie. W takim systemie wyróżniamy dwa typy wątków:

- wątki workerów *W*, w ramach których uruchamiane są operatory fizyczne,
- wątki *Src*, które uruchamiają źródła strumieni.



Rys. 4.3. Moduł workerów z kolejką operatorów i pulą wątków

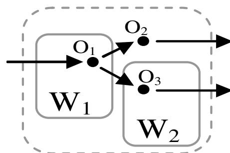
Na rys. 4.3 przedstawiono moduł workerów realizujący te założenia. Uruchomienie nowego zapytania poprzedza wstawienie operatorów fizycznych do kolekcji operatorów *niegotowych do uruchomienia*. Następnie zapytanie jest

zasilone danymi. W wyniku tego operatory fizyczne z niepustymi wejściami są wstawione do kolejki operatorów oczekujących na uruchomienie. Pojawienie się operatora w tej kolejce wyzwala metodę przydzielającą operatorowi fizycznemu wolny worker z puli dostępnych workerów. Wyliczenie zadania zdefiniowanego przez operator wiąże się z przeniesieniem workera do puli workerów przetwarzających. Po ukończeniu obliczeń wyróżniamy trzy potencjalne przypadki. Operator zostaje ponownie wstawiony do kolejki, gdy nadal posiada na wejściach krotki do przetworzenia. Operator jest wstawiony do kolekcji operatorów niegotowych do uruchomienia, gdy nie posiada danych do przetworzenia. Operator opuszcza system, gdy zapytanie jest wycofywane.

W rozpatrywanym modelu wyróżniamy trzy akcje wymagające synchronizacji:

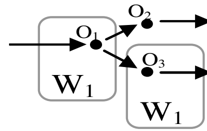
- przemieszczanie workerów pomiędzy kolekcjami workerów dostępnych oraz workerów przetwarzających,
- wstawianie oraz usuwanie elementów z kolejki operatorów oczekujących na uruchomienie oraz
- obsługa buforów strumieni.

Na rys. 4.4 przestawiono migawkę z działającego systemu. W danym momencie przetwarzane są operatory O_1 i O_3 na workerach W_1 i W_2. Realizacja strumienia łączącego te operatory musi być synchronizowana, ponieważ operator O_1 zasila operator O_3. Należy zauważyć, że w rozpatrywanym modelu operator fizyczny może zostać uruchomiony na dowolnym workerze, gdyż po wyjęciu z kolejki *operatorów oczekujących na uruchomienie* jest on przekazywany pierwszemu wolnemu workerowi. A zatem, każdy bufor może być potencjalnie współbieżnie zapisywany i odczytywany. W konsekwencji, każdy strumień musi być synchronizowany.



Rys. 4.4. Dystrybucja losowa operatorów pomiędzy wątki

Jeżeli chcemy zaimplementować bardziej złożony scheduler taki jak Chain lub Chain-flush, przedstawioną powyżej architekturę należy rozszerzyć o dodatkowy wątek *S* gromadzący statystyki o operatorach i aktualizujący priorytety operatorów. Zaletą modelu korzystającego z pojedynczej puli wątków i pojedynczej kolejki operatorów jest równomierne obciążenie workerów. Jest to konsekwencją tego, że omówiona strategia nie pozwala, aby operator czekał na przetworzenie, gdy w systemie istnieje wolny worker. Jeżeli przyjmiemy, że czas przetwarzania pojedynczego operatora fizycznego jest znacząco większy niż czas potrzebny na synchronizację operacji odczytu kolejnego operatora z kolejki oraz przemieszczenie workerów pomiędzy kolekcjami, wtedy wydajność przedstawionej architektury wielowątkowej jest wysoka. Niestety wydajność tą psuje konieczność synchronizacji buforów strumieni. Druga wada dotyczy implementacji bardziej złożonych schedulerów takich jak Chain. Zwróćmy uwagę na to, że wtedy do struktur operatora fizycznego odwołuje się wątek workera *W* oraz wątek sterujący S. Powyższy przypadek wskazuje na potrzebę dodatkowej synchronizacji, co spowalnia działania operatora.



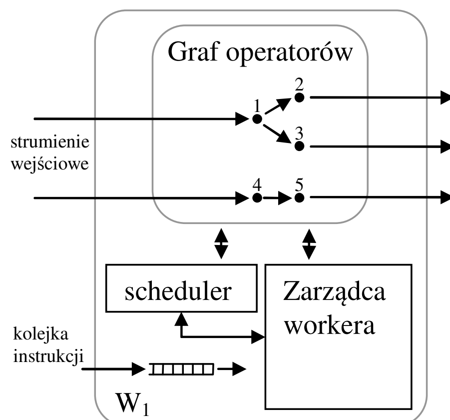
Rys. 4.5. Dystrybucja kontrolowana operatorów pomiędzy wątki

Przyjmijmy, że dwa następujące po sobie operatory są zawsze uruchamiane w ramach tego samego workera. Ten szczególny przypadek przedstawiono na rys. 4.5. W konsekwencji realizacja bufora strumienia nie musi być synchronizowana, ponieważ operacje wstawiania oraz odczytu nie są współbieżne. Jeżeli w systemie wielowątkowym potrafimy wyróżnić taką własność, wtedy można zredukować liczbę synchronizowanych buforów.

Powyższy wstęp prowadzi do podziału modułu workerów na takie, które nie definiują powiązań pomiędzy wątkami workerów oraz operatorami fizycznymi. Oraz architekturę wiążącą ze sobą operatory fizyczne i workery.

4.1.3 Pula wątków zorientowana na szybki transport krotek

Moduł workerów tego typu składa się z zbioru workerów, z których każdy posiada przypisany jeden wątek. Budowę pojedynczego workera ilustruje rys. 4.6).



Rys. 4.6. Budowa pojedynczego workera

Worker W_1 posiada lokalny zbiór operatorów fizycznych, które uruchamia zgodnie z regulaminem lokalnego schedulera. Ponadto worker W_1 posiada dwa typy kolejek wejściowych. Kolejkę zawierającą polecenia sterujące, takie jak dodanie/usunięcie operatora z workera lub odczyt statystyk operatora. Drugi typ kolejek wejściowych to bufory strumieni wejściowych, które są zasilane przez zewnętrzne źródła. Jeżeli W_1 w danym momencie posiada zarówno niepustą kolejkę poleceń oraz co najmniej jeden gotowy do uruchomienia operator fizyczny, wówczas wpierw zostaną wykonane polecenia sterujące, potem przetwarzane są operatory fizyczne.

W odróżnieniu do architektury opisanej w sekcji 5.1.2, każdy z workerów dysponuje lokalną kolejką operatorów oczekujących na uruchomienie. Aby zrównoważyć obciążenie workerów stosuje się migrację operatorów fizycznych pomiędzy workerami. W tym rozdziale pomijamy kwestię przemieszczania operatorów fizycznych, przeanalizujemy wydajność platformy, przyjmując że operatory fizyczne po rozmieszczeniu na wskazanych workerach pozostają tam, aż do zakończenia procesu przetwarzania strumieni.

Zarówno operacje sterujące jak i przetwarzanie operatorów fizycznych są wykonywane przez pojedynczy wątek. W konsekwencji, gdy są przetwarzane polecenia sterujące, wszystkie operatory fizyczne należące do workera W_1 są

wstrzymane oraz należą do przestrzeni pamięci tego samego wątku. Własność ta upraszcza implementację poleceń sterujących takich jak odczyt statystyk operatorów oraz modyfikacje ich priorytetów. Dzięki niej nie ma potrzeby tworzyć dodatkowych mechanizmów synchronizujących dostęp do operatora. Także operacje modyfikujące konfigurację sieci wewnątrz pojedynczego workera nie wymagają dodatkowej synchronizacji, ponieważ tylko wątek workera W_1 w danym momencie ma dostęp do zasobów operatorów. Omawiana architektura mapuje operatory fizyczne na zbiór workerów. Dzięki temu możemy wykryć konfigurację zaprezentowaną na rys. 4.5. Co z kolei pozwala zastosować niesynchronizowane bufory strumieniowe, które są wydajniejsze od ich odpowiedników synchronizowanych.

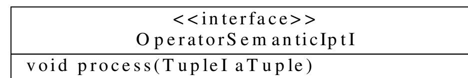
Podsumowując wprowadzony typ architektury jest zorientowany na szybki transport krotek pomiędzy operatorami. Jest to uzyskane dzięki budowie, która pozwala zastąpić bufory synchronizowane odpowiednikami niesynchronizowanymi. Inną zaletą omawianego workera jest fakt, że struktury danych związane z nim są obsługiwane wyłącznie przez jego wątek. Własność ta redukuje liczbę obiektów współdzielonych pomiędzy wiele wątków a także upraszcza rozwój oprogramowania. Wadą tej architektury jest brak prostego mechanizmu równomiernie obciążającego wątki workerów.

4.1.4 Architektura operatora fizycznego

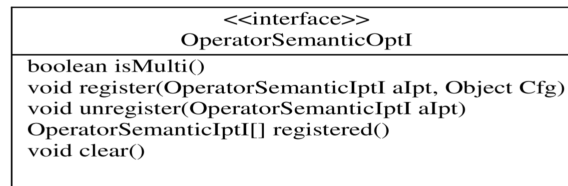
W systemie StreamAPAS architektura operatorów fizycznych została zoptymalizowana pod kątem minimalizacji czasu obsługi krotek i sterowania operatorem. Podobnie jak w systemie PIPES [18], model wejścia/wyjścia danych skonstruowano tak, aby przy użyciu tego samego interfejsu móc zaimplementować zarówno wejście operatora, jak również wejście strumienia. Takie rozwiązanie pozwala dołączać do wejść zarówno strumienie jak i bezpośrednio operatory fizyczne.

```
<<interface>>
OperatorSemanticI
-----------------------------------
OperatorSemanticIptI[]  getIpt()
OperatorSemanticOptI[]  getOpt()

void flush(long aTs, long aTe)
void register(StopObserverI aObserver)
void unregister(StopObserverI aObserver)
```

Rys. 4.7. Interfejs operatora fizycznego



Rys. 4.8. Interfejs wejścia operatora



Rys. 4.9. Interfejs wyjścia operatora

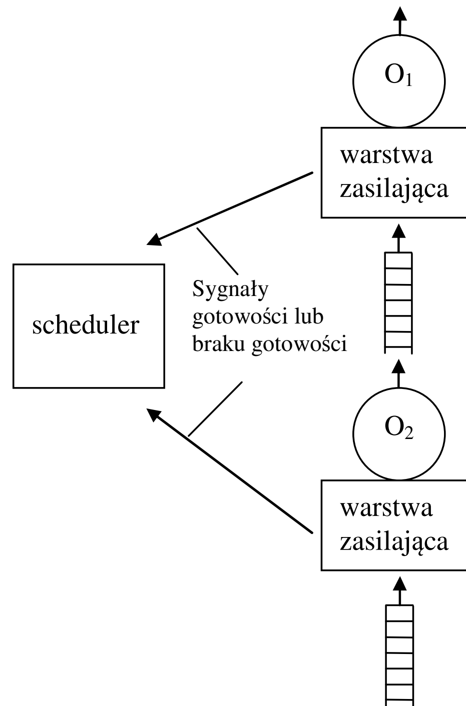
Interfejs operatora fizycznego (rys. 4.7) składa się z metod `getIpt` oraz `getOpt`, które zwracają obiekty reprezentujące odpowiednio wejścia oraz wyjścia. W odróżnieniu do architektury zaproponowanej w PIPES, omawiany interfejs jest rozszerzony o możliwość rejestracji obserwatora, który jest informowany o zakończeniu działania operatora. Dzięki temu powiadamianie o zakończeniu pracy operatora jest identyczne dla: operatorów źródeł, operatorów leżących wewnątrz ścieżek przetwarzania i operatorów ujść. Reprezentacja każdego wejścia (rys. 4.8) i wyjścia (rys. 4.9) poprzez oddzielny obiekt pozwala dołączyć dodatkową funkcjonalność do każdego z wejść/wyjść indywidualnie. Dzięki temu można swobodnie konstruować operatory składające się z kilku pod-operatorów. Przykładem takiego użycia jest dołączenie pomiaru średniego czasu przetwarzania krotek lub selektywności operatora. Dodatkowo interfejs operatora zawiera metodę `flush`, która implementuje aktualizację czasu lokalnego operatora. Metoda ta jest używana w celu przyspieszenia obsługi krotek typu Boundary[88,15] wewnątrz operatora złożonego. Osiągane jest to poprzez wywołanie kaskadowo metod `flush` na operatorach składowych.

Implementacja wyjścia operatora fizycznego (rys. 4.9) pozwala na dwie konfiguracje. Pierwsza konfiguracja dopuszcza rejestrację tylko jednego obiektu wejścia, w rezultacie implementacja wyjścia operatora jest wydajniejsza. Druga konfiguracja jest dedykowana dla operatora z wieloma wyjściami. Aby zarządzać listą obiektów wejść zdefiniowano metody: dodawania (`register`), usuwania (`unregister`), listowania (`registred`), czyszczenia (`clear`). Konfiguracja wyjścia operatora jest

odczytywana przy użyciu metody `isMulti` zwracającej *false*, gdy operator dopuszcza rejestrację tylko pojedynczego obiektu wyjścia.

4.1.5 Transport krotek

Operator fizyczny jest tylko obiektem realizującym obliczenia. Zarządzanie strumieniami leży po stronie workera. W tym celu zdefiniowano warstwę zasilająca operator fizyczny, odpowiada ona za transport krotek oraz sygnalizowanie *gotowości* i *niedostępności* operatora. Rolę warstwy zasilania ilustruje rys. 4.10.



Rys. 4.10. Warstwa zasilająca operator

Obiekt spełniający rolę warstwy zasilania udostępnia funkcjonalności:

1) rejestrację obserwatora wydarzeń: *gotowości* oraz *niedostępności*,
2) obsługę wstawiania krotek do strumienia s_i,
3) implementuje metodę *getNextTuple* zwracającą kolejną krotkę do przetworzenia oraz indeks strumienia, z którego ją pobrano.

Biorąc pod uwagę własności architektury opisanej w sekcji 5.1.3 wyróżniamy dwa typy warstw zasilania:

- Warstwa zasilająca operator O_i niewymagająca synchronizacji, wtedy operator O_i posiada n wejść i należy do workera W_j oraz dodatkowo każde z jego wejść jest podłączone do operatorów, które również należą do workera W_j.
- Warstwa zasilająca operator O_i wymagająca synchronizacji, wtedy co najmniej jedno z wejść operatora O_i jest podłączone do operatora nie należącego do W_j.

Kluczowe dla transportu krotek jest to, aby były one przekazywane ze strumieni wejściowych do operatora w kolejności chronologicznej [1] (w przypadku krotek temporalnych leksykograficznej [56]). Interesującą realizację tych założeń przedstawiono w pracy [12], idea polega na tym aby przechowywać dodatkowo ostatnie wartości znaczników czasu, w celu skrócenia okresów kiedy operator jest niedostępny do przetwarzania. Ideę tego rozwiązania ilustruje algorytm 5.1. Przyjmijmy, że operator strumieniowy składa się z n wejść. Każde wejście operatora x_i jest połączone ze strumieniem s_i. Dodatkowo warstwa zasilania z każdym wejściem x_i kojarzy rejestr znacznika czasu r_i. Wartość r_i jest aktualizowana zawartością znacznika czasu krotki, która jest przekazana na wejście x_i. Wartość r_i pozostaje niezmieniona do momentu przekazania na wejście x_i kolejnej krotki. Przypuśćmy, że na obu wejściach operatora *złączenia* występują krotki o identycznych znacznikach czasu. W powyższej sytuacji można przetworzyć krotki z obu strumieni wejściowych. Jeżeli algorytm składałby się wyłącznie z punktu 1) przetwarzanie krotek zostałoby wstrzymane po opróżnieniu jednego ze strumieni. Dzięki punktowi 2) w algorytmie 5.1 odczyt krotek jest wstrzymany dopiero, gdy pobrana zostanie ostatnia krotka z identyczną wartością znacznika czasu.

Algorytm 5.1: Wybór następnej krotki do przetworzenia

1. Jeżeli każdy strumień s_i jest nie pusty: wybierana jest krotka o najmniejszym znaczniku czasu,
2. Jeżeli co najmniej jeden strumień s_i jest pusty: wybierana jest krotka o znaczniku czasu równym $\min_{i \in [1,n]} r_i$.

Do realizacji optymalizacji potrzebny jest odczyt statystyk operatorów. Aby ich pomiar był spójny wprowadzono synchronizację odczytu przy użyciu krotek

z flagą URGENT, które są propagowane od źródeł w kierunku ujść. Odstęp pomiędzy kolejnymi krotkami URGENT definiuje interwał, dla którego są naliczane statystyki. Dodanie tych krotek wiąże się z rozszerzeniem algorytmu warstwy zasilania. Gdy na wejściu dowolnego strumienia pojawi się krotka z tą flagą oznacza to, że należy ją natychmiast przekazać na wejście operatora, niezależnie od wartości znacznika czasu. Gdy operator odbierze na każdym z wejść taką krotkę, aktualizuje statystyki i propaguje na wyjścia nową krotkę URGENT.

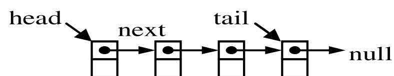
Implementacja powyższej funkcjonalności na poziomie warstwy zasilania jest następująca. Jeżeli na wejściu strumienia s_i pojawi się krotka z ustawioną flagą URGENT, wtedy do oddzielnej kolejki FIFO wstawiana jest para (*i*, *krotka*). Funkcja wybierająca następną krotkę do przetworzenia wpierw pobiera elementy z tej kolejki, następnie ze strumieni wejściowych zgodnie z algorytmem 5.1. W chwili, kiedy kolejka jest pusta oraz brak krotki spełniającej kryteria 1) i 2) warstwa zasilająca generuje sygnał *niedostępności*. Sygnał *gotowości* jest wysyłany, gdy w warstwie zasilania operatora pojawi się pierwsza krotka spełniająca kryteria 1) i 2) lub pojawi się krotka z ustawioną flagą URGENT. Należy podkreślić, że sygnał *gotowości* jest generowany przez wątek wstawiający krotkę do strumienia, z kolei sygnał *niedostępności* jest generowany przez wątek odczytujący strumienie. Oznacza to, że muszą one być synchronizowane, gdy komunikacja następuje pomiędzy dwoma różnymi wątkami.

4.1.6 Implementacja synchronizowanej warstwy zasilania

Bufor strumienia realizuje funkcję prostej kolejki FIFO, gdzie tylko jeden operator wstawia krotki do strumienia. Podobnie tylko jeden operator pobiera krotki ze strumienia. Oznacza to, że kolejka działa w konfiguracji 1 producent – 1 konsument. Kolejka FIFO obsługująca krotki z flagą URGENT działa w konfiguracji wielu producentów - jeden konsument, ponieważ wstawiane są do niej krotki z *n* wejść.

W [63,85,26,62,94] przestawiono algorytm oraz dowód poprawności szybkiej kolejki FIFO działającej w konfiguracji wielu producentów - wielu konsumentów. Jej zaletą jest brak operacji lock/unlock, korzysta ona z operacji atomowych i operacji typu CAS (Compare-And-Swap). Powyższy algorytm użyto

do implementacji kolejki zawierającej krotki z flagą URGENT. Strumień krotek realizuje funkcję kolejki FIFO o własności: 1 producent – 1 konsument, dlatego wprowadzono dodatkowe uproszczenia algorytmu.



Rys. 4.11. Kolejka z elementem wartownikiem na początku listy

Na rys. 4.11 przedstawiono przykładową kolejkę. Dodanie nowego elementu *new* składa się z podoperacji: ustawienia *new.next = null*, odczytu ostatniego elementu *t* przy użyciu zmiennej *tail*, ustawienia wartości pola *t.next = new*, aktualizacji zmiennej *tail = new*. Odczyt elementu składa się z podoperacji: odczyt elementu *t* wskazywanego przez zmienną *head*, odczyt pola *t.next* i zapisanie go do zmiennej *tmp*, aktualizacji zmiennej *head = tmp*, przekazanie na wyjście elementu *tmp*. Zauważmy, że przez cały czas pobytu elementu w kolejce zachodzą następujące własności. Operacja wstawiania zmienia tylko wartość zmiennych *tail* oraz *next*, z kolei operacja odczytu zmienia tylko wartość zmiennej *head*. Efekt ten uzyskujemy dzięki wprowadzeniu dodatkowego elementu na początku kolejki. Jeżeli przedstawiony algorytm działa w modelu 1 producent – 1 konsument, nie ma wtedy konieczności synchronizacji operacji wstawiania oraz odczytu. Implementacja powyższego algorytmu w języku Java wymaga dodatkowo uwzględnienia modelu pamięci i specyfikacji wątków w JVM [84,61]. Zgodnie z tą specyfikacją instrukcja *synchronized* nie tylko zakłada oraz zdejmuje blokadę na obiekcie monitora, ale również zapewnia aktualność odczytanych wartości zmiennych. Przyjmijmy, że wątek *A* zmienił wartość zmiennej *x*, następnie wątek *B* odczytał wartość zmiennej *x*. Otrzymane wartości mogą być różne, jeżeli dostęp do zmiennej *x* nie był objęty instrukcją *synchronized*. Przyczyna takiej sytuacji tkwi w tym, że procesor w celu przyspieszenia operacji IO korzysta z mechanizmu cache. Jeżeli wątek *B* działa na innym rdzeniu procesora niż wątek *A*, zmiany zawartości zmiennej wymagają dodatkowej synchronizacji. Dokument JSR-133 tłumaczy zasady widoczności zmian zmiennych *volatile* oraz *final* w modelu pamięci JVM. Zmienna *final* raz zainicjalizowana pozostaje niezmienna, dodatkowo każdy wątek może bezpiecznie odczytać jej zawartość bez potrzeby synchronizacji. Jeżeli zmienna *final* jest typu obiektowego, JVM gwarantuje, że z chwilą jej udostępnienia w systemie, wskazuje ona na obiekt w pełni zainicjalizowany obiekt. Wyobraźmy sobie, że nowo utworzony

obiekt zapisujemy do zwykłej zmiennej obiektowej. Specyfikacja JVM zezwala na podział tej operacji na kilka faz, w których optymalizator wpierw tworzy surowy obiekt, następnie go przypisuje do zmiennej, na koniec go inicjalizuje. Powyższy scenariusz tłumaczy, dlaczego spójny odczyt zmiennej obiektowej przez kilka wątków nie jest gwarantowany, nawet jeżeli jest ona ustawiona jednokrotnie wewnątrz konstruktora klasy. Specyfikacja modelu pamięci JVM gwarantuje, że zapis oraz odczyt zmiennych typu *volatile* jest atomowy oraz zmiana wartości zmiennej *volatile* jest natychmiast widoczna przez inne wątki. Podsumowując, zmienne typu *volatile* oraz *final* zapewniają odczyt spójnych wartości bez konieczności użycia instrukcji *synchronized*. Dodatkowo, wprowadzane zmiany wartości zmiennych są natychmiast widoczne przez inne wątki bez konieczności korzystania z instrukcji *synchronized*. Specyfikacja modelu pamięci JVM wskazuje także, że zapis/odczyt zmiennych *final* jest szybszy aniżeli *volatile*. Po uwzględnieniu dokumentu JSR-133 algorytm kolejki FIFO pracującej w konfiguracji 1 producent – 1 konsument wymaga następujących modyfikacji. Zmienne *tail*, *head* oraz *next* należy zadeklarować jako zmienne typu *volatile*. Zmienną *item* należy zadeklarować jako *final*. Poniżej zamieszczono implementację algorytmu w języku Java.

Algorytm 5.2. Kolejka FIFO 1 producent – 1 konsument.

```
class TupleNode {
 final TupleI item;
 volatile TupleNode next;
 public TupleNode(TupleI aItem, TupleNode aNext) {
 item = aItem;
 next = aNext;
 }
}

volatile TupleNode head;
volatile TupleNode tail;

public PipeTupleList() {
 head = tail = new TupleNode(null,null);
}

public final void pushNode(TupleI newNode) {
 TupleNode node = new TupleNode(newNode, null);
 tail.next = node;
 tail = node;
}

public final TupleNode popNode() {
```

```
 TupleNode tmp;
 if((tmp = head) == tail)
 return null;
 head = tmp.next;
 tmp.next = null;
 return head;
}
```

W celu osiągnięcia dodatkowej optymalizacji czasowej, implementując warstwę zasilania wprowadzono wersje dedykowane ze względu na liczbę strumieni zasilających. Wyróżniamy warstwę zasilania z jednym lub wieloma strumieniami wejściowymi. Dla osiągnięcia większej czytelności, w przedstawionej poniżej analizie pomijamy obsługę krotek z ustawioną flagą URGENT. Implementacja warstwy zasilania z wieloma strumieniami korzysta z następującej obserwacji. Przyjmijmy, że zmienna *minSize* jest równa minimalnej długości wszystkich strumieni wejściowych. Zauważmy, że jeżeli wartość *minSize = 0*, wtedy zachodzi punkt 2) algorytmu 5.2. Jeżeli *minSize > 0*, wtedy zachodzi punkt 1) algorytmu 5.2. Sygnał niedostępności nie jest generowany, jeżeli funkcja *getNextTuple* jest wywołana w chwili *minSize > 1*. Sygnał niedostępności może zostać wygenerowany tylko, gdy *minSize = 1* lub *minSize = 0*. Sygnał dostępności jest generowany tylko, gdy do strumienia wstawiamy pierwszą krotkę. Warstwa zasilania obsługuje kilka wątków, dlatego należy zaimplementować synchronizację gwarantującą, że sygnał niedostępności będzie poprzedzany sygnałem dostępności. Utworzony mechanizm synchronizacji opisano poniżej, korzysta on z obserwacji wartości zmiennej *minSize* oraz rozmiaru strumienia wejściowego. Operacja wstawiania krotki do strumienia jest synchronizowana przez monitor *m*, gdy strumień jest pusty. Operacja odczytu jest synchronizowana przez monitor *m*, gdy *minSize = 0* lub *minSize = 1*. Powyższy algorytm nie wymaga dostępu do dokładnej wartości *minSize* tylko do informacji kiedy *minSize > 1*. Własność ta pozwala sprowadzić *minSize* do trzech wartości. Jeżeli *minSize = 2* oznacza to, że minimalny rozmiar strumieni jest *> 1*. Poniżej przestawiono algorytm synchronizacji funkcji *getNextTuple*.

Algorytm 5.3. Synchronizacja funkcji *getNextTuple*.

```
if(minSize.get() > 1) {
 pobranie następnej krotki do przetworzenia
 punkt 2) algorytm 5.1
```

```
if(po wyjęciu krotki strumień ma rozmiar 1)
minSize = 1
else {
synchronized(m) {
switch(minSize.get()) {
case 0:
pobranie następnej krotki do przetworzenia
punkt 1) algorytm 5.1
aktualizacja minSize
if(brak krotki do przetworzenia w następnym
kroku iteracji)
wysłanie sygnału niedostępności
break;
case 1:
pobranie następnej krotki do przetworzenia
punkt 2) algorytm 5.1
aktualizacja minSize
if(brak krotki do przetworzenia w następnym
kroku iteracji)
wysłanie sygnału niedostępności
break;
default:
pobranie następnej krotki do przetworzenia
punkt 2) algorytm 1
if(po wyjęciu krotki strumień ma rozmiar 1)
minSize = 1
break;
}
}
}
}
```

Jeżeli minimalna długość strumieni jest *>1*, wtedy funkcja *getNextTuple* nie korzysta z monitora *m*. Zyskujemy dzięki temu dodatkowe przyspieszenie algorytmu. Aby mechanizm synchronizacji obsługiwał krotki z ustawioną flagą URGENT, należy zmodyfikować *minSize = minimalny_rozmiar_strumieni_wejściowych + rozmiar_kolejki_z_krotkami_priorytetowymi*. Zmienna *minSize* jest wtedy aktualizowana zarówno przez wątki wstawiające krotki jak i wątek odczytujący, dlatego wymagana jest synchronizacja odczytu/zapisu, do czego użyto poleceń typu CAS. W przypadku pojedynczego strumienia wejściowego wybór następnej krotki sprowadza się do sprowadzenia zawartości: 1) kolejki z krotkami priorytetowymi, 2) bufora strumienia.

4.1.7 Implementacja lokalnej warstwy zasilania

Działanie warstwy zasilania w wersji lokalnej różni się tym, że można zastosować implementację buforów bez synchronizacji. Szczególnie prosta jest wtedy implementacja obsługująca pojedynczy strumień wejściowy, która sprowadza się do pojedynczej kolejki FIFO.

4.1.8 Partycje operatorów

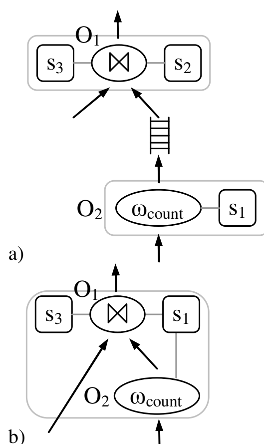
Tworząc partycje operatorów gwarantujemy, że grupa operatorów będzie uruchamiana bez dodatkowych opóźnień związanych z kolejkowaniem krotek w strumieniach. Własność ta stanowi główną zaletę tej techniki. W literaturze partycja operatorów pojawia się również pod nazwą operator złożony.

W strumieniowych bazach danych przekazywanie krotek pomiędzy operatorami może się odbywać się bezpośrednio lub za pośrednictwem strumieni. Przyjmijmy, że mamy prostą ścieżkę operatorów A i B. Jeżeli operator A po przetworzeniu krotki uruchamia operator B, wówczas mamy do czynienia z bezpośrednim przekazywaniem krotek. Jeżeli operator A wstawia wynikowe krotki do strumienia łączącego operator A z operatorem B. Następnie scheduler decyduje, kiedy te krotki są przetworzone przez operator B, wówczas mamy do czynienia z przekazywaniem krotek w sposób pośredni.

Definicja 4.2. Partycja operatorów

Partycją operatorów nazywamy grupę operatorów przekazujących krotki pomiędzy sobą tylko poprzez połączenia bezpośrednie. Tak połączone operatory w partycję są postrzegane przez system strumieniowy jako nowy operator [18].

W systemie StreamAPAS budowa partycji jest nadzorowana przez optymalizator regułowy wyróżniający trzy przypadki. Pierwsza reguła definiuje, że operator projekcji jest łączony z operatorem poprzedzającym. Druga reguła zakłada, że łączone są następujące po sobie operatory selekcji. W myśl trzeciej reguły operatory okna czasowego lub liczebnościowego są łączone z operatorem złączenia.



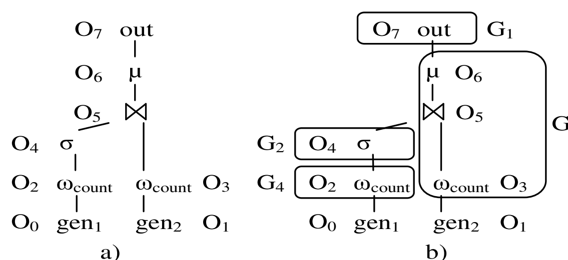
Rys. 4.12. Dwukrotne zmniejszenie zapotrzebowania pamięciowego dla kolekcji s_1 przez zastosowanie partycji operatorów

Omówmy bliżej ostatnią regułę. Implementacja okien liczebnościowych wymaga utworzenia kolekcji o ustalonym rozmiarze. Jeżeli operator okna liczebnościowego oraz operator łączenia działają oddzielnie (rys. 4.12 a)), wówczas oba operatory tworzą oddzielne kolekcje krotek. Rys. 4.12 b) ilustruje sytuację, kiedy oba operatory działają w ramach pojedynczego operatora złożonego, wtedy kolekcje s_2 i s_1 można zastąpić referencją do wspólnej kolekcji krotek s_1. W wyniku tego dwukrotnie zmniejsza się zapotrzebowanie na pamięć, ponadto operacja wstawiająca krotkę do kolekcji jest wykonywana raz. W przypadku pierwszej konfiguracji, w systemie rezerwowana jest pamięć dla kolekcji s_1 oraz s_2, ponadto dla każdej kolekcji operacje wstawiania i usuwania są wywoływane oddzielnie.

4.1.9 **Testy**

Testy przeprowadzono na komputerze z procesorem Intel Pentium M 715 / 1.5 GHz i pamięcią RAM 1G. Celem eksperymentów było zbadanie wpływu synchronizacji na wydajność przetwarzania zapytań. Do badań użyto mikro-test zilustrowany na rys. 4.13. Składał się on z operatora filtracji O_4, który realizował funkcję: $value_1 < 50$. Rozmiar okien liczebnościowych wynosił 500 elementów, a predykatem operatora łącznia O_5 była nierówność: $value_1 < value_2$. Taka konfiguracja skutkowała tworzeniem się na wyjściu operatora O_5 dużych zgrupowań krotek w losowych momentach, co stanowiło dodatkowe obciążenie testu.

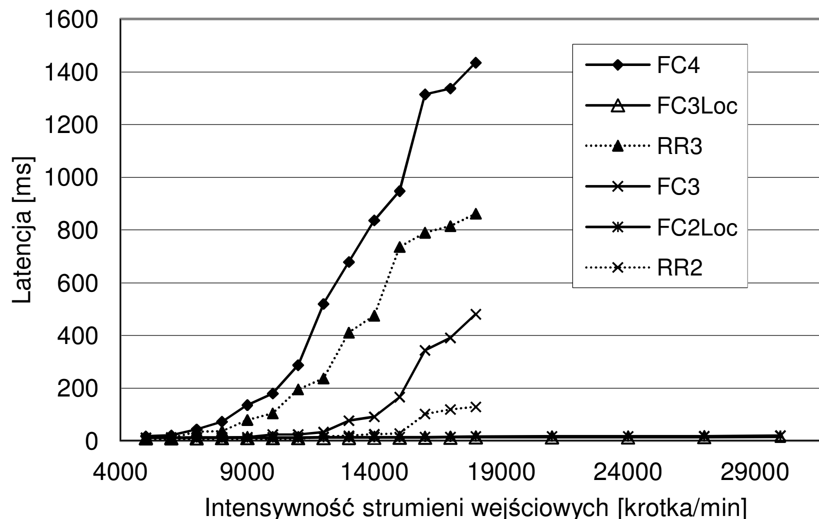
Pojedynczy eksperyment polegał na pomiarze średniego czasu odpowiedzi krotek wynikowych przy ustalonej szybkości źródeł gen_1 i gen_2, oraz dla zadanej konfiguracji schedulera i realizacji warstwy zasilania. Źródła danych generowały po 1000 krotek, tworząc przy tym stałe obciążenie z zakresu od 1000 do 30000 krotek na minutę. Każde ze źródeł generowało krotki z wartościami odpowiadającymi zmiennej losowej o rozkładzie równomiernym z zakresu 0-100. Konfiguracja schedulerów obejmowała algorytmy: Chain-flush oraz Round-Robin. Wpierw przeprowadzono eksperymenty dla architektury podstawowej opisanej w sekcji 4.1.2 dla schedulera Round-Robin oraz Chain-flush. Następnie zmierzono czasy odpowiedzi dla architektury zorientowanej na szybki przepływ krotek dla schedulera Chain-flush. Wymienione testy zostały przeprowadzone zarówno, gdy tworzenie operatorów złożonych było załączone (rys. 4.13 b)) jak i wyłączone (rys. 4.13 a)). Dla czytelności opis zbadanych konfiguracji zestawiono w tab. 4.1. Kolumna typ strumienia informuje czy w teście zastosowano implementację strumieni opartą na instrukcjach CAS, którą oznaczono nazwą *new*, albo skorzystano z standardowego mechanizmu *synchronized*(std).



Rys. 4.13. Optymalizacja pamięciowa uzyskana dzięki połączeniu operatorów

Tabela 4.1. Konfiguracja eksperymentów dla rys. 4.14

Nazwa wykresu	Typ shedulera	Grupowanie operatorów	Typ strumieni	Typ architektury
FC3	Chain-Flush	Nie	new	Pula wątków
FC4	Chain-Flush	Nie	std	Pula wątków
RR2	Round-Robin	Nie	new	Pula wątków
RR3	Round-Robin	Nie	std	Pula wątków
FC2Loc	Chain-Flush	Tak	new	Workery
FC3Loc	Chain-Flush	Nie	new	Workery

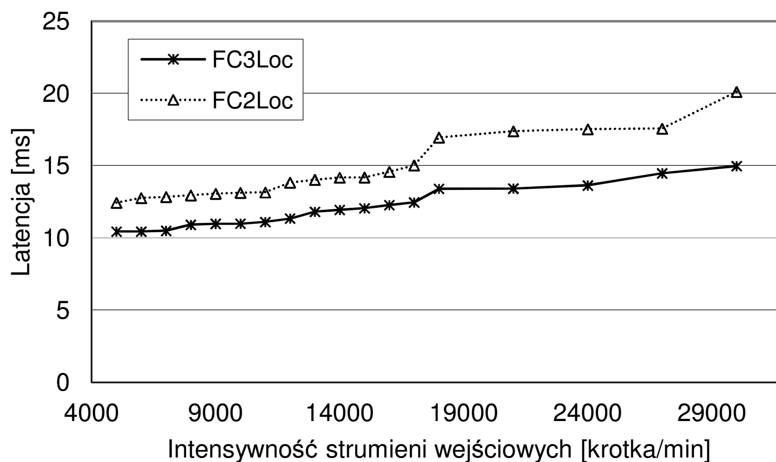


Rys. 4.14. Zestawienie opóźnień dla przeprowadzonych badań

Analiza zebranych pomiarów wskazuje jednoznacznie przewagę architektury workerów nad rozwiązaniem opartym o pojedynczą pule wątków. Jeżeli porównamy czasy odpowiedzi zgromadzone dla najwydajniejszej konfiguracji RR2 dla rozwiązania z pulą wątków, z najsłabszą konfiguracją FC2Loc opartą na nowej architekturze, poprawa jest ponad pięciokrotna dla intensywności 20000 krotek na minutę, co widzimy na rysunkach 4.14 i 4.15. Warto zaznaczyć, że zastosowanie kolejki wątków zmusza do dodatkowej synchronizacji modułów strumieniowej bazy danych, w konsekwencji implementacja bardziej złożonych schedulerów staje się mało wydajna, ilustruje to zestawienie pomiarów FC3 z RR2 i FC4 z RR3. Drugim wnioskiem wypływającym z analizy pomiarów jest trzykrotna poprawa wydajności dzięki zastosowaniu instrukcji typu CAS. Tą poprawę można odnotować porównując ze sobą wykresy FC3 z FC4 oraz RR2 z RR3. Rząd zmian pokazuje jak ważną rolę w całkowitym czasie odpowiedzi odgrywa wybór metody zasilania operatorów. Podsumowując, połączenie obu technik zaowocowało architekturą, która pozwala na wydajną obsługę złożonych algorytmów. Przykładowo dla schedulera Chain-flush zmiana architektury zaowocowała spadkiem czasów odpowiedzi z 950ms do 12ms dla natężenia strumieni wejściowych na poziomie 15000 krotek na min.

Zebrane wyniki pozwalają także odnotować dwa zjawiska. Zaskakujące jest, że analiza opóźnień architektury opartej o pulę wątków wskazuje na przewagę prostego algorytmu Round-Robin nad algorytmem Chain-fulsh. Sytuacja ta ma

miejsce, ponieważ uruchomione zapytanie jest na tyle proste, że ważną rolę odgrywa czas obsługi algorytmu schedulera. Algorytm Round-Robin korzysta z prostej kolejki, podczas gdy w Chain-flush istnieją dwie kolejki priorytetowe. Dlatego z wykresów można odczytać przewagę Round-Robin. Warto też dokładniej przyjrzeć się porównaniu konfiguracji FC2Loc oraz FC3Loc na rys. 4.15. Jak widzimy sieć przetwarzania, w której załączono grupowanie operatorów (wykres FC2Loc) jest wolniejsza niż sieć przetwarzania składająca się z prostych operatorów fizycznych. Przyczyna takiej sytuacji leży w szczegółach implementacyjnych buforów strumieniowych. Strumienie synchronizowane występują tam, gdzie operator O jest zasilany, przez co najmniej jedno źródło nie należące do Wolkera, do którego jest przypisany O. Konfiguracja z załączonym grupowaniem operatorów jest zilustrowana na rys. 4.13 b). Zgodnie z definicją, połączenia synchronizowane występują pomiędzy O_1-G_3, O_0-G_4 i G_2-G_3. Należy dodatkowo zwrócić uwagę, że warstwa zasilająca G_3 składa się z kilku buforów strumieniowych z powodu czego jest wolniejsza od warstwy zasilania składającej się z pojedynczego bufora. Konfiguracja z wyłączonym grupowaniem operatorów jest zilustrowana na rys. 4.13 a). Dla tej sieci przetwarzania strumienie synchronizowane to tylko O_0-O_2 i O_1-O_3, przy czym oba połączenia są zrealizowane przez warstwę zasilania z pojedynczym buforem. W rezultacie, pomimo że zastosowanie operacji grupowania operatorów fizycznych pozwala zredukować liczbę strumieni, bilans finalny okazał się gorszy.



Rys. 4.15. Zestawienie opóźnień dla przeprowadzonych badań

4.1.10 **Wnioski i uwagi**

Często przyjmuje się uproszczenie w którym marginalizuje się wpływ synchronizacji operatorów w trakcie pracy schedulerów [9,10,53]. W [18] autorzy zidentyfikowali sposoby przekazywania krotek pomiędzy operatorami w ramach pojedynczego i wielu wątków, rozwiązanie to jednak poruszało tylko kwestię implementacji operatorów z pominięciem zagadnień związanych z integracją modułów strumieniowej bazy danych. Warstwa integrująca łączy wiele modułów, wprowadza dodatkowe ograniczenia, których uwzględnienie jest ważne podczas optymalizacji. Ta luka wiedzy o strumieniowych bazach danych zainspirowała do badań nad wielowątkową implementacją przetwarzania strumieniowego oraz oceną szeroko rozumianej synchronizacji na czasy odpowiedzi. Należy zauważyć, że zaproponowane rozwiązania korzystają z ogólnodostępnych mechanizmów programowania współbieżnego, co pozwala je zastosować nie tylko w środowisku Java.

Analiza komunikacji operator-operator i operator-scheduler wskazała, że obecne algorytmy schedulerów szeroko pomijają ograniczenia związane z implementacją wielowątkową i rozproszoną. Aby wydajność operatorów strumieniowych nie została stłumiona przez warstwę zasilającą. W tym podrozdziale zwrócono uwagę na dysproporcję pomiędzy intensywnością komunikacji pomiędzy modułami strumieniowych baz danych, a częstością wymiany krotek pomiędzy operatorami. Uwzględnienie tego czynnika było podstawą nowej architektury modułu workerów.

Przeprowadzone testy pokazały, że nowa architektura znacząco obniża czasy odpowiedzi. Wprowadzone rozwiązanie cechuje się tym, że nie tylko wydajnie koordynuje pracę pomiędzy operatorami i schedulerem, ale również umożliwia dołączanie innych funkcjonalności do workera bez konieczności dodatkowej synchronizacji. Przeprowadzone badania objęły również studium wyboru algorytmów realizujących wymianę komunikatów i krotek pomiędzy workerami. Finalnie zaadaptowano algorytmy klasy *non-blocking and wait free*, które korzystają z instrukcji typu CAS oraz własności modelu pamięci JVM. Ta grupa instrukcji pozwoliła zmniejszyć trzykrotnie czasy opóźnień krotek wynikowych. Ponadto zaproponowane rozwiązanie jest uniwersalne, ponieważ specyfikacja modelu

pamięci JVM ogranicza się do technologii ogólnie dostępnych w systemach komputerowych.

Przeprowadzono dodatkowo testy algorytmu grupowania operatorów. Okazało się, że nie możliwa jest analiza zebranych wyników bez uwzględnienia ograniczeń warstwy zasilania, ponieważ wpływ synchronizacji na czasy odpowiedzi jest wysoki. W rozważanym przypadku ograniczenia warstwy zasilania sprawiły, że można było zmniejszyć dwukrotnie obciążenie pamięciowe dla operatora okna liczebnościowego i operatora łączenia albo skrócić o około 20% czasy odpowiedzi. Przyczyną takiego efektu jest budowa warstwy zasilania, która wymaga synchronizacji wszystkich strumieni wejściowych w chwili, kiedy co najmniej jedno ze źródeł operatora O nie należy do workera, w którym jest zarejestrowany ten operator.

Zebrane wyniki podczas testów architektury strumieniowej bazy danych stały się przyczynkiem do przeprowadzenia oddzielnych badań nad optymalizatorami grupowania oraz modelem estymującym czasy odpowiedzi, co zostanie przedstawione odpowiednio w podrozdziałach 4.3 i 4.2.

4.2 Modelowanie czasu odpowiedzi

W trakcie prac nad systemem StreamAPAS przeanalizowano szereg dedykowanych algorytmów schedulerów, do ważniejszych zalicza się: Round-Robin, FIFO, Greedy omówione w [10], HR, HNR przedstawione w[76], Chain, Chain-FIFO wraz z modyfikacjami [9,70] oraz rozwiązania pokrewne [75,13]. Przyglądając się nim bliżej można zauważyć, że zbudowane modele kosztowe ograniczają się do systemów jednowątkowych, gdzie obiektem optymalizacji jest pamięć albo czas odpowiedzi. Strumieniowe bazy danych są systemami rozproszonymi, w których istotna jest optymalizacja zarówno zasobów pamięciowych jak i minimalizacja czasów odpowiedzi. Brak dokładniejszych modeli analitycznych ogranicza wykorzystanie w pełni współistnienia w jednym systemie kilku algorytmów optymalizacji, ponieważ nie potrafimy zmierzyć ich wpływu na siebie nawzajem.

Podczas implementacji algorytmu Chain-flush w systemie StreamAPAS okazało się, że w literaturze dobór parametrów pracy algorytmów był doświadczalny.

W szczególności modelowanie czasu odpowiedzi w strumieniowych bazach danych jest słabo rozwinięte. Stosowane obecnie optymalizatory ograniczają opis pojedynczego operatora do średniej wartości selektywności i średniego czasu obsługi krotki. W szczególności modele te pomijają wpływ synchronizacji. Lepsze zrozumienie źródeł niestabilności podczas pracy strumieniowej bazy danych jest istotne dla rozwoju nowych optymalizatorów, dlatego zagadnieniu temu zostały poświęcone oddzielne badania[38]. W wyniku przeprowadzonych prac powstał nowatorski model wyznaczania czasów odpowiedzi w strumieniowych bazach danych.

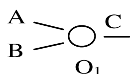
Omówienie tego modelu podzielono na trzy etapy. Na początku w punkcie 4.2.1 przeanalizowany będzie wpływ synchronizacji na przetwarzanie danych strumieniowych dla pojedynczego operatora. Następnie w punkcie 4.2.2 omówiono używany obecnie model korzystający z analizy wartości średnich oraz prawa Little'a. Finalnie w punkcie 4.2.3 oba podejścia są połączone w celu uzyskania modelu, który uwzględnia zarówno stopień obciążenia poszczególnych operatorów, jak również warunki wynikające z charakteru pracy strumieniowej bazy danych. Na koniec zaprezentowano testy oraz podsumowanie osiągniętych rezultatów odpowiednio w punktach 4.2.4 i 4.2.5.

4.2.1 **Analiza wpływu synchronizacji na czasy odpowiedzi**

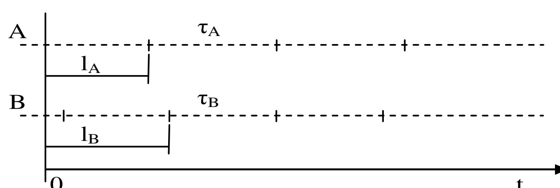
Operator strumieniowy przetwarza krotki wejściowe tylko wtedy, gdy można ustalić kolejność chronologiczną ich przetwarzania. Gdy operator posiada dwa strumienie wejściowe, wtedy jest on gotowy do uruchomienia, jeżeli oba strumienie posiadają co najmniej jedną krotkę, ponieważ dopiero wtedy można wskazać krotkę z najmniejszym znacznikiem czasu. Gdy operator jest zasilany pojedynczym strumieniem, wtedy zawsze jest gotowy do uruchomienia, jeżeli istnieje krotka w strumieniu wejściowym. Oznacza to, że dla tego przypadku nie istnieje zagadnienie synchronizacji. Mechanizm synchronizacji sprawia, że najwolniejszy strumień determinuje szybkość pracy operatora. W konsekwencji czas odpowiedzi może stać się wysoki, nawet jeżeli system nie jest obciążony. Istnieją rozwiązania przyspieszające synchronizację [12], jednak osiągana poprawa nie jest wysoka.

Dlatego analiza synchronizacji zostanie przeprowadzona dla rozwiązania podstawowego.

W dalszej części rozdziału skorzystamy z następujących założeń. Krotka posiada znacznik czasowy reprezentujący czas, w którym została zarejestrowana w systemie. Strumień przesyła krotki uporządkowane chronologicznie. Dodatkowo znany jest średni czas l opóźnienia krotek w strumieniu, informujący ile średnio czasu upłynęło od zarejestrowania krotki w systemie do momentu, w którym jest ona odczytana ze strumienia. Średni czas τ odstępu grup krotek w strumieniu wskazujący na rozmiar średniego kroku, z jakim jest aktualizowany czas wewnątrz strumienia. Jako grupę krotek rozumiany jest zbiór krotek o tych samych wartościach znacznika czasu. Zastosowanie pomiaru wartości średnich jest uzasadnione faktem, że strumienie są intensywnymi źródłami danych i konieczne jest użycie miar o niskiej złożoności obliczeniowej. Błąd wprowadzany przez takie uproszczenie zostania zmierzony w trakcie testów.



Rys. 4.16. Ilustracja operatora z dwoma wejściami

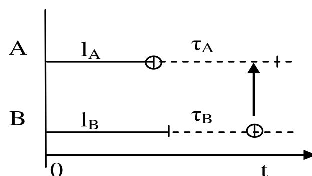


Rys. 4.17. Interpretacja graficzna opóźnienia oraz średniego odstępu pomiędzy grupami krotek

Rys. 4.16 zawiera poglądowy model operatora O_1 z dwoma strumieniami wejściowymi A i B oraz strumieniem wyjściowym C. Rys. 4.17 przedstawia sposób, w jaki modelowany jest upływ czasu w strumieniach. Kolejne grupy krotek napływające na wejścia A i B są zaznaczone na liniach przerywanych. Odstępy pomiędzy nimi są równe średnim czasom τ_A i τ_B. Jeżeli grupa krotek jest przetwarzana przez kilka operatorów, wówczas pojawi się na wejściu operatora O_1 z opóźnieniem. Wartości opóźnienia dla każdego z wejść jest opisana zmiennymi l_A i l_B.

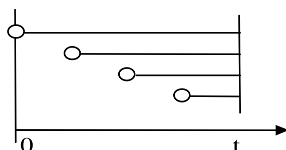
Zauważmy, że jeżeli krotki napływają na wejście A z opóźnieniem większym niż $l_B + \tau_B$, wtedy w strumieniu B zawsze znajdzie się co najmniej jedna krotka gdy

napływa krotka na wejście *A*. Podsumowując, jeżeli $l_B + \tau_B < l_A$, wtedy krotka na wejściu *A* może zostać natychmiast przetworzona. Przypadek przeciwny ($l_B + \tau_B \geq l_A$) ilustruje rys. 4.18.



Rys. 4.18. Pesymistyczny scenariusz dla wejścia A

Krotki na wejściu *A* pojawiają się z średnim opóźnieniem l_A, przy czym nie znamy dokładnie ich czasu pojawienia się. Wiemy jedynie, że grupa krotek pojawi się w przeciągu czasu τ_A. Ta sama sytuacja dotyczy wejścia *B*. W przypadku pesymistycznym grupa krotek na wejściu *A* pojawia się w chwili l_A, z kolei na wejściu *B* grupa krotek pojawia się w chwili $l_B + \tau_B$. Wtedy krotki z wejścia *A* są najdłużej buforowane.



Rys. 4.19. Metoda wyliczania średniego czasu opóźnienia

Wyliczymy teraz średni czas opóźnienia powstający podczas przetwarzania krotek z wejścia *A*. Jeżeli w chwili *t* operator stanie się gotowy do uruchomienia, wtedy średni czas opóźnienia grup krotek jest równy sumie szeregu arytmetycznego podzielonej przez liczbę grup, przedstawia to rys. 4.19. Suma opóźnień grup krotek dla wejścia *A* dla przypadku pesymistycznego wynosi:

$$l_{AC} = \frac{l_B + \tau_B - l_A + l_B + \tau_B - l_A - n_A \tau_A}{2}(n_A - 1) \tag{4.1}$$

Gdzie:

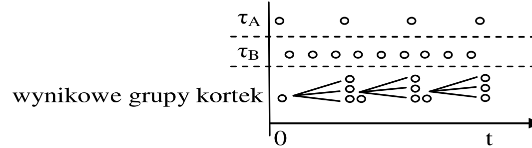
$$n_A = \frac{l_B + \tau_B - l_A}{\tau_A}$$

Gdy użyjemy powyższych równań, możemy wyliczyć średni czas opóźnienia przetwarzania grup krotek dla wejścia *A* który wynosi:

$$l_{AC} = \frac{l_B - l_A + \tau_B}{2} \tag{4.2}$$

Podsumowując, opóźnienie wynikające z potrzeby uporządkowania krotek wejściowych dla operatora z dwoma wejściami wynosi dla wejścia A:

$$l_{AC} = \begin{cases} 0 & gdy\ l_B - l_A + \tau_B < l_A \\ \frac{l_B - l_A + \tau_B}{2} & \end{cases} \tag{4.3}$$



Rys. 4.20. Tworzenie się nowych grup krotek na wyjściu operatora z dwoma wejściami

Rysunek 4.20 przedstawia zagadnienie tworzenia nowych grup krotek na wyjściu operatora z dwoma wejściami. Przyjmijmy, że średnia odległość pomiędzy kolejnymi grupami dla strumienia A jest trzy krotnie dłuższa niż dla strumienia B. W efekcie strumień B głównie czeka na krotki ze strumienia A. Innymi słowy średnio trzy grupy krotek ze strumienia B będą przetwarzane jednocześnie, w wyniku czego powstanie nowa grupa krotek na wyjściu. Aby wyliczyć średnią odległość bieżącej grupy krotek od swojego poprzednika należy wyliczyć poniższą całkę:

$$\tau_C = \int_{x=0}^{\tau_1} \frac{x + n(\tau_1 - x)}{1+n} dx = \frac{-2.5\tau_0^2 + \tau_0\tau_1 - (2\tau_0^2 + \tau_0\tau_1)\ln\frac{\tau_1}{\tau_0 + \tau_1}}{\tau_0} \tag{4.4}$$

Gdzie:

$$\tau_0 = \min(\tau_A, \tau_B)$$

$$\tau_1 = \max(\tau_A, \tau_B)$$

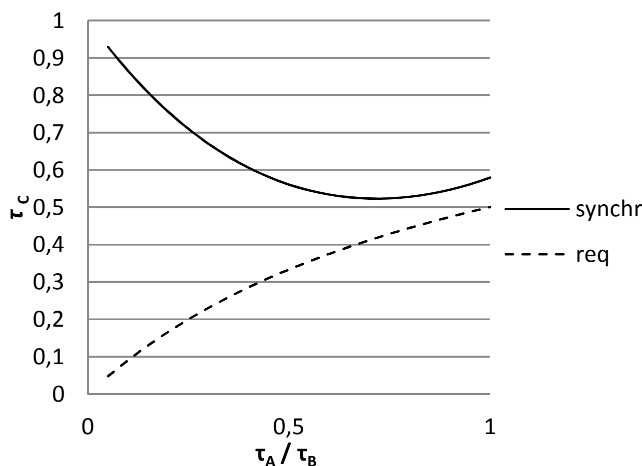
Korzystając z powyższego równania możemy wyznaczyć jak zmienia się średni odstęp pomiędzy kolejnymi grupami krotek w wyniku przetwarzania strumieni wejściowych przez operator o dwóch wejściach. Zauważmy, że średni odstęp między grupami krotek wskazuje średni odstęp pomiędzy kolejnymi znacznikami czasu przekazywanymi w strumieniu.

Aby przeanalizować znaczenie otrzymanego wzoru (4.4), przyjmijmy że dysponujemy operatorem binarnym. Na jego wejście B dołączono strumień z $\tau_B = 1$,

z kolei wartość τ_A dla strumienia A zmieniamy w zakresie *(0; 1]*. Następnie wyliczono średni czas odstępu dla krotek wynikowych zgodnie z wzorem na τ_C. Wynik zaznaczono liną ciągłą na rys. 4.21. Rozważmy teraz sytuację hipotetyczną, że znany jest porządek napływu krotek. Wtedy można byłoby natychmiast ustalić którą krotkę należy pobrać ze strumienia wejściowego, ponieważ nie ma potrzeby czekać na nadejście nowych krotek na wszystkie wejścia operatora. Scenariusz ten zatem odzwierciedla przypadek, kiedy nie występuje opóźnienie wywoływane przez algorytm porządkujący krotki wejściowe. Dla takiego modelu odstęp pomiędzy kolejnymi grupami krotek na wyjściu wyliczamy:

$$\tau_C = \frac{\tau_B}{1 + \frac{\tau_B}{\tau_A}} \tag{4.5}$$

Wartości dla tego przypadku zostały przedstawione na rys. 4.21 linią przerywaną.



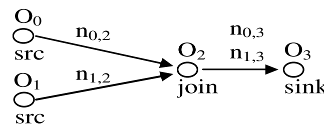
Rys. 4.21. Średni odstęp pomiędzy grupami krotek na wyjściu operatora dla różnych proporcji τ_A / τ_B

Zestawienie obu wykresów pokazuje, że wpływ synchronizacji na działanie strumieniowej bazy danych nie jest marginalny. W wyniku synchronizacji strumieni odstępy pomiędzy grupami krotek są znacznie większe niż w systemie bez synchronizacji. Odstęp pomiędzy grupami ma bezpośredni wpływ na ich rozmiar. Czym rzadziej pojawiają się grupy krotek w strumieniu tym większy ich rozmiar, a zatem powstają większe piki obciążenia. Linia ciągła na wykresie pokazuje ciekawą własność synchronizacji. Jeżeli τ_A jest z zakresu *(0.6; 1.0]* wówczas średni odstęp pomiędzy grupami krotek jest niemal dwukrotnie mniejszy na wyjściu operatora.

Zatem bez zastosowania dodatkowych mechanizmów optymalizacji (np. krotek Boundary [15]) odstęp pomiędzy kolejnymi znacznikami czasu w strumieniu jest zmniejszany w przybliżeniu o połowę. Jeżeli proporcja τ_A / τ_B jest mniejsza niż *0.3*, wtedy średni odstęp pomiędzy grupami krotek zbliża się do waratości maksymalnej równej *1*. Podsumowując, na podstawie wykresu oznaczonego linią ciągłą możemy stwierdzić, że ryzyko powstania pik obciążenia staje się wysokie, gdy wartość τ_A jest mniejsza od $0.6\tau_B$.

4.2.2 Podstawowy model opóźnień

Dynamika zmian intensywności strumieni jest wysoka, ponadto dąży się do tego, aby statystyki wyliczane na strumieniach nie były złożone obliczeniowo. W konsekwencji przetwarzanie danych strumieniowych jest najczęściej opisywane przy użyciu modelu opartego na analizie wartości średnich. Teoria ta jest używana w strumieniowych bazach danych, ponieważ nie uwzględnia rozkładów prawdopodobieństw obsługi i nadejścia kolejnej krotki. O popularności tego modelu świadczy fakt, że używany jest on przez wszystkie algorytmy schedulerów dedykowanych dla strumieniowych baz danych. Korzystają z niego również inne komponenty strumieniowej bazy danych: przykładowo algorytm podtrzymujące wysoką dostępność [50]. Aby opisać zapytanie przy użyciu tego podejścia należy przyjąć, że modelujemy sieć otwartą z wieloma klasami klientów oraz wieloma węzłami obliczeniowymi [28]. Na wskazanym węźle obliczeniowym są przetwarzane te operatory, które należą do pod-zapytania *u*.



Rys. 4.22. Podejście podstawowe do naliczania opóźnień

Rysunek 4.22 przedstawia przykładowe zapytanie, dla którego przeprowadzimy analizę operacyjną. Przyjmijmy, że $n_{a,b}$ oznacza średnią liczbę krotek wpływających do operatora *b* w wyniku przetworzenia jednej krotki ze źródła *a*. Ponadto definiujemy średnie opóźnienie $l_{a,b}$ krotek na wyjściu operatora *b*, które powstało w wyniku przetworzenia krotek ze źródła *a*. Zauważmy, że opóźnienie

krotek na wyjściu operatora 2 zależy od tego, którą ścieżką były one przetwarzane. Dla analizowanego zapytania istnieją dwie ścieżki operatorów: O_0, O_2 oraz O_1, O_2. Aby wyznaczyć całkowite opóźnienie l dla operatora O_2 należy wyznaczyć opóźnienie krotek dla obu ścieżek, następnie wyznaczyć wartość średnią:

$$l_b = \frac{\sum_{a \in Src} X^a n_{a,b} l_{a,b}}{\sum_{a \in Src} X^a n_{a,b}} \quad (4.6)$$

Aby dokonać estymacji $n_{a,b}$ należy wyznaczyć zbiór ścieżek P, które łączą pośrednio lub bezpośrednio źródło a z operatorem b. Wartość $n_{a,b}$ jest równa:

$$n_{a,b} = \sum_{path \in P} s_{path} \quad (4.7)$$

Do wyliczenia $l_{a,b}$ skorzystano z prawa Little'a [28]. Każdy z operatorów jest pośrednio lub bezpośrednio zasilany przez źródła. Częstotliwości z jaką krotki ze źródeł są przekazywane do operatora O wyznacza wektor przepustowości $X_o = (X_o^1, X_o^2, \ldots, X_o^K)$. Współczynnik wykorzystania węzła W_i przez operator O dla danych ze źródła k wynosi:

$$U_i^{(o,k)} = X_o^{(k)} \overline{B}_i^{(o)} = X^{(k)} \overline{D}_i^{(o,k)} \quad (4.8)$$

Gdzie:

$\overline{D}_i^{(o,k)} = B_i^{(o)} n_{k,o}$,

$\overline{B}_i^{(k)}$ - średni czas obsługi krotki przez operator o na węźle W_i.

Obciążenie węzła W_i wynosi:

$$U_i = \sum_{o \in u_i} \sum_{k=1}^{K} U_i^{(o,k)} \quad (4.9)$$

Dla stabilności sieci otwartej wykorzystanie wszystkich stanowisk musi być mniejsze od jedności, czyli: $U_i < 1$.

Średni czas przetwarzania przez operator o, które zostały wzbudzone przez źródło k w węźle W_i wynosi:

$$\overline{R}_i^{(o,k)} = \frac{\overline{D}_i^{(o,k)}}{1 - U_i} \quad (4.10)$$

Korzystając z powyższych wzorów potrafimy wyliczyć wszystkie wartości $\overline{R}_i^{(o,k)}$ dla zadanego zapytania. Oznaczają one całkowity czas przetwarzania krotek przez operator *O* wzbudzonych przez źródło *k*. Aby wyznaczyć czas odpowiedzi $l_{a,b}$, należy zsumować przyrost opóźnień pomiędzy operatorami na ścieżce *path* łączącej źródło *a* z operatorem *b*. Wiedząc tylko, że krotki napływają z częstotliwością równą wartości średniej, możemy wyliczyć średni czas obsługi jednej krotki który wynosi $\frac{\overline{R}_i^{(o,k)}}{2}$ dla operatora *o* wzbudzonego przez źródło *k*. Korzystając z tej własności opóźnienie dla ścieżki operatorów *path* wynosi:

$$l_{path} = \sum_{o \in path} \frac{\overline{R}_{Node(o)}^{(o,k)}}{2} \tag{4.11}$$

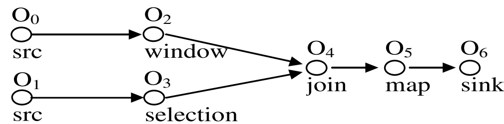
Gdzie:

path – ścieżka operatorów,

Node(o) – funkcja zwracająca indeks węzła, na którym uruchomiono operator *o*.

4.2.3 Podejście rozszerzone

Podejście rozszerzone łączy podstawową analizę wartości średnich z modelem opóźnień wynikającym z synchronizacji. Aby zilustrować działanie tego algorytmu skorzystamy z zapytania DAG1 na rys. 4.23.



Rys. 4.23. Zapytanie DAG1

Przyjmijmy, że struktura opisująca operator *op* składa się ze zmiennych:

τ – odstęp pomiędzy kolejnymi grupami krotek,

L – mapa wartości opóźnień w zależności od źródła pobudzenia *i*.

Wyznaczenie opóźnienia sprowadza się do przejścia grafu DAG od dołu w górę i wyliczenia $\overline{R}_i^{(o,k)}$ dla każdego operatora *o* i każdego źródła *k* tak jak opisano w punkcie 4.2.2. Kolejność odwiedzania operatorów jest zapisana w liście *AList*.

Przykładowo dla zapytania DAG1 przejście grafu od dołu w górę wiąże się z odwiedzeniem operatorów zgodnie z listą *AList*: O_0, O_1, O_2, O_3, O_4, O_5 i O_6. Metodę wyliczania czasu odpowiedzi definiuje algorytm 5.3.

Algorytm 5.3. Algorytm wyznaczania czasu odpowiedzi dla metody mieszanej.

```
foreach(op:AList){
 if(op jest źródłem) {
      initializeValues(op);
 } else if(op jest operatorem unarnym) {
      updateOp1(op);
 } else if(op jest operatorem binarnym) {
      updateOp2(op);
 }
}
```

Zgodnie z powyższym algorytmem, wpierw następuje inicjalizacja wartościami początkowymi. Metoda `initializeValues(op)` pobiera wartość przepustowości $X^{(op.id)}$ dla operatora *op* i dokonuje podstawień:

1) $op.\tau = \dfrac{1}{X_{op.id}}$

2) $op.L[X_{op.id}] = 0$

W kolejnych krokach algorytmu aktualizowane są czasy odpowiedzi w zależności czy mamy do czynienia z operatorem unarnym czy binarnym.

Metoda `updateOp1(op)` wylicza zmiany czasu odpowiedzi dla operatorów unarnych. Wyszukuje ona operator op_s, który zasila operator *op*. Następnie realizuje obliczenia:

1) Dla każdego źródła *k*:

$$op.L[k] = op_s.L[k] + \frac{\overline{R}_i^{(op.id,k)}}{2}$$

2) $op.\tau = \dfrac{op_s.\tau}{op.\mu}$

Za aktualizację czasów odpowiedzi dla operatorów binarnych odpowiada metoda `updateOp2(op)`. Wyszukuje ona operatory: op_{s1} i op_{s2}, które zasilają operator *op*. Następnie przeprowadzone są obliczenia:

1) Wpierw wyliczane jest opóźnienie l_{s1}(l_{s2}) dla operatora op_{s1}(op_{s2}). Przyjmijmy, że częstotliwości z jakimi krotki opuszczają operator O ze względu na źródło pobudzenia k definiuje wektor $Y_o = (Y_o^1, Y_o^2, \ldots, Y_o^K)$. Wtedy l_{s1} jest równe:

$$l_{s1} = \frac{\sum_{k \in K} Y_{s1}^k . L[k]}{\sum_{k \in K} Y_{s1}^k} \qquad (4.12)$$

Analogicznie wyliczamy l_{s2}.

2) Wzór (4.3) jest użyty do wyliczenia wartości $l_{s1,op}$ i $l_{s2,op}$.
3) Wyznaczana jest wartość l_{op} gdzie :

$$l_{op} = \frac{(l_{s1} + l_{s1,op})ta_{s1} + (l_{s2} + l_{s2,op})ta_{s2}}{ta_{s1} + ta_{s2}}$$

4) Aktualizowana jest wartość $op.\tau$ zgodnie ze wzorem (4.4).
5) Dla każdego źródła k wyliczany jest czas odpowiedzi krotek wynikowych:

$$op.L[k] = l_{op} + \frac{\overline{R}_i^{(op.id,k)}}{2}$$

Po zakończeniu algorytmu 5.3 wartości opóźnień są wyznaczane przy użyciu wzoru (4.12).

4.2.4 Testy

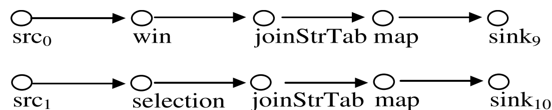
Testy zostały przeprowadzone na symulatorze zbudowanym tak, aby obrazował zjawiska zachodzące w naszym systemie StreamAPAS. Głównym powodem wyboru symulatora był fakt, że takie narzędzie pozwala na realizację dokładnych pomiarów cykli strumieniowej bazy danych oraz obciążenia pamięciowego. Innymi zaletami zastosowania symulatora było odizolowanie badanego procesu przetwarzania strumieniowego od wpływu jakości połączeń pomiędzy węzłami oraz zmieniającego się obciążenia komputera wywołanego programami współdzielącymi procesor.

Do eksperymentów przygotowano dwie grupy zbiorów danych. Pierwsza reprezentowała krotki napływające do systemu zgodnie z rozkładem Poissona. Do zbudowania drugiego zbioru danych skorzystano z rzeczywistych pomiarów czasu napływu zgłoszeń do obsługi [81]. Pomiary te polegały na zapisie ruchu sieciowego występującego na kilku serwerach udostępniających strony WWW oraz pliki. Każdy zestaw testowy posiadał rozmiar z zakresu od 100 000 do 800 000 krotek.

Aby zweryfikować jakość estymacji czasów odpowiedzi, symulowano przetwarzanie zapytania strumieniowego dla różnych obciążeń strumieniowej bazy danych oraz dla obu grup danych testowych. Następnie otrzymane wyniki porównano z wartościami wygenerowanymi przez modele analityczne. Przyjęto, że modele analityczne są wyliczane dla całego zbioru danych testowych. W ten sposób można zaobserwować wrażliwość modeli na rozkład danych wejściowych. Zmianę poziomu obciążenia strumieniowej bazy danych uzyskiwano poprzez dodatkowe zapytanie. Składało się one z operatora symulującego pracę przez okres zapisany w krotce wejściowej. Zmiana obciążenia sprowadzała się do sterowania intensywnością strumienia zasilającego to zapytanie. Należy zauważyć, że metoda ta umożliwia sterowanie poziomem obciążenia, jak również jego granulacją, co pozwala wiernie oddać losowość obciążenia serwera. Do wygenerowania strumienia symulującego obciążenie wykorzystano rzeczywiste pomiary opublikowane w [81].

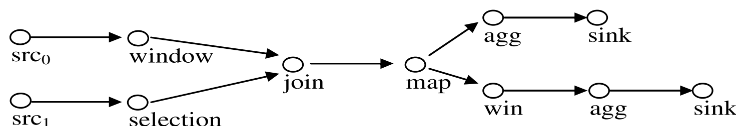
Na rysunku 4.23 oraz na rysunkach od 4.24 do 4.28 przedstawiono listę zapytań testowych od DAG1 do DAG6. Składają się one przeciętnie z dziesięciu operatorów strumieniowych. Każde z zapytań zawiera inny komponent będący objęty badaniem. Na początku zapytanie DAG1 posiada pojedynczy operator binarny wraz z dołączonymi do niego ciągami operatorów. Aby zmierzyć wpływ synchronizacji w zapytaniu DAG1 skonstruowano zapytanie referencyjne DAG2. Wyobraźmy sobie, że znamy z wyprzedzeniem krotki, jakie napłyną na jedno z wejść operatora O_4 złączenia. Wtedy dane strumienia można zastąpić tabelą o identycznej zawartości. Wówczas operator łączący dwa strumienie jest zastąpiony operatorem łączącym krotki ze strumienia z tabelą danych. Zauważmy, że taki operator nie jest obciążony zjawiskiem synchronizacji, ponieważ posiada jedno wejście strumieniowe. Po wprowadzeniu powyższych zmian do zapytania DAG1 otrzymano zapytanie DAG2. Wartości selektywności operatorów *joinStrTab* są takie same jak dla operatora *join* w zapytaniu DAG1. Aby sprawdzić na ile zmiana rozkładu intensywności danych wejściowych wpływa na dokładność modelu opartego na analizie wartości średnich stworzono zapytanie DAG3 składające się z pojedynczego ciągu operatorów. Następnym elementem poddanym testom były rozwidlenia, które mogą wystąpić w trakcie przetwarzania danych strumieniowych. W tym celu zdefiniowano zapytanie DAG4. Zapytanie DAG5 posłużyło do sprawdzenia połączeń kaskadowych operatorów

binarnych. Na koniec skonstruowano zapytanie DAG6 w sposób analogiczny jak skonstruowano zapytanie DAG2 dla zapytania DAG1.

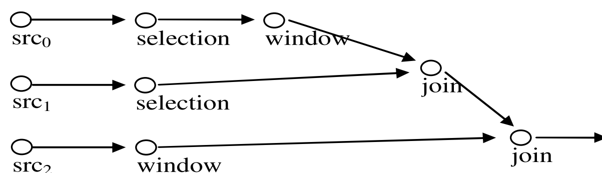


Rys. 4.24. Zapytanie DAG2

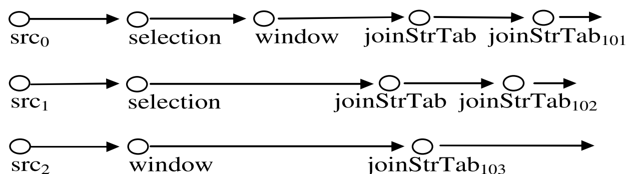
Rys. 4.25. Zapytanie DAG3



Rys. 4.26. Zapytanie DAG4



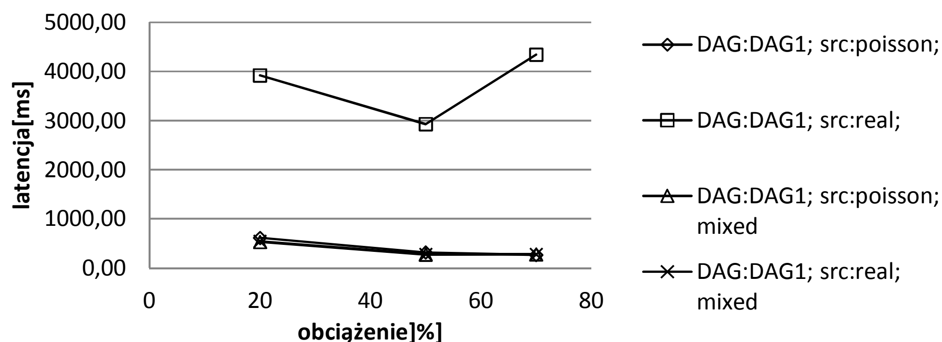
Rys. 4.27. Zapytanie DAG5



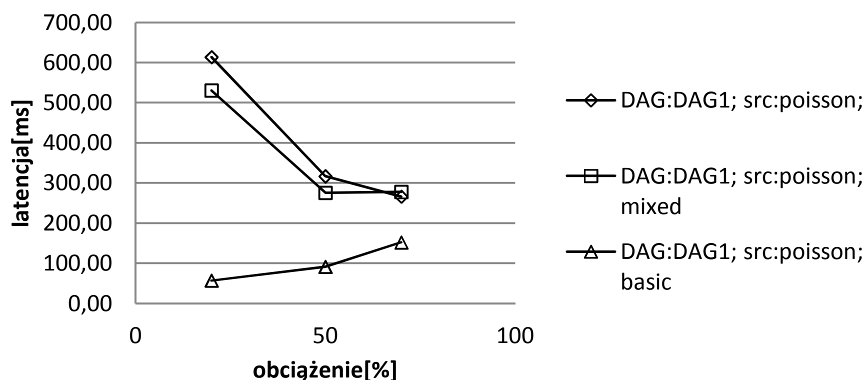
Rys. 4.28. Zapytanie DAG6

Dla każdego zapytania zebrane wyniki zilustrowano w postaci wykresu latencji krotek wynikowych względem obciążenia systemu strumieniowego. Latencja została zmierzona w milisekundach, a obciążenie podano jako procentowe wykorzystanie mocy obliczeniowej węzła. Otrzymane wyniki zaprezentowano na wykresach. Ich ostatni człon nazwy informuje o źródle pochodzenia pomiaru. Wyróżniono trzy typy źródeł wartości:

- nazwa *basic* informuje, że wartość wyliczono przy użyciu podstawowego modelu analitycznego,
- nazwa *mixed* oznacza wartości wyliczone przez rozszerzony model,
- brak etykiety informuej, że wartości zostały odczytane w trakcie symulacji.

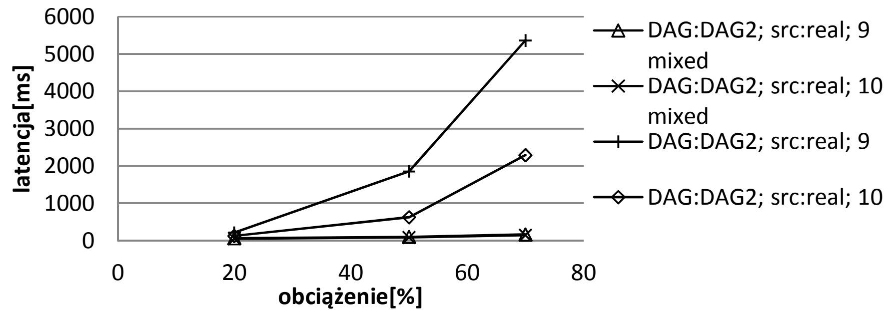


Rys. 4.29. Sprawdzenie dokładności modelu mixed dla pytania DAG1

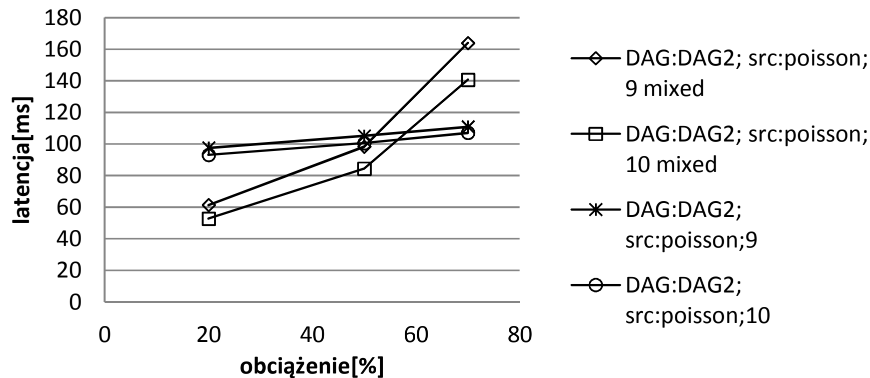


Rys. 4.30. Porównanie dokładności modeli dla danych o rozkładzie Poisson dla pytania DAG1

Na rysunku 4.29 zestawiono pomiary opóźnień dla DAG1 z wartościami wyliczonymi przez model rozszerzony. Dodatkowo na rysunku 4.30 zestawiono zmierzone czasy latencji dla danych o rozkładzie Poisson z wartościami wyznaczonymi przy użyciu modelu podstawowego i modelu rozszerzonego. Z wykresów możemy odczytać, że błąd modelu podstawowego względem wartości zmierzonej dochodzi do 90% dla danych o rozkładzie Poisson. Wprowadzenie rozszerzonego modelu pozwala zredukować ten błąd do poziomu 17% dla najgorszego przypadku. Jak widzimy na rys. 4.30 poprawa jakości jest znacząca. Słabość obu modeli pojawia się dla danych o rozkładzie rzeczywistym.

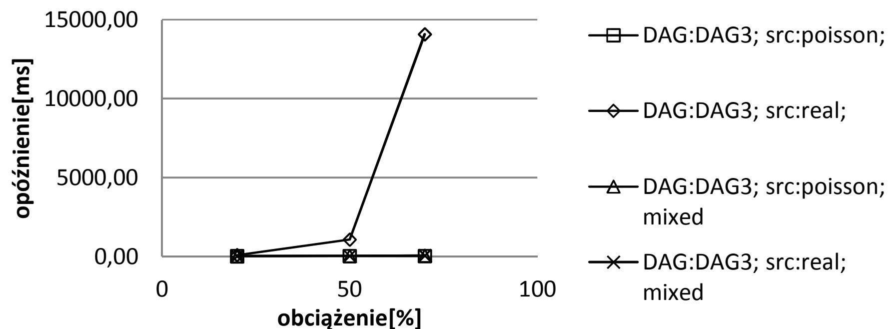


Rys. 4.31. Sprawdzenie dokładności modelu mixed dla danych o rozkładzie rzeczywistym dla pytania DAG2

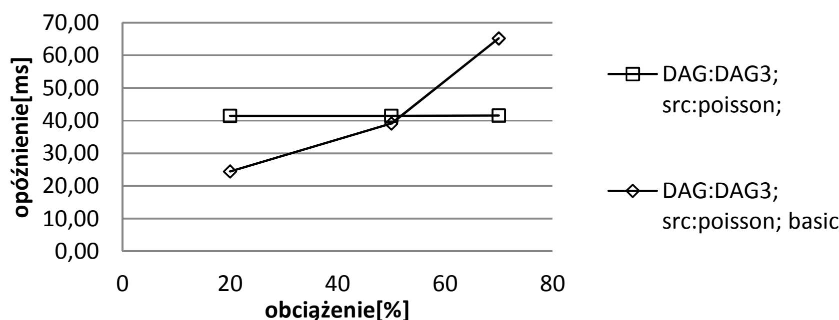


Rys. 4.32. Sprawdzenie dokładności modelu mixed dla danych o rozkładzie Poisson dla pytania DAG2

W kolejnym kroku zmierzono opóźnienia dla zapytania DAG2, w celu sprawdzenia wpływu synchronizacji. Porównanie zmierzonych wartości na rys. 4.31 i rys. 4.29 potwierdza kluczową rolę synchronizacji w kształtowaniu się wartości opóźnień dla zapytań strumieniowych. Zapytanie DAG2 pełni również rolę testu na ile model rozszerzony sprawdza się w modelowaniu opóźnień dla dwóch niezależnych pod-zapytań. Otrzymane zestawienie na rys. 4.32 dla danych o rozkładzie Poisson, potwierdza wysoką wierność estymacji.



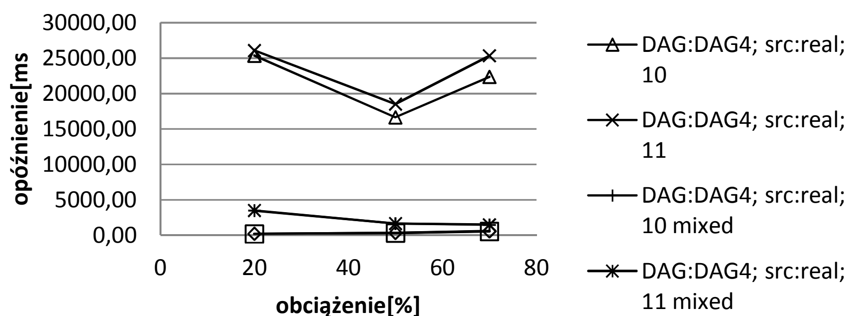
Rys. 4.33. Sprawdzenie dokładności modelu mixed dla pytania DAG3



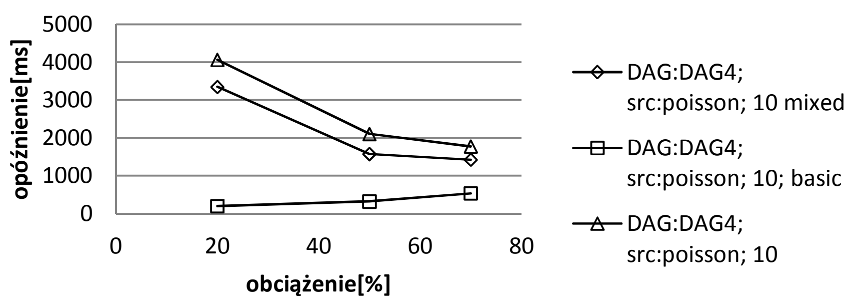
Rys. 4.34. Sprawdzenie dokładności modelu basic dla danych o rozkładzie Poisson dla pytania DAG3

Różnica pomiędzy opóźnieniami dla danych o rozkładzie Poisson a rozkładem rzeczywistym jest wysoka, dlatego przeprowadzono dodatkowo test na zapytaniu, które składa się z pojedynczego pod-zapytania bez elementów synchronizacji. W konsekwencji do opisu takiego zapytania wystarcza model w wersji podstawowej. Na rys. 4.33 widzimy, że różnica zmierzonych opóźnień w zależności od rozkładu danych wejściowych jest ogromna. Z drugiej strony dokładność estymacji dla danych o rozkładzie Poisson jest wysoka, co ilustruje rys. 4.34. Jedną z hipotez, dlaczego występuje taka rozbieżność to zbyt szerokie okno czasowe, dla którego wyliczane są wartości średnie. Parametr ten może sprawiać, że wartości średnie są odległe od wartości bieżących, co w konsekwencji prowadzi do błędów estymacji. Weryfikacja tej hipotezy jest jednym z przedmiotów dalszych badań nad tym modelem.

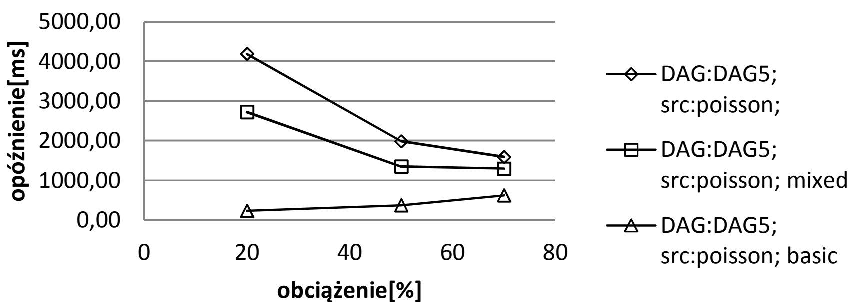
Analogicznie do przeprowadzonych dotąd pomiarów zrealizowano badania zapytań DAG4-DAG6. Podstawowy wniosek, jaki wypływa z analizy tych wykresów to fakt, że dla źródeł danych o rozkładzie Poisson estymacja wyników przez model rozszerzony jest dobra, błędy dla większości eksperymentów są niższe niż 15%. Biorąc pod uwagę jakość obecnie stosowanego modelu podstawowego postęp w jakości estymacji jest wysoki. Warto również zauważyć, że jakość estymacji jest nadal wysoka dla zapytań zawierających wiele pod-zapytań (DAG6) oraz zapytań zawierających kaskadowo ułożone operatory binarne (DAG5).



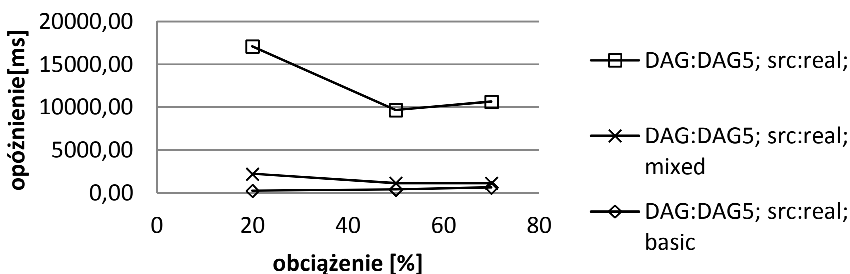
Rys. 4.35. Sprawdzenie dokładności modelu mixed dla danych o rozkładzie rzeczywistym dla pytania DAG4



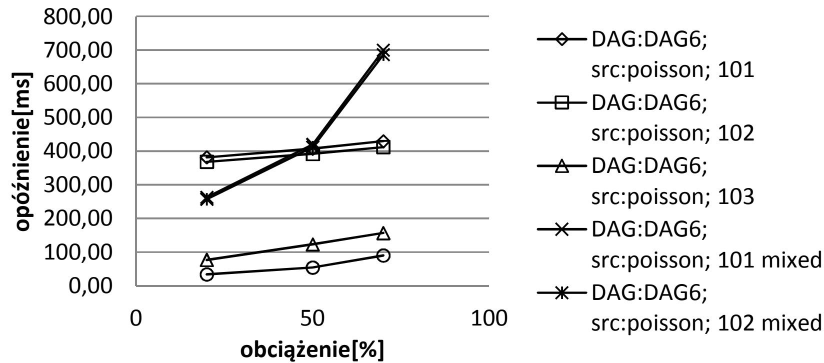
Rys. 4.36. Porównanie dokładności modeli dla danych o rozkładzie Poisson dla pytania DAG4



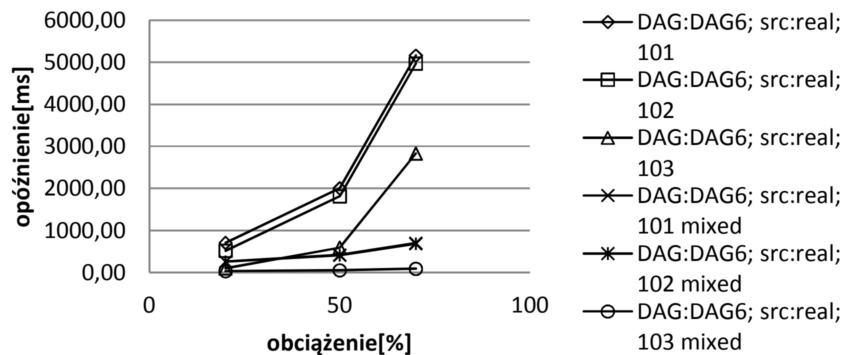
Rys. 4.37. Porównanie dokładności modeli dla danych o rozkładzie Poisson dla pytania DAG5



Rys. 4.38. Porównanie dokładności modeli dla danych o rozkładzie rzeczywistym dla pytania DAG5



Rys. 4.39. Porównanie dokładności modelu mixed dla danych o rozkładzie Poisson dla pytania DAG6



Rys. 4.40. Porównanie dokładności modelu mixed dla danych o rozkładzie rzeczywistym dla pytania DAG6

4.2.5 **Wnioski i uwagi**

Obecnie istnieje niewiele metod przewidywania wartości opóźnień w strumieniowej bazie danych. Głównie algorytmy schedulerów i optymalizatorów korzystają z podstawowego modelu przedstawionego w punkcie 4.2.2. Drugie rzadziej stosowane podejście zbudowane jest na analizie systemu kolejkowego składającego się z stanowisk M/M/1. Przykładowo, model ten zastosowano do zdefiniowania wydajności wirtualnych potoków VPipe [91]. Należy zwrócić tutaj uwagę, że artykuły opisujące optymalizatory dedykowane dla strumieniowych baz danych przedstawiają jedynie uzyskiwaną finalnie poprawę czasową lub pamięciową. Rzadko dołączana jest informacja o dokładności modelu, na którym zbudowano optymalizator.

W tej sekcji przeprowadzone testy pokazują, że pominięcie wpływu synchronizacji jest znaczącym uproszczeniem. Uzyskane błędy estymacji dla modelu

podstawowego były bardzo wysokie sięgające przeciętnie 90%, co wskazuje na jego praktyczną bezużyteczność. Otrzymane wyniki pozwalają postawić tezę, że również model M/M/1 jest obarczony dużym błędem estymacji. Wynika to z faktu, że ten model także nie uwzględnia mechanizmu synchronizacji krotek dla operatorów binarnych. Aby zmniejszyć błąd estymacji zbudowano nowy komponent, który wylicza wzrost latencji, jaki ma miejsce podczas obsługi synchronizacji. Przeprowadzone testy wskazują na znaczącą poprawę modelu estymującego czasy odpowiedzi. Zastosowanie nowego podejścia pozwala obniżyć błąd do 17% dla źródeł o rozkładzie Poisson.

Gdy rozkłady intensywności strumieni są rzeczywiste, wtedy szerokie okno czasowe dla którego wyznaczane są średnie intensywności strumieni, prowadzi do coraz większych rozbieżności pomiędzy wartością bieżącą a średnią. W konsekwencji dokładność estymacji latencji spada. Można to zaobserwować porównując wyniki otrzymywane dla rozkładu Poissona a rozkładu rzeczywistego. Wyróżniamy dwie drogi poprawy jakości estymacji. Pierwsza polega na zastąpieniu pomiaru wartości średnich, pomiarem rozkładu intensywności strumieni. Zmiana ta jednak wiąże się z utworzeniem modelu o wysokiej złożoności obliczeniowej, w konsekwencji jego wyliczanie na bieżąco byłoby ograniczone. Już samo zebranie rozkładu intensywności dla strumieni dla takiego modelu mocno obciążałoby proces przetwarzania, co czyniłoby taki model nie atrakcyjnym dla strumieniowej bazy danych. Drugie rozwiązanie pozostawiałoby model oparty na analizie wartości średnich. Poprawa precyzji estymacji polegałaby na doborze rozmiaru okna czasowego, dla którego wyliczane statystyki byłyby bliższe wartościom rzeczywistym. Pojawia się tutaj potencjalny kierunek dalszych badań obejmujący automatyzację doboru okresu pomiaru statystyk, tak aby poprawić dokładność estymacji, przy jednoczesnej minimalizacji obciążenia generowanego w trakcie wyliczania modelu kosztowego.

Zbudowany model daje nowe możliwości w rozwiązaniu dwóch problemów strumieniowych baz danych. Grupa algorytmów do której zaliczamy między innymi schedulery [10] i algorytmy zarządzające płynnością przetwarzania[79,54] wymagają skonfigurowania parametrów związanych z czasem przetwarzania zapytań. Rozwiązania te jednak nie podają algorytmu, jak dobrać te parametry optymalizacji. Wysoka dokładność modelu rozszerzonego jest potencjalnym punktem wyjścia dla

przyszłych badań mający na celu zaadaptowanie tego modelu do automatyzacji doboru parametrów optymalizacji powyższych algorytmów.

Innym otwartym problemem w strumieniowych bazach danych jest zagadnienie obsługi pik obciążeń. Przykładowo algorytmy klasy load shedding przystępują do działania w chwili, gdy nastąpi przeciążenie. Należy zauważyć, że mechanizm synchronizacji jest głównym źródłem powstawania paczek krotek w strumieniach, a zatem pik obciążania. Zjawisko to jest dobrze modelowane przy użyciu skonstruowanego modelu rozszerzonego. Pojawia się tutaj kolejny potencjalny temat dalszych badań: czy w oparciu o ten model można zidentyfikować fragmenty zapytań, w których ryzyko powstawania przeciążeń jest wysokie. Następnie przeciwdziałać temu, zamiast czekać na powstanie problemu, tak jak to realizują obecne algorytmy.

4.3 Podział operatorów na partycje

Wzrastająca liczba danych przetwarzana przez strumieniowe bazy danych skutkuje potrzebą rozwiązań, które coraz lepiej korzystają z przetwarzania współbieżnego. Kluczową rolę odgrywa tutaj rozwój architektur silników strumieniowych [23,5], schedulerów [22,9,76], reorganizacja operatorów [18,13], zastosowanie load sheddingu [86] oraz optymalizacja na poziomie pojedynczych operatorów strumieniowych[45,42,69,68]. Mechanizmy te wpływają na siebie nawzajem. Przykładowo algorytm schedulera, reorganizacja połączeń pomiędzy operatorami oraz load shedding są ze sobą ściśle związane, z drugiej strony stan literatury wskazuje na to, że badania nad nimi prowadzone są jak nad rozwiązaniami samodzielnymi.

W tej sekcji zbadane zostanie nowe podejście do integracji optymalizatorów partycji operatorów i schedulerów. Połączenie obu mechanizmów zaproponowano pierwszy raz w [1], gdzie wprowadzono pojęcie pociągów krotek. Ich rolą jest redukcja liczby operatorów strumieniowych, co pozwala zmniejszyć udział czasu pracy schedulera w systemie. Później zaproponowano radykalne rozwiązanie [12], gdzie operatory zapytania strumieniowego są uruchamiane w kolejności, która usuwa potrzebę stosowania strumieni pomiędzy operatorami wewnętrznymi. Podejście

takie jednak nie można zrealizować w środowisku rozproszonym lub wielowątkowym, co stanowi jego główną wadę. Obecnie nowoczesną metodę budowy partycji przedstawiono w [13]. Zdefiniowano tam zestaw reguł decydujących, jakie operatory należy połączyć w partycje biorąc pod uwagę kryterium minimalizacji czasu odpowiedzi lub zapotrzebowania pamięciowego. Słabością tego rozwiązania są reguły, które pozwalają zminimalizować tylko dwie topologie operatorów.

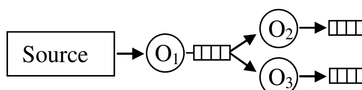
Zaproponowany w tej sekcji optymalizator partycji jest kontynuacją badań nad rozwiązaniem [13]. Wprowadzono do niego nowatorską metodę integrującą scheduler z optymalizatorem partycji. Dzięki temu, zbudowane reguły tworzenia partycji nie ograniczają się do dwóch topologii. Drugą zaletą wprowadzonej integracji jest uzyskanie efektu synergii, dzięki któremu można obniżyć zarówno czas odpowiedzi jak również obciążenie pamięciowe, znacznie wydajniej niż bez integracji obu mechanizmów.

4.3.1 Optymalizacja partycji

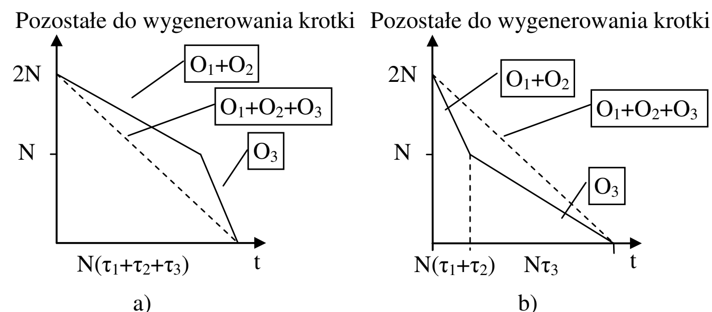
W punkcie tym streszczone zostaną istniejące algorytmy tworzące partycje operatorów minimalizujące zapotrzebowanie pamięciowe lub czasy odpowiedzi. Dokładny opis powyższych algorytmów można odnaleźć w [1,13].

Minimalizacja ORT (Output Response Time)

Na rys. 4.41 przedstawiono tzw. topologię widelca oznacza ona, że jedna krotka jest współdzielona przez kilka operatorów. Przyjmijmy, że każdy z operatorów posiada selektywność 1 a czasy przetwarzania krotek wynoszą odpowiednio τ_1, τ_2 i τ_3 dla operatorów O_1, O_2 i O_3. Aby osiągnąć minimalizację ORT należy wybierać ten operator, dla którego szybkość przetwarzania krotek jest największa. Jeżeli $\tau_2<\tau_3$, wówczas operator O_1 powinien najpierw przekazywać krotki operatorowi O_2. Krotki mogą być tutaj przekazywane albo w sposób bezpośredni albo pośredni.



Rys. 4.41. Współdzielenie krotek z jednego operatora *O₁*



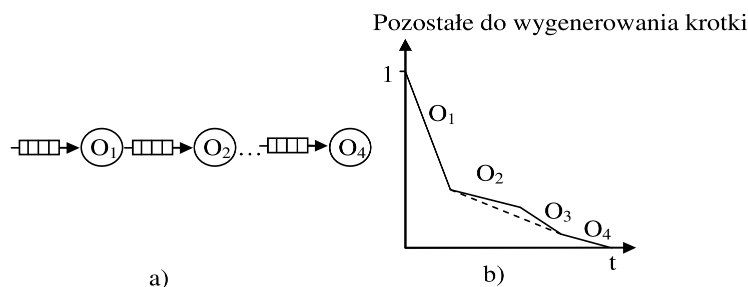
Rys. 4.42. Wykresy finalizacji przetwarzania dla przykładu z rys. 4.41

Przeanalizujmy rys. 4.42 aby zdefiniować optymalny sposób przekazywania krotek. Pojedyncza linia na wykresie przedstawia ile w zadanym czasie zostanie wygenerowanych krotek. Na rysunku 4.42 a) mamy linię reprezentującą operatory O_1 i O_2 pracujące w ramach jednej partycji oraz działający oddzielnie operator O_3. Dla tej konfiguracji pole powierzchni pod wykresem podzielone przez $N(\tau_1+\tau_2+\tau_3)$ daje średnią liczbę krotek przebywających w systemie. Chcąc poprawić działanie systemu strumieniowego należy zdefiniować takie partycje, które najefektywniej minimalizują pole powierzchni pod wykresem. Dla analizowanego przykładu wyróżniamy dwa przypadki:

1) Jeżeli $\tau_3<\tau_1+\tau_2$, wówczas wszystkie operatory włączamy do jednej partycji. Wygenerowaną krotkę przez operator O_1 przekazujemy wpierw operatorowi O_2 a następnie O_3. Powyższy scenariusz jest zilustrowany na rys. 4.42 a), gdzie nachylenie linii przerywanej wskazuje średnią szybkość generowania krotek wynikowych dla nowej partycji.
2) Jeżeli $\tau_3>\tau_1+\tau_2$, wtedy powinniśmy utworzyć partycje składającą się wyłącznie z operatorów O_1 i O_2. Przypadek ten ilustruje rys. 4.42 b).

Z analogiczną sytuacją mamy do czynienia, gdy operatory tworzą prosty szereg jak na rys. 4.43. Przyjmijmy, że operator O_1 przetworzył jedną krotkę. W wyniku tego na wyjściu operatora O_1 pojawi się s_1 krotek po upływie τ_1 czasu, na wyjściu O_2 pojawi się s_1*s_2 krotek po upływie $s_1*\ \tau_2$, itd. Na rys. 4.43 b) przedstawiono zmianę liczności krotek na wyjściach operatorów O_1, O_2, O_3 i O_4 w kolejnych momentach czasu. Wyznaczając pole pod wykresem na rys. 4.43 b) a następnie dzieląc go przez czas przetwarzania, otrzymujemy średnią liczbę krotek w systemie.

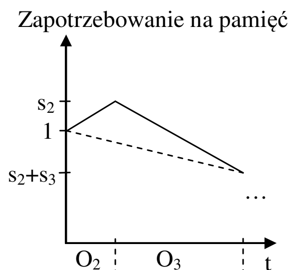
Tworząc partycję składającą się z operatorów (O_2+O_3) sprawiamy, że powstały operator złożony ma na wejściu tyle krotek co operator O_2, oraz generuje tyle krotek co operator O_3 w czasie łączącym pracę operatorów O_2 i O_3. Finalnie, średnia liczba krotek przebywająca w systemie po utworzeniu takiej partycji jest mniejsza, niż kolejkując oddzielnie operatory O_2 i O_3. Widzimy to na rys. 4.43 b), gdzie pole powierzchni pod krzywą przerywaną jest mniejsze niż pole pod krzywą ciągłą.



Rys. 4.43. Zapytanie składające się z prostego ciągu operatorów

Minimalizacja zapotrzebowania na pamięć

Przeanalizujmy ponownie graf zapytania na rys. 4.41. Operatory O_2 i O_3 są zasilane tym samym strumieniem, w wyniku czego nie możemy ich analizować oddzielnie. Krotka *t* jest usuwana z systemu, gdy zostanie przetworzona przez operator O_2 i O_3. Po przetworzeniu krotki przez operator O_2, niezależnie od jej selektywności w systemie wzrasta zapotrzebowanie pamięciowe. Dopiero, gdy operator O_3 przetworzy krotkę wejściową, wówczas współdzielona krotka może zostać usunięta.



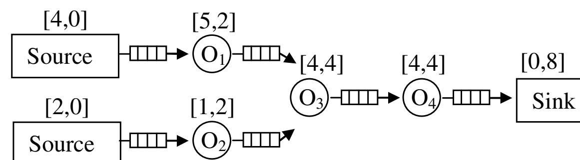
Rys. 4.44. Wykres zapotrzebowania na pamięć

Przyjmijmy, że na rys. 4.41 operatory O_2 i O_3 posiadają selektywności równe s_2 i s_3 oraz rozmiary krotek są identyczne na wyjściach operatorów O_1, O_2 i O_3. Wtedy po przetworzeniu krotki na wejściu operatora O_2 w systemie wzrasta zużycie pamięci do wartości: $1 + s_2$, co przedstawia rys. 4.44. Po przetworzeniu krotki na wejściu

operatora O_3, krotka ze strumienia wejściowego zostaje usunięta i całkowite zapotrzebowanie pamięciowe w systemie wyniesie: $s_2 + s_3$. Podobnie tutaj celem optymalizacji jest zmniejszenie pola powierzchni pod wykresem. Aby to osiągnąć uzasadnione jest stworzenie partycji: $(O_1+O_2+O_3)$, co ilustruje rys. 4.44.

4.3.2 Rozszerzone rozwiązanie

Istniejące optymalizatory partycji operatorów nie uwzględniają połączeń zawierających operatory binarne lub n-arne. Wadę tą ilustruje rys. 4.45. Partycje rozpoczynające się od operatora binarnego, tak jak dla przykładu (O_3+O_4), nie mogą być przeanalizowane przez istniejące optymalizatory, ponieważ selektywność oraz rozmiar krotek wejściowych zależy od tego, które wejście operatora O_3 zostanie wpierw przetworzone. Analogiczne ograniczenie zachodzi dla partycji (O_1+O_3), gdzie wydajność optymalizacji zależy od porządku, w jakim uruchamiane są operatory O_1 i O_2. Zauważmy, że przedstawione algorytmy budowy partycji są niezależne od wyboru algorytmu schedulera. Jeżeli nowoutworzona partycja otrzyma niski priorytet od schedulera, wówczas wprowadzona przez nią optymalizacja będzie osłabiana. Z drugiej strony, tworzenie partycji zmniejsza liczbę elementów obsługiwanych przez scheduler, co prowadzi do obniżenia jego udziału w zarządzaniu jakością przetwarzania.



Rys. 4.45. Zapytanie zawierające operator wielowejściowy

Zaproponowane rozwiązanie korzysta z obserwacji, że rekonfiguracja partycji jest realizowana rzadko z powodu wprowadzanego obciążenia obliczeniowego. Sprawia to, że celem optymalizatora jest wyszukanie długoterminowej konfiguracji partycji. Ta własność uzasadnia zastosowanie pomiaru średniej wartości intensywności strumieni. Przeciwieństwem są schedulery, od których wymaga się dopasowywania do krótkookresowych zmian. Przyjmuje się, że intensywność strumieni w strumieniowych bazach danych jest zmienna, dlatego algorytmy schedulerów ograniczają wykorzystanie parametrów związanych

z pomiarem intensywności strumieni, ponieważ wymuszałoby to częste aktualizowanie statystyk.

Przedstawione w tym punkcie algorytmy zakładają, że dla każdego operatora O_i strumieniowa baza danych aktualizuje i udostępnia następujące statystyki: selektywność s_i, średni czas przetwarzania krotki przez operator τ_i oraz intensywność generowania krotek przez operator u_i. Znając te wartości można wyznaczyć dla każdego operatora średnią liczbę krotek wypływających w jednostce czasu n_i oraz czas przetwarznia krotek wejściowych t_i. Przykładowe wartości tych statystyk naniesiono na rys. 4.45 w konwencji [n_i, t_i].

Minimalizacja ORT

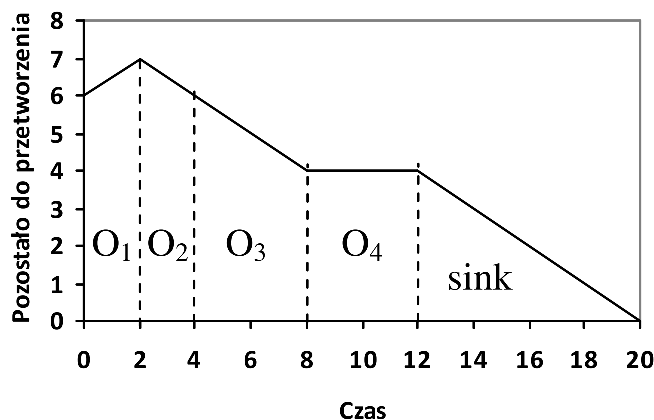
Aby zweryfikować czy partycja składająca się z O_4 i *sink* zmniejsza opóźnienie, wyznaczymy pole pod krzywą liczby krotek w systemie względem czasu. Dla wersji bez partycji, pole powierzchni składa się z trapezu reprezentującego operator O_4, gdzie na wejściu są 4 krotki na wyjściu również 4 krotki przetworzone w czasie 4ms, co daje: 0.5*(4+4)*4=16; dla *sink*: 0.5*(4+0)*8=16. W sumie pole powierzchni dla obu wynosi 32. Jeżeli wprowadzimy partycję (O_4+*sink*) wówczas pole będzie wynosiło: 0.5*(4+0)*12=24, a zatem utworzenie partycji pozwala osiągnąć poprawę wydajności. Analogiczne rozwiązanie zostało przedstawione w punkcie 5.3.1.1. Różnica polega na liniowym przeskalowaniu wartości n_i i t_i, ponieważ tam założono, że na wejściu operatora O_4 pojawia się jedna krotka.

Sprawdźmy czy partycja (O_3+O_4) skraca czasy odpowiedzi. W oparciu o wartości t_i i n_i wyznaczamy pole powierzchni dla wersji bez partycji: 0.5*(6+4)*4+0.5*(4+4)*4=36 oraz pole powierzchni dla nowej partycji: 0.5*(6+4)*(4+4)=40. Otrzymane wartości wskazują brak poprawy wydajności po wprowadzeniu partycji. Zauważmy, że operator O_3 w zależności od tego, z którego wejścia przetwarza krotki może mieć inną selektywność oraz inny koszt przetwarzania. Cecha ta nie pozwala zastosować metodę przedstawioną w punkcie 5.3.1.1.

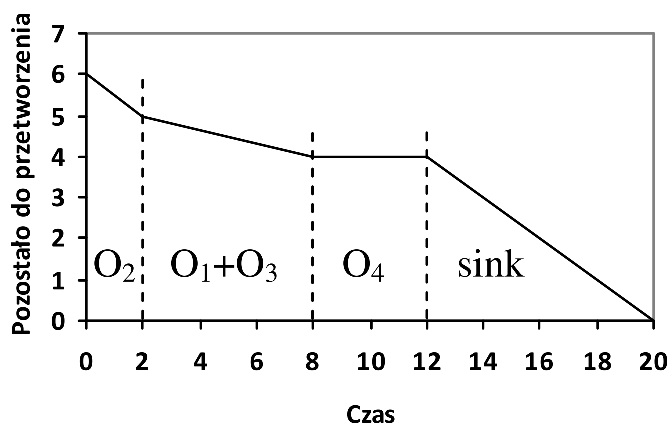
Rozważmy teraz partycję: (O_2+O_3). Jest to konfiguracja operatorów, w której nie można pominąć wpływu operatora O_1, ponieważ liczba wygenerowanych krotek przez operator O_3 zależy od O_1 i O_2. Aby rozwiązać ten problem przyjęto założenie, że

strategia schedulera opiera się na priorytetach operatorów. Aby zachować ogólność rozwiązania korzystamy tylko z informacji, w jakiej kolejności priorytety porządkują operatory. Dla rozważanego przykładu niech będzie to porządek: $O_1 \succ O_2 \succ O_3 \succ O_4 \succ sink$. Wcześniej analiza wydajności partycji obejmowała tylko wchodzące w jej skład operatory. W rozwiązaniu rozszerzonym analizowany jest wpływ partycji na całe zapytanie. Aby to osiągnąć, na początku wszystkie strumienie przyłączone do źródeł są oznaczone, jako gotowe do odczytu oraz wyliczana jest liczba przebywających w nich krotek. Dla zapytania z rys. 4.45 wynosi ona: 4+2=6. Następnie wybierany jest operator O_i o najwyższym priorytecie, który posiada wszystkie strumienie wejściowe w stanie gotowym do odczytu. Po przeanalizowaniu operatora O_i oznaczamy jego strumienie wyjściowe, jako gotowe do odczytu. Operacje te powtarzamy do momentu przeanalizowania wszystkich operatorów. Dla zapytania na rys. 4.45 otrzymujemy następujący porządek analizy operatorów: O_1, O_2, O_3, O_4 i *sink*. Podczas analizy pojedynczego operatora przesuwamy się na osi czasu o wartość t_i oraz aktualizujemy liczbę krotek w systemie. Ujścia są tutaj traktowane, jako operatory niegenerujące krotek. W oparciu o powyższy algorytm powstaje wykres przedstawiony na rys. 4.46. Zaprezentowany algorytm w przeciwieństwie do rozwiązań konkurencyjnych umożliwia analizę dowolnej konfiguracji operatorów. Istotną rolę odgrywa znajomość priorytetów operatorów. W oparciu o nią budowana jest sekwencja przybliżająca kolejność uruchamiania operatorów, gdy system nie jest przeciążony. Bez znajomości priorytetów operatorów, nie można ustalić czy preferowane jest uruchomienie wpierw operatora O_1 a potem O_2 lub na odwrót. To z kolei jest kluczowe podczas analizy operatorów binarnych i n-arnych.

Powróćmy do pytania, czy partycja (O_1+O_3) poprawia wydajność ORT. Przyjmijmy, że po wprowadzeniu tej zmiany układ priorytetów jest następujący: $(O_1 + O_3) \succ O_2 \succ O_4 \succ sink$, a zatem kolejność analizy operatorów wynosi: O_2, (O_1+O_3), O_4 i *sink*.

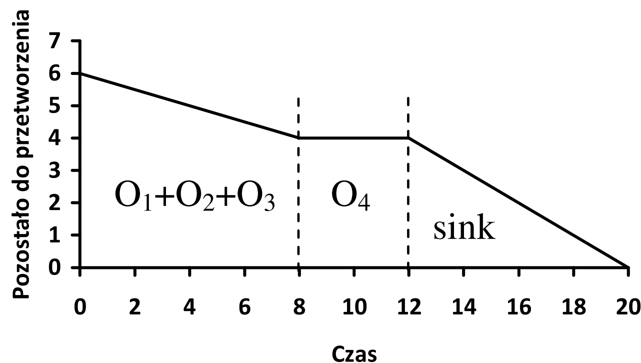


Rys. 4.46. Wykres liczby krotek w systemie w funkcji czasu



Rys. 4.47. Wykres liczby krotek w systemie w funkcji czasu po wprowadzeniu partycji O_1+O_3

Po wyliczeniu pola powierzchni dla rozwiązania pierwotnego otrzymujemy: 78, podczas gdy dla rozwiązania z partycją: 70; a zatem partycja (O_1+O_3) minimalizuje średnią liczbę krotek w systemie. Ilustracją dla tego przypadku jest rys. 4.47. Zaproponowany nowy optymalizator partycji pozwala sprawdzać dowolną konfigurację operatorów. Przykładowo dla partycji (O_1+O_2+O_3) wykres zmiany liczby krotek w funkcji czasu przestawiono na rys. 4.48, gdzie pole pod wykresem wynosi: 72. Różnica pomiędzy wartością metryki przed wprowadzeniem partycji a po nazywamy efektywność partycji, dla powyższego przykładu wynosi ona: *78–72=6*. Podsumowując, najlepszą partycją spośród sprawdzonych konfiguracji jest (O_1+O_3).



Rys. 4.48. Wykres zmiany liczby krotek w systemie po wprowadzeniu partycji O_1+O_2+O_3

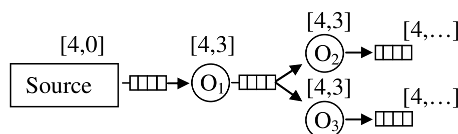
Optymalizacja pamięciowa

Minimalizacja obciążenia pamięciowego podobnie jak optymalizacja ORT polega na odnalezieniu konfiguracji partycji, która zapewnia najmniejsze pole powierzchni tym razem pod wykresem zapotrzebowania pamięciowego. W odniesieniu do optymalizacji ORT konieczna jest dodatkowo wspólna analiza operatorów współdzielących jeden strumień jak zasygnalizowano w punkcie 5.3.1.2.

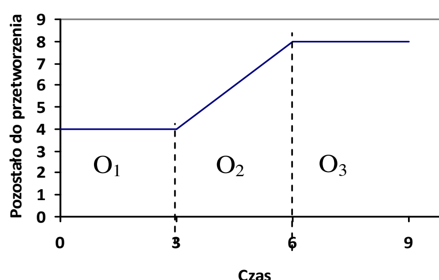
Działanie optymalizatora składa się z następujących etapów. Na początku algorytm oznacza wszystkie strumienie przyłączone do źródeł, jako gotowe do odczytu oraz wylicza rozmiar pamięci v zajęty przez te strumienie. Dla zapytania z rys. 4.49 wynosi on: 4. Następnie wybierany jest operator O_i o najwyższym priorytecie, który posiada wszystkie strumienie wejściowe w stanie gotowym do odczytu. W kolejnym kroku każdy ze strumieni wejściowych $S_{k,i}$ łączący operator poprzedzający k z operatorem i jest oznaczony jako odczytany. Strumień wyjściowy operatora k może być współdzielony przez wiele operatów. Jeżeli w wyniku odczytu strumienia $S_{k,i}$ wszystkie strumienie wyjściowe operatora O_k są odczytane, wówczas aktualizowana jest wartość $v = v - b_k$. Następnie dodawany jest wpływ operatora O_i: $v = v+b_i$. Na koniec strumienie wypływające z O_i są oznaczane jako gotowe do odczytu. Powyższy algorytm jest powtarzany, aż do przeanalizowania wszystkich operatorów.

Działanie optymalizatora ilustruje rys. 4.49, gdzie do każdego operatora dołączono parę przykładowych wartości [b_i, t_i]. Wartość b_i oznacza rozmiar pamięci zajęty przez wygenerowane krotki, a t_i reprezentuje czas przetwarzania krotek wejściowych. Ponadto przyjęto, że operatory ułożone zgodnie z priorytetami tworzą ciąg: $O_1 \succ O_2 \succ O_3$. Na początku wartość v wynosi 4. Następnie analizowany jest

operator O_1, który zwalnia 4 jednostki pamięci i zajmuje 4 jednostki. W kolejnym kroku analizowany jest operator O_2, ponieważ jest kolejnym z najwyższym priorytetem i gotowym do odczytania strumieniem. Na koniec obsługiwany jest operator O_3.



Rys. 4.49. Fragment zapytania przeznaczony do optymalizacji pamięciowej



Rys. 4.50. Wykres obciążenia pamięciowego w funkcji czasu

4.3.3 Algorytmy tworzenia partycji

Optymalizacja jednokryterialna

Optymalizator partycji korzysta z strategii zachłannej. Przyjęto, że na początku zapytanie składa się z jedno-operatorowych partycji, które w kolejnych iteracjach są złączane. Budowa optymalizatora dopuszcza zastosowanie różnych metryk, przy czym metryka składa się z dwóch funkcji. Pierwsza sprawdza czy podana partycja jest obsługiwana. Przykładowo metryki podane w punkcie 5.3.1 dopuszczają tylko prosty łańcuch operatorów lub topologie widelca. Druga funkcja wylicza wartość metryki dla zadanej partycji. Dokładny algorytm podano poniżej.

1) Dla każdej partycji E_i dla zapytania Q, budowana jest potencjalna nowa partycja składająca się z E_i oraz partycji zasilanych przez E_i. Gwarantuje to, że strumienie współdzielone przez kilka operatorów nie są rozpięte pomiędzy kilkoma partycjami. Po zakończeniu tego korku powstaje zbiór potencjalnych nowych partycji.
2) Usuwane są partycje, które nie są wspierane przez zastosowaną metrykę.

3) Następnie wyznaczana jest efektywność dla każdej potencjalnej nowej partycji.
4) Wybierana jest partycja z największą i nie ujemną wartością efektywności.
5) Jeżeli w ostatniej iteracji wprowadzono do zapytania nową partycję lub nie została osiągnięta zadana liczba iteracji *N* algorytm powraca do punktu 1).

Optymalizacja dwukryterialna

Aby w zrównoważony sposób obniżyć zapotrzebowanie pamięciowe i czasowe skonstruowano optymalizator dwukryterialny, który wyszukuje niezdominowane rozwiązania spośród potencjalnych nowych partycji. Dodatkowo wprowadzono ograniczenie dopuszczające tylko partycje optymalizujące oba kryteria jednocześnie. Podobnie jak w optymalizatorze jednokryterialnym, dobór metryk optymalizacji jest konfigurowalny. Pełen opis algorytmu podano poniżej:

1) Dla każdej partycji E_i dla zapytania Q, budowana jest potencjalna nowa partycja składająca się z E_i oraz partycji zasilanych przez E_i.
2) Usuwane są partycje, które nie są wspierane przez dowolną z obu metryk.
3) Wyznaczana jest efektywność partycji ze względu na obie metryki.
4) Ze zbioru potencjalnych nowych partycji odrzucane są rozwiązania zdominowane.
5) Ze zbioru potencjalnych nowych partycji odrzucane są rozwiązania nieoptymalizujące obu kryteriów jednocześnie.
6) Zbiór potencjalnych nowych partycji jest wprowadzany do pytania Q z uwzględnieniem poniższego przypadku. Przyjmijmy, że mamy trzy operatory tworzące prosty ciąg: O_1, O_2 i O_3. Może dojść do sytuacji, że w zbiorze znajdują się dwie partycje. Jedna partycja E_1 wprowadza połączenie: (O_1+O_2); a druga E_2 połączenie (O_2+O_3). W tym wypadku po wprowadzeniu E_1, E_2 jest pomijana, ponieważ O_2 już nie jest dostępny.
7) Jeżeli w ostatniej iteracji zbiór potencjalnych nowych partycji nie był pusty lub nie została osiągnięta zadana liczba iteracji algorytm powraca do punktu 1).

4.3.4 Testy

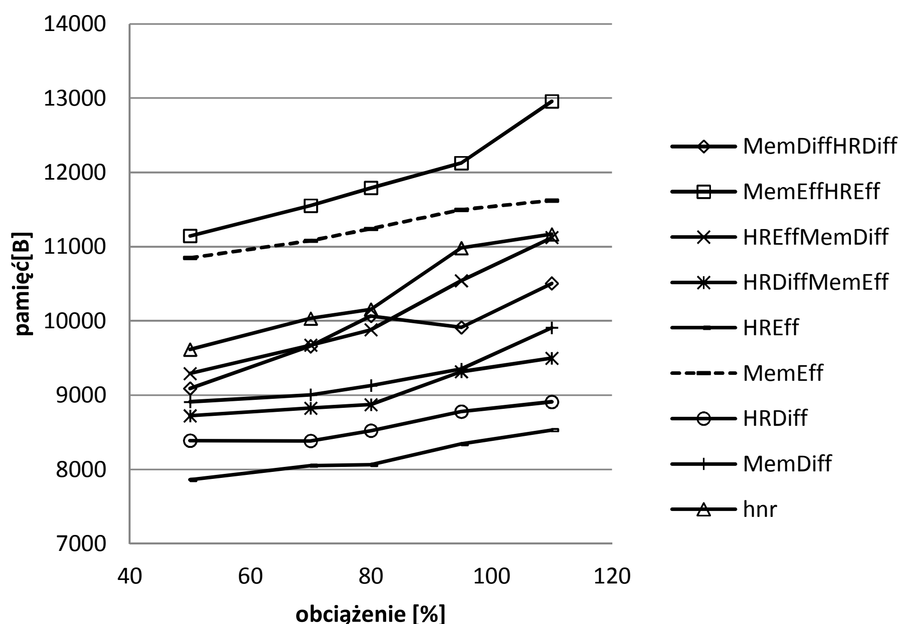
Aby uzyskać precyzyjne pomiary obciążenia pamięciowego oraz opóźnień, eksperymenty przeprowadzono na symulatorze opisanym w punkcie 5.2.5. Wzorując

się na środowisku testowym NEXMark [89,66] zdefiniowano test składający się z 32 operatorów. NEXMark zawiera szereg zapytań przetwarzających dane giełdowe. Analizując go od strony połączeń pomiędzy operatorami stworzono test składający się z najczęściej pojawiających się topologii połączeń operatorów. Aby przeprowadzone badania uwydatniły zmiany wydajności pomiędzy różnymi konfiguracjami optymalizatorów uruchomiono jednocześnie 5 kopi testu. Do wygenerowania danych testowych dla każdej kopi użyto rozkłady intensywności zebrane podczas monitoringu pracy rzeczywistych serwerów [81]. Podsumowując, badania zostały przeprowadzone w środowisku składającym się z ponad 150 operatorów, z zastosowaniem zbioru danych, którego rozkład intensywności jest oparty na pomiarach rzeczywistych.

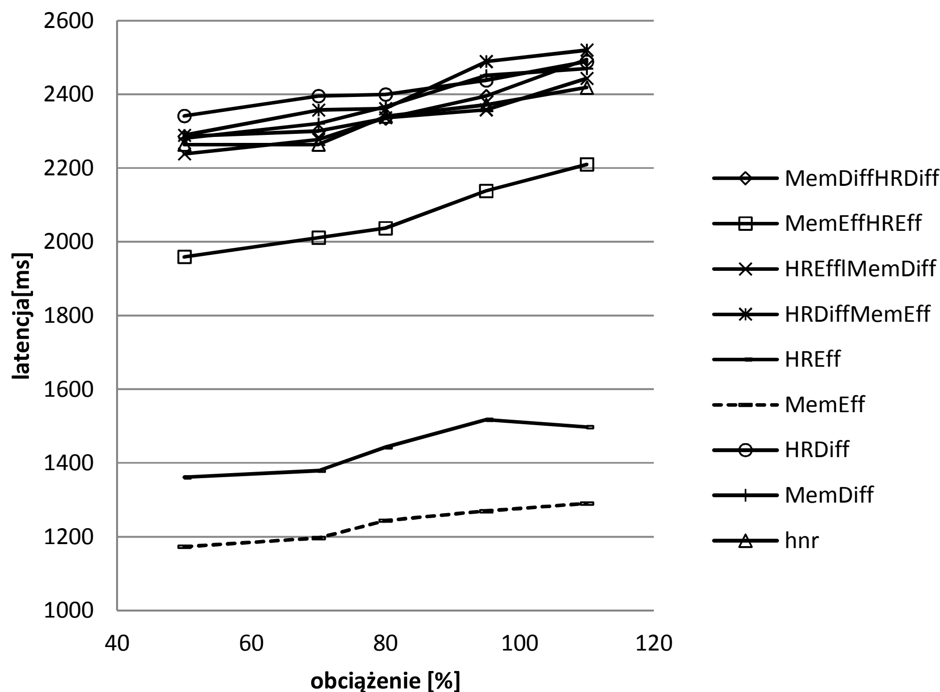
Konfiguracja pojedynczego eksperymentu składała się z ustawienia: obciążenia systemu i algorytmu optymalizacji partycji. Sterowanie obciążeniem systemu zostało zrealizowane dwuetapowo analogicznie jak w punkcie 4.2.4. Przyjęto, że w trakcie eksperymentu uruchamiane jest testowe zapytanie oraz zapytanie wytwarzające dodatkowe obciążenie. Parametry s_i i c_i dla każdego operatora są znane, dlatego możliwe było przeskalowanie czasu napływu danych, aby otrzymać obciążenia CPU na poziomie 50%. Zapytanie wytwarzające dodatkowe obciążenie było zasilane strumieniem, którego krotki definiują czas, przez jaki są obsługiwane. Pomiar wydajności optymalizacji dla zadanej konfiguracji metryk składał się z serii eksperymentów z różnymi poziomami obciążenia. Wszystkie eksperymenty zostały przeprowadzone z zastosowaniem schedulera HNR[76], który jest jednym z wydajniejszych stosowanych w strumieniowych bazach danych. Dla uproszczenia porównań nazewnictwo eksperymentów usystematyzowano. Nazwa każdego eksperymentu składa się z <*kryterium*><*typ metryki optymalizacji*>. Wyróżniamy dwa kryteria: HR i Mem, które reprezentują odpowiednio minimalizację czasu odpowiedzi i minimalizację obciążenia pamięciowego. Istnieją także dwa typy metryk: Diff i Eff. Pierwsza oznacza grupę metryk opisaną w punkcie 4.3.1. Drugi typ odnosi się do metryk wprowadzonych w punkcie 4.3.2. Jako przykład, krzywa o nazwie HRDiff reprezentuje wyniki zgromadzone dla optymalizacji czasu odpowiedzi przy użyciu metryki opisanej w sekcji 4.3.1.

Celem opracowania zebranych wyników jest sprawdzenie wydajności minimalizacji obciążenia pamięciowego i minimalizacji czasów odpowiedzi.

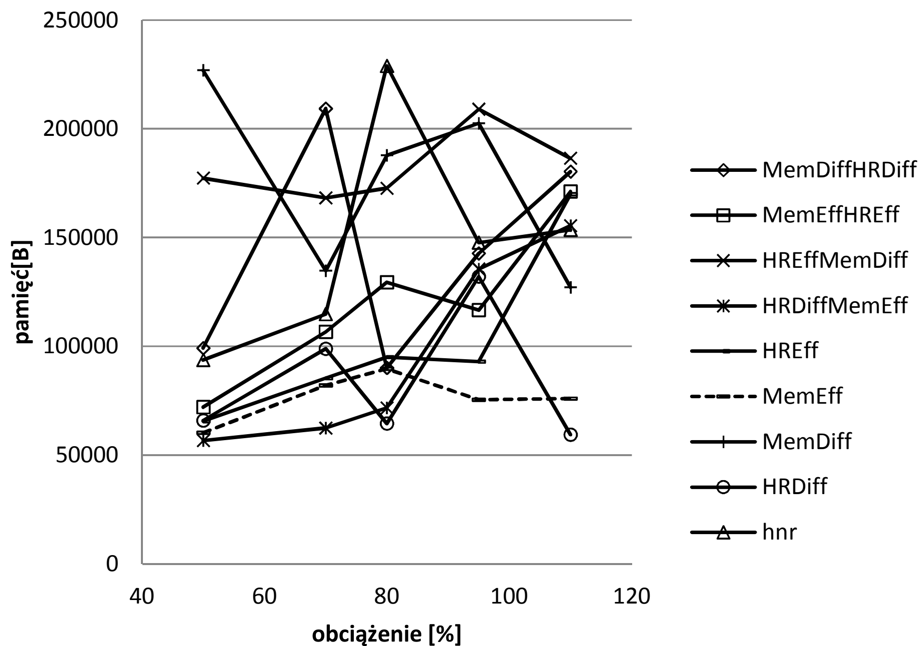
W literaturze badania eksperymentalne koncentrują się na pomiarze średniego czasu odpowiedzi lub średniego obciążenia pamięciowego, pomimo że obie wartości średnie mogą zostać łatwo przekłamane przez wartości skrajne. Dlatego pomiar średnich zastąpiono medianą i wartością maksymalną. Wartości maksymalne pozwalają ustalić na ile uzyskane rozwiązanie jest stabilne. Z kolei mediana daje wgląd w przeciętną wydajność optymalizacji. W zebranych pomiarach może zaskoczyć zmierzone zapotrzebowanie pamięciowe, które sięga 200kB. Wartość taka jest wynikiem zastosowanej metody pomiarowej oraz tego, że w NEXMark krotki składają się tylko z kilku atrybutów. Przyjmuje się, że w systemach strumieniowych zmiany zapotrzebowania na pamięć aproksymuje się poprzez pamięć zajmowaną przez krotki w strumieniach, ponieważ system ten nie przechowuje trwale danych. Na rozmiar krotki składa się rozmiar danych, rozmiar struktury tworzącej krotkę oraz rozmiar struktur reprezentujących strumienie. Aby uzyskać pomiar uniezależniony od implementacji krotek i strumieni ograniczono się do pomiaru rozmiaru danych, co umożliwia porównać wydajności konfiguracji pod kątem zapotrzebowania pamięciowego a z drugiej strony abstrahuje od wyboru języka programowania i jakości implementacji.



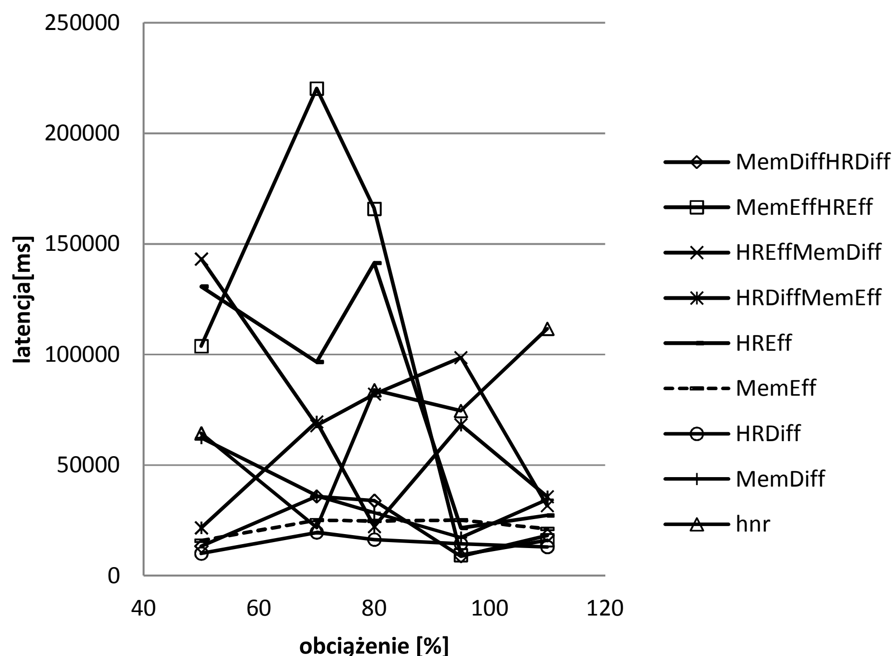
Rys. 4.51. Obciążenie pamięciowe w funkcji obciążenia systemu



Rys. 4.52. Czasy latencji w funkcji obciążenia systemu



Rys. 4.53. Maksymalne wartości obciążenia pamięciowego



Rys. 4.54. Maksymalne wartości latencji

Na początku przeprowadzono eksperyment bez użycia podziału na partycje, jego wyniki zostały umieszczone na wykresach o nazwach HNR. Stanowi on punkt odniesienia dla osiąganych wydajności po zastosowaniu optymalizacji. Zebrane pomiary pokazują, że dużą rolę odgrywa współpraca pomiędzy schedulerem a optymalizatorem partycji. Zaproponowane nowe metryki MemEff i HREff pozwoliły osiągnąć najkrótsze czasy odpowiedzi w odniesieniu do rozwiązań konkurencyjnych: HNR, HRDiff i MemDiff. Odnotowana poprawa jest rzędu 30%, co obrazuje rys. 4.52. Z perspektywy minimalizacji obciążenia pamięciowego rozwiązanie HREff jest lepsze w odniesieniu do rozwiązania bazowego, a MemEff gorsze co przedstawia rys. 4.51. Przypadek ten jest zaskakujący, ponieważ zastosowanie metryki minimalizującej czas odpowiedzi dało rezultat lepszy od metryki przeznaczonej do minimalizacji obciążenia pamięciowego. Aby wyjaśnić odnotowane zjawisko należy zauważyć, że kryteria optymalizacji obu algorytmów są przeciwne. W przypadku schedulera optymalizowany jest zrównoważony czas opóźnień, z kolei algorytm partycjonowania optymalizuje zapotrzebowanie pamięciowe. W konsekwencji współpraca schedulera z algorytmem tworzącym partycje jest słaba. Podobną obserwację odnotowano dla dwukryterialnego optymalizatora. Sprawdzono kombinacje metryk minimalizaujących pamięć: MemDiff/MemEff oraz minimalizujących czasy

odpowiedzi: HRDiff/HREff. Wykresy dla tych kombinacji zostały nazwane od nazw wybranych metryk. Przykładowo konfiguracja metryki MemDiff z HRDiff ma nazwę MemDiffHRDiff. Przeprowadzone testy pokazują, że połączenie obu metryk nie pozwoliło osiągnąć efektu synergii.

Zebrane pomiary wartości maksymalnych na rys. 4.53 i rys. 4.54 pozwoliło odnotować interesującą własność. Jeżeli dana konfiguracja prowadziła do minimalizacji mediany czasu odpowiedzi, wtedy również odnotowywana była minimalizacja maksymalnych wartości obciążenia pamięciowego i vice versa. Z punktu widzenia zastosowań, własność ta wskzuje, że stworzenie optymlizatora minimalizaującego czasy odpowiedzi i obciążenie pamięciowe można osignąć łatwiej łącząc minimalizację wartości średnich z minimalizacją wartości maksymalnych zamiast budować optymalizator minimalizujący wartości średnie dla obu kreyteriów. Budowa oraz testy rozwiazania które skorzystałoby z tego spostrzeżenia przewidziane jest w dalszych pracach badawczych.

4.3.5 **Wnioski i uwagi**

Zaproponowany autorski algorytm optymalizacji podziału operatorów na partycje wyróżnia się dwiema cechami. W przeciwieństwie do istniejących rozwiązań jest on uniwersalniejszy, ponieważ nie ogranicza się do konfiguracji operatorów tworzących prosty ciąg lub rozwidlenie. Ponadto zaproponowane podejście uwzględnia priorytety nadane partycjom przez algorytm schedulera. Dzięki temu na początku tworzone są partycje najkorzystniejsze z punktu widzenia schedulera. Dzięki takiej integracji jak pokazują przeprowadzone testy zaproponowane rozwiązanie pozwala znacząco zmniejszyć czasy odpowiedzi o 30%, przy jednoczesnym utrzymaniu wartości maksymalnych czasów odpowiedzi i zapotrzebowania pamięciowego na niskim poziomie.

W przeprowadzonych badaniach zwrócono również uwagę na metodykę oceny wydajności schedulerów oraz algorytmów tworzenia partycji. Dotąd w pracach koncentrowano się na wartościach średnich. Przeprowadzone testy pokazują, że znaczne różnice pomiędzy rozwiązaniami są odnotowywane, gdy porównamy mediany i wartości maksymalne. Takie zestawienie umożliwia ocenić stabilność rozwiązań. Przykładowo podczas analizy wyników dla optymalizacji

wielokryterialnej, wartości średnie nie ulegały dużym zmianom, podczas gdy wartości maksymalne ulegały mocnemu pogorszeniu.

Rozdział 5. Podsumowanie

Celem niniejszej rozprawy jest weryfikacja tez postawionych w rozdziale 1. Dotyczą one zastosowania drzew atrybutów oraz elementów języków obiektowych w rozwoju języków zapytań strumieniowych oraz integracji komponentów strumieniowych baz danych. Aby przybliżyć czytelnikowi poruszaną problematykę, w rozdziałach 1 i 2 zamieszczono przegląd istniejących rozwiązań oraz zasady działania strumieniowej bazy danych.

Pierwsza z tez pracy zakłada, że możliwe jest rozszerzenie języka zapytań strumieniowych o drzewo atrybutów oraz elementy języka obiektowego, co ograniczy potrzebę zmiany jego składni oraz ułatwia budowę złożonych zapytań. W celu dowiedzenia tej koncepcji stworzono prototypowy autorski język zapytań StreamAPAS. Jego opis w rozdziale 3 obejmuje omówienie szczegółów implementacyjnych decydujących o uniwersalności rozwiązania oraz analizę z perspektywy użytkownika strumieniowej bazy danych. Możliwości składniowe języka zostały zaprezentowane w dwóch etapach. Wpierw poprzez porównanie składni zapytań zapisanych w języku CQL oraz StreamAPAS dla popularnego w tej dziedzinie przykładu Linear Road Benchmark. Następnie opisano sposób zastosowania autorskiego języka do obsługi strumieniowych hurtowni danych. W trakcie tych dwóch odsłon wskazano także na zalety stosowania nowego sposobu manipulowaniem danymi nazwanego drzewem atrybutów.

Kluczowym elementem decydującym o potencjalnym szerokim zastosowaniu języka StreamAPAS jest zdefiniowanie translacji elementów składni języka zapytań na pojęcia stosowane w językach obiektowych. Dzięki niej w czytelny sposób można rozszerzać składnię i funkcjonalność języka. Ponadto zdefiniowana translacja elementów języka zapytań na obiekty umożliwiła stworzenie automatycznej integracji języka zapytań z językiem implementującym komponenty strumieniowej bazy danych poprzez mechanizm refleksji. Obecnie producenci baz danych umożliwiają dołączanie nowej funkcjonalności do języków zapytań. Zaproponowane rozwiązanie wyróżnia się tym, że w sposób zwięzły integruje rozszerzanie

funkcjonalności przetwarzania strumieniowego jak i funkcjonalności służącej do analizy semantycznej.

Druga teza pracy dotyczyła obniżania czasów odpowiedzi lub obciążenia pamięciowego poprzez taką integrację komponentów strumieniowej bazy danych, która skutkuje redukcją obszarów synchronizowanych. Jak opisano w rozdziale 4, strumieniowa baza danych składa się z wielu modułów działających współbieżnie. W rezultacie czas synchronizacji ma znaczący wpływ na osiąganą wydajność strumieniowej bazy danych. W niniejszej rozprawie rozważono dwie drogi zmniejszenia udziału synchronizacji. Pierwsze podejście polegało na zbudowaniu architektury silnika strumieniowej bazy danych, która uwzględnia charakterystyki komunikacji pomiędzy modułami. W sekcji 4.1 wykazano, że zaproponowane rozwiązanie przyczyniło się do znaczącego obniżenia czasów odpowiedzi. Aby zrozumieć genezę opóźnień, w sekcji 4.2 przeprowadzono analizę teoretyczną zapytań strumieniowych. Doświadczenie związane z modelowaniem zostało następnie użyte w sekcji 4.3 do zbudowania autorskiego optymalizatora partycji, który łączy wiedzę o parametrach pracy schedulera oraz konfiguracji zapytań. Otrzymane rozwiązanie pozwala obniżyć czasy odpowiedzi albo obciążenie pamięciowe systemu poprzez trafniejszy dobór połączeń strumieniowych i połączeń bezpośrednich, czyli metod synchronizacji pracy operatorów.

Prawdziwość tez rozprawy została potwierdzona za pomocą rozważań teoretycznych i uzupełniona wynikami testów empirycznych wykorzystujących działające rozwiązania. Należy podkreślić, że przedstawione wyniki zostały opublikowane na międzynawowych konferencjach i w czasopiśmie. Część badań została dodatkowo objęta grantem unijnym. Treść niniejszej rozprawy opracowano po publikacji materiałów, co pozwoliło na wprowadzenie uwag pochodzących od recenzentów.

5.1 Plany przyszłych prac

Badania omówione w rozprawie obejmują szeroki wachlarz komponentów strumieniowej bazy danych począwszy od zagadnienia konstruowania języka zapytań skończywszy na integracji modułów strumieniowej bazy danych w celu redukcji

synchronizacji. Takie podejście do tematyki pozwoliło stworzyć autorskie rozwiązania mające na celu poprawienie wydajności czasowej i pamięciowej, poprzez integrację modułów w strumieniowych bazach danych. Przyszłe prace badawcze wiąże z dziedziną systemów przetwarzania strumieniowego. Obecnie w zespole przeprowadzono szereg badań nad systemami agentowymi, indeksami, wznawianiem przetwarzania po wystąpieniu awarii oraz strumieniowymi bazami danych, co przedstawiono w [38]. Spójność tematyczna prowadzonych badań i zdobyte doświadczenie pozwala podjąć się większych wyzwań mających na celu zastosowanie wiedzy o strumieniowych bazach danych w strumieniowych hurtowniach danych oraz systemach agentowych. Do interesujących kierunków najbliższych prac zaliczam cztery problematyki.

Najbardziej obiecującą dziedziną jest rozwój języka zapytań strumieniowych. Obecnie przetwarzanie strumieniowe jest stosowane nie tylko w strumieniowych bazach danych, ale również w systemach agentowych, hurtowniach danych i serwerach aplikacji. Powyższe przesłanki wskazują, że w przyszłości grozi powstanie wielu języków zapytań dopasowanych do konkretnych zastosowań, co może finalnie utrudnić dostęp do nowych technologii. Aby temu zapobiec interesujące jest zastosowanie założeń języka StreamAPAS w celu stworzenia łatwo rozszerzalnego języka zapytań, tak aby móc go dopasować do wymienionych nowych obszarów przetwarzania strumieniowego.

Drugą tematyką, którą pragnę się zająć w przyszłych badaniach jest skonstruowanie architektury platformy przewarzania strumieniowego korzystającej z systemów agentowych i chmur obliczeniowych. Obecnie przesłanki ekonomiczne skłaniają do wynajmowania mocy obliczeniowej i pamięci w serwisach zewnętrznych. Dodatkowo takie rozwiązanie zwalnia klienta od konieczności prowadzenia szeregu prac administracyjnych; przykładowo aktualizacji serwisów, z których korzysta oprogramowanie. Gdy dołączymy do tego obrazu fakt, że wzrasta zainteresowanie narzędziami służącymi do realizacji obliczeń analitycznych na bieżąco, umieszczenie platformy przetwarzania strumieniowego w chmurach obliczeniowych wydaje się atrakcyjnym podejściem. Z drugiej strony wiąże się ono z szeregiem nowych wyzwań, do których należy zaliczyć między innymi: zarządzanie polityką prywatności oraz równoważenie obciążenia w systemie rozproszonym.

Prowadzone badania nad strumieniową bazą danych pokazały, że dla optymalizacji czasowej i pamięciowej kluczową rolę odgrywa definicja operatorów strumieniowych. W zależności od przyjętych założeń, możliwości optymalizacji są szersze lub węższe. Zagadnienie budowy logiki i optymalizacji operatorów strumieniowych jest dobrze rozpoznane dla operatorów mających swoje odpowiedniki w relacyjnych bazach danych. Otwarte pole do badań jest nadal w dziedzinie adaptacji operatorów stosowanych w hurtowniach danych do przetwarzania strumieni. Należy tutaj podkreślić, że nie chodzi wyłącznie o drobne modyfikacje mające na celu zastąpienie tabel relacyjnych danymi strumieniowymi. Główny problem powstaje, kiedy tworzone są zapytania złożone. Wtedy pojawia się trudność w interpretacji całego zapytania, ponieważ należy dodatkowo uwzględnić czasy życia danych. W analogi do strumieniowych baz danych, obiecującym kierunkiem jest zastosowanie do optymalizacji znajomości monotoniczności danych w strumieniowych hurtowniach danych. Wiąże się to jednak z problemem identyfikacji czasu względem, którego będą realizowane obliczenia. Praktycznie większość systemów produkcyjnych i sprzedażowych obsługiwanych przez hurtownie danych posiada kilka czasów, do znaczników czasu po stronie producenta możemy zaliczyć: czas zamówienia, czas produkcji, czas testów, czas wypuszczenia z fabryki, czas wstawienia do systemu informatycznego; po stronie sprzedaży mamy: czas produkcji towaru, czas sprzedaży, czas płatności, czas dostarczenia. W obliczu tak licznej grupy znaczników czasu trudno jest ustalić, który będzie właściwy do przetwarzania krotek. Pojawia się także pytanie czy istnieją niezmienniki opisujące te czasy, które pozwoliłyby na optymalizację realizacji zapytań.

Na zakończenie, nie mniej ważną dziedziną, której rozwój jest istotny dla badań nad platformami strumieniowego przetwarzania jest zagadnienie integracji z serwerami aplikacji. Stanowią one środowisko programistyczne, w którym umieszcza się znaczącą liczbę nowych aplikacji przeznaczonych do analizy danych. Z tej perspektywy uzasadnione jest stworzenie wygodnego w użytkowaniu i wydajnego interfejsu rozszerzającego standardowy zestaw interfejsów przykładowo dla serwerów aplikacji zgodnych z standardem J2EE. Obecnie istnieją interfejsy do przekazywania danych strumieniowych, nadal jednak brakuje interfejsów

obsługujących zarządzanie zapytaniami. Moim zdaniem osiągnięcie tego celu jest ważne, ponieważ usprawniłoby to komercjalizację rozwiązań z tej dziedziny.

5.2 Podziękowania

Czas jest zasobem ograniczonym i nie rzadko deficytowym. Dlatego chciałbym podziękować za pomoc ze strony wiecznie zapracowanych: Rafała Malczoka, Pawła Marksa, Sławomira Bańkowskiego, Michała Gorawskiego, Michała Thiele i Pawła Jureczek. Możliwość skonfrontowania własnych pomysłów, z ich uwagami i doświadczeniem było nieocenioną pomocą.

Bibliografia

[1] Daniel J. Abadi et al., "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, pp. 120-139, 2003.

[2] M. H. Ali et al., "Nile-PDT: a phenomenon detection and tracking framework for data stream management systems," in *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, 2005, pp. 1295-1298.

[3] Mohamed H. Ali, Badrish Chandramouli, Jonathan Goldstein, and Roman Schindlauer, "The extensibility framework in Microsoft StreamInsight," in *ICDE*, 2011, pp. 1242-1253.

[4] Arvind and Widom, Jennifer Arasu, "A denotational semantics for continuous queries over streams and relations," in *SIGMOD Rec.*, vol. 33, 2004, pp. 6-11.

[5] Arvind Arasu, Shivnath Babu, and Jennifer Widom, "The CQL continuous query language: semantic foundations and query execution," in *The VLDB Journal*, 2006, pp. 121-142.

[6] Arvind Arasu et al., "Linear Road: A Stream Data Management Benchmark," in *VLDB*, 2004, pp. 480-491.

[7] Lars Arge, Octavian Procopiuc, and Sridhar Ramaswamy, "Scalable Sweeping-Based Spatial Join," in *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, 1998, pp. 570-581.

[8] Arasu Arvind and Jennifer Widom, "A denotational semantics for continuous queries over streams and relations," in *SIGMOD Rec.*, 2004, pp. 6-11.

[9] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwani, "Chain: Operator Scheduling for Memory Minimization in Data Stream Systems," in *ACM International Conference on Management of Data (SIGMOD 2003)*, 2003.

[10] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas, "Operator scheduling in data stream systems," *The VLDB Journal*, vol. 13, pp. 333-353, 2004.

[11] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom, "Models and issues in data stream systems," in *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2002, pp. 1-16.

[12] Yijian Bai, Hetal Thakkar, Haixun Wang, and Carlo Zaniolo, "Optimizing Timestamp Management in Data Stream Management Systems Data Engineering," in *ICDE 2007. IEEE 23rd International Conference on*, 2007, pp. 1334-1338.

[13] Yijian Bai and Carlo Zaniolo, "Minimizing latency and memory in DSMS: a unified approach to quasi-optimal scheduling," in *SSPS '08: Proceedings of the 2nd international workshop on Scalable stream processing system*, 2008, pp. 58-67.

[14] Hari Balakrishnan et al., "Retrospective on Aurora," *The VLDB Journal*, vol. 13, pp. 370-383, 2004.

[15] Magdalena Balazinska, "Fault-tolerance and load management in a distributed stream processing system," Cambridge, MA, USA, Thesis 2006.

[16] Roger S. Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong, "Consistent Streaming Through Time: A Vision for Event Stream Processing," in *CIDR*, 2007, pp. 363-374.

[17] Irina Botan, Gustavo Alonso, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul, "Flexible and scalable storage management for data-intensive stream processing," in *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, 2009, pp. 934-945.

[18] Michael Cammert, Christoph Heinz, Jűrgen Krämer, and Markowetz Alexander, "PIPES: A Multi-Threaded Publish-Subscribe Architecture for Continuous Queries over Streaming Data Sources," Department of Mathematics and Computer Science, University of Marburg, 2003.

[19] Donggang Cao, Hong Mei, Qianxiang Wang, and Gang Huang, "Microkernel architecture: Making application servers open to change," *Chinese Journal of Electronics*, no. 14, pp. 443-448, 2005.

[20] Luca Cardelli and Giorgio Ghelli, "A Query Language Based on the Ambient Logic," in *ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems*, 2001, pp. 1-22.

[21] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon, "Types for the ambient calculus," *Inf. Comput.*, vol. 177, pp. 160-194, 2002.

[22] Don Carney et al., "Operator scheduling in a data stream manager," in *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, 2003, pp. 838-849.

[23] Sirish Chandrasekaran et al., "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," in *CIDR*, 2003, pp. 668-679.

[24] Giovanni Conforti, Orlando Ferrara, and Giorgio Ghelli, "TQL Algebra and its Implementation," in *In Proc. of IFIP TCS*, 2002, pp. 422-434.

[25] Giovanni Conforti et al., "The Query Language TQL," in *WebDB*, 2002, pp. 13-18.

[26] Guojing Cong and David Bader, "Lock-free parallel algorithms: An experimental study," in *In Proceedings of the 11th International Conference High Performance Computing*, 2004, pp. 516-528.

[27] Chuck Cranor, Theodore Johnson, and Oliver Spataschek, "Gigascope: a stream database for network applications," in *SIGMOD international conference on Management of data*, 2003, pp. 647-651.

[28] Tadeusz Czachórski, "Modele kolejkowe systemów komputerowych," Politechnika Śląska, skrypt 2151.

[29] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald, "Semantic Approximation of Data Stream Joins," *IEEE Trans. on Knowl. and Data Eng.*, vol. 17, pp. 44-59, 2005.

[30] Definicja strumieniowej bazy danych w wiki. http://pl.wikipedia.org/wiki/Strumieniowa_baza_danych

[31] Luping Ding, Elke A. Rundensteiner, and George T. Heineman, "MJoin: a metadata-aware stream join operator," in *DEBS '03: Proceedings of the 2nd international workshop on Distributed event-based systems*, 2003, pp. 1-8.

[32] Hui Ding, Goce Trajcevski, and Peter Scheuermann, "OMCAT: optimal maintenance of continuous queries' answers for trajectories," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006, pp. 748-750.

[33] Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor, and Peter Widmayer, "Progressive merge join: a generic and non-blocking sort-based join algorithm," in *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, 2002, pp. 299-310.

[34] Françoise Fabret, H. Arno Jacobsen, François Llirbat, and Pereira João, "Filtering Algorithms and Implementation for Very Fast Publish/Subscribe," in *In SIGMOD*, 2001, pp. 115-126.

[35] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy, "Mining data streams: a review," *SIGMOD Rec.*, vol. 34, pp. 18-26, 2005.

[36] T. Ghanem, M. Hammad, M. Mokbel, W. G. Aref, and A. Elmagarmid, "Query Processing using Negative Tuples in Stream Query Engines," *Purdue University*, no. 04-040, 2005.

[37] Lukasz Golab, "Sliding Window Query Processing over Data Streams," Thesis 2006.

[38] Marcin Gorawski, *Advanced Data Warehouses*.: Studia Informatica, vol. 30, nr 3B, 2009.

[39] Marcin Gorawski and Aleksander Chrószcz, "Optimization of Operator Partitions in Stream Data," in *DOLAP*, Glasgow, 2011, pp. 61-66.

[40] Marcin Gorawski and Aleksander Chrószcz, "Prototypowy język zapytań strumieniowych – StreamAPAS v2.0," *Studia Informatica*, vol. 29, pp. 5-22, 2008.

[41] Marcin Gorawski and Aleksander Chrószcz, "Query Processing Using Negative and Temporal Tuples in Stream Query Engines," in *CEE-SET*, Kraków, 2009, pp. 51-64.

[42] Marcin Gorawski and Aleksander Chrószcz, "StreamAPAS: Query Language and Data Model," in *CISIS*, Fukuoka, Japan, 2009, pp. 75-82.

[43] Marcin Gorawski and Aleksander Chrószcz, "StreamAPAS: Query Language and Data Model," in *Complex Intelligent Systems and Their Applications*.: Springer Publishing Company, Incorporated, 2010, ch. 9, pp. 187-205.

[44] Marcin Gorawski and Aleksander Chrószcz, "System przetwarzania strumieniowego: StreamAPAS v5.0," *Studia Informatica*, vol. 30, pp. 7-34, 2009.

[45] Marcin Gorawski and Aleksander Chrószcz, "The Design of Stream Database Engine in Concurrent Environment," in *OTM Conferences*, Vilamoura, Portugal, 2009, pp. 1033-1049.

[46] Xiaohui Gu, "Adaptive Load Diffusion for Multiway Windowed Stream Joins," in *ICDE*, 2007, pp. 146-155.

[47] Moustafa A. Hammad, Walid G. Aref, Michael J. Franklin, Mohamed F. Mokbel, and Ahmed K. Elmagarmid, "Incremental Evaluation of Sliding-Window Queries over Data Streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, pp. 57-72, 2007.

[48] Eric N. Hanson and Moez Chaabouni, "The IBS-tree: A Data Structure for Finding All Intervals That Overlap a Point," Wright State University Dept. of Computer Science, 1994.

[49] Haibo Hu, Jianliang Xu, and Dik Lun Lee, "A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects," in *In SIGMOD*, 2005, pp. 479-490.

[50] Jeong-Hyon Hwang, Ying Xing, Uğur Çetintemel, and Stan Zdonik, "A Cooperative, Self-Configuring High-Availability Solution for Stream Processing," in *International Conference on In Data Engineering, ICDE 2007*, 2007, pp. 176 - 185.

[51] J.P.Corrigan, "OPRA Traffic Projections for 2005 and 2006," 2005.

[52] J.P.Corrigan, "OPRA Traffic Projections for 2009 and 2010," 2009.

[53] Qingchun Jiang and Sharma Chakravarthy, "Scheduling strategies for processing continuous queries over streams," in *BNCOD'04*, 2004, pp. 16-30.

[54] Mohamed Ali, Rogrer Barga, Mingsheng Hong Jonathan Goldstain, "Consistency Sensitive Streaming Operators in CEDR," Microsoft Research, MSR-TR-2007-158,.

[55] Jűrgen Krämer, "Continuous queries over data streams semantics and implementation," Philipps-Universität, Marburg, Thesis 2007.

[56] Jűrgen Krämer and Bernhard Seeger, "A Temporal Foundation for Continuous Queries over Data Streams," in *COMAD*, 2005, pp. 70-82.

[57] Geetika T. Lakshmanan, Ying Li, and Rob Strom, "Placement Strategies for Internet-Scale Data Stream Systems," *IEEE Internet Computing*, vol. 12, no. 6, pp. 50-60, 2008.

[58] Ramon Lawrence, "Early hash join: a configurable algorithm for the efficient and early production of join results," in *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, 2005, pp. 841-852.

[59] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005, pp. 311-322.

[60] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman, "Continuously Adaptive Continuous Queries over Streams," in *SIGMOD Conference*, 2002, pp. 49-60.

[61] Jeremy Manson and William Pugh, "Requirements for Programming Language Memory Models.," in *In PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[62] Maged M. Michael, "CAS-based lock-free algorithm for shared deques," in *In the 9th Euro-Par Conference on Parallel Processing*, 2003, pp. 651-660.

[63] Maged M. Michael and Michael L. Scott, "Simple, Fast and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," in *Proc. 15th ACM Symp. on Principles of Distributed Computing*, 1996, pp. 267-275.

[64] Rajeev Motwani et al., "Query Processing, Resource Management, and Approximation in a Data Stream Management System," in *CIDR*, 2003, pp. 245-256.

[65] Rimma V. Nehme, "Continuous Query Processing on Spatio-Temporal Data Streams," Worcaster Polytechnic Insititue, Thesis 2005.

[66] OGI School of Science & Engineering at OHSU. Strona domowa opisująca benchmark NEXMark. http://datalab.cs.pdx.edu/niagara/NEXMark/

[67] Gultekin Özsoyoğlu and Richard T. Snodgrass, "Temporal and Real-Time Databases: A Survey," in *IEEE Trans. on Knowl. and Data Eng.*, 1995, pp. 513-532.

[68] Neoklis Polyzotis, Spiros Skiadopoulos, Panos Vassiliadis, Alkis Simitsis, and Nils Frantzell, "Meshing Streaming Updates with Persistent Data in an Active Data Warehouse," *IEEE Educational Activities Department*, vol. 20, pp. 976-991, 2008.

[69] Neoklis Polyzotis, Spiros Skiadopoulos, Panos Vassiliadis, Alkis Simitsis, and Nils-Erik Frantzell, "Supporting Streaming Updates in an Active Data Warehouse," in *ICDE*, 2007, pp. 476-485.

[70] Shao Qian and YiLi Lu, "A modified chain scheduling algorithm in data stream system," in *Computer and Automation Engineering (ICCAE),* 2010, pp. 568 - 570.

[71] Frederick Reiss, Kurt Stockinger, Kesheng Wu, Arie Shoshani, and Joseph M. Hellerstein, "Enabling Real-Time Querying of Live and Historical Stream Data," in *SSDBM '07: Proceedings of the 19th International Conference on Scientific and Statistical Database Management*, 2007, pp. 28-37.

[72] Walid Rjaibi, Klaus R. Dittrich, and Dieter Jaepel, "Event Matching in Symmetric Subscription Systems," in *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, 2002, pp. 9-20.

[73] Sangeetha Seshadri, Vibhore Kumar, and Brian F. Cooper, "Optimizing Multiple Queries in Distributed Data Stream Systems," in *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on Data Engineering Workshops*, 2006, pp. 25-30.

[74] Mehul A. Shah, Michael J. Franklin, Samuel Madden, and Joseph M. Hellerstein, "Java support for data-intensive systems: experiences building the telegraph dataflow system," *SIGMOD Rec.*, vol. 30, pp. 103-114, 2001.

[75] Mohamed A. Sharaf, "Metrics and algorithms for processing multiple continuous queries," University of Pittsburgh, Thesis 2007.

[76] Mohamed A. Sharaf, Panos K. Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs, "Efficient scheduling of heterogeneous continuous queries," in *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, 2006, pp. 511-522.

[77] Giedrius Slivinskas, Christian S. Jensen, and Richard Thomas Snodgrass, "A Foundation for Conventional and Temporal Query Optimization Addressing Duplicates and Ordering," *IEEE Trans. on Knowl. and Data Eng.*, vol. 13, no. 1, pp. 21-49, 2001.

[78] Giedrius Slivinskas, Christian S. Jensen, and Richard T. Snodgrass, "Query Plans for Conventional and Temporal Queries Involving Duplicates and Ordering," in *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*, 2000, pp. 547-558.

[79] Utkarsh Srivastava and Jennifer Widom, "Flexible time management in data stream systems," in *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2004, pp. 263-274.

[80] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik, "The 8 requirements of real-time stream processing," *SIGMOD Record*, vol. 34, pp. 42-47, 2005.

[81] Strona z rzeczywistymi zbiorami pomiarów ruchu sieciowego. [Online]. http://ita.ee.lbl.gov/html/contrib/LBL-PKT.html

[82] Strona domowa projektu STREAM. http://infolab.stanford.edu/stream/sqr/.

[83] Mark Sullivan and Andrew Heybey, "Tribeca: A System for Managing Large Databases

of Network Traffic," in *In USENIX*, 1998, pp. 13-24.

[84] Sun, "JSR-133: JavaTM Memory Model and Thread Specification," Sun, JSR-133 2004.

[85] Håkan Sundel and Philippas Tsigas, "Lock-Free and Practical Deques using Single-Word Compare-And-Swap," *CoRR*, vol. cs.DC/0408016, 2004.

[86] Emine Nesime Tatbul, "Load shedding techniques for data stream management systems," Brown University, Thesis 2007.

[87] Douglas Terry, David Goldberg, David Nichols, and Brian Oki, "Continuous queries over append-only databases," in *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, 1992, pp. 321-330.

[88] Peter A. Tucker, "Punctuated Data Streams," Thesis 2005.

[89] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier, "NEXMark – A Benchmark for Queries over Data Streams," OGI School of Science & Engineering at OHSU, http://datalab.cs.pdx.edu/niagara/NEXMark/, 2002.

[90] Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger, "Maximizing the output rate of multi-way join queries over streaming information sources," in *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, 2003, pp. 285-296.

[91] Song Wang, Chetan Gupta, and Abhay Mehta, "VPipe: Virtual Pipelining for Scheduling of DAG Stream Query Plans," in *In BIRTE*, 2009, pp. 32-49.

[92] Haixun Wang, Carlo Zaniolo, and Chang Luo, "ATLAS: A Small but Complete SQL Extension for Data Mining and Data Streams," in *VLDB*, 2003, pp. 1113-1116.

[93] Wikipedia. http://pl.wikipedia.org/wiki/Prawo_Amdahla

[94] Eran Yahav and Mooly Sagiv, "Automatically verifying concurrent queue algorithms," in *Electr. Notes Theor. Comput. Sci*, 2003, pp. 450-463.

Dodatek A. Pełna gramatyka języka StreamAPAS

Lekser

W języku StreamAPAS komentarz jest wprowadzany na dwa sposoby:

1) // tekst – wszystkie znaki od // do znaku końca linie tworzą komentarz
2) /* tekst */ – wszystkie znaki od '/*' do '*/' tworzą komentarz

W języku występują cztery typy literałów:

1) Liczba całkowita która jest zdefiniowana jako:

```
(([1-9][0-9]*)|(0))[lL]?
```

Jeżeli literał jest zakończony znakami 'l' lub 'L' wtedy reprezentuje on liczbę całkowitą typu Long. W przeciwnym wypadku tworzona jest liczba całkowita typu Integer.

2) Liczba zmiennoprzecinkowa:

```
[0-9]+ (\. [0-9]+)? (e [0-9]+)?[fFdD]?
```

Jeżeli literał jest zakończony znakami 'f' lub 'F' wtedy reprezentuje on wartość typu Float w przeciwnym wypadku literał definiuje wartość typu Double.

3) Zmienna typu String jest definiowana przez literał:

```
\"[^\"]*\"
```

4) Identyfikator w języku zapytań jest zdefiniowany przez ciąg znaków:

```
[$a-zA-Z][#$a-zA-Z0-9]
```

Dodatkowo ciąg znaków nie może tworzyć nazwy będącej słowem zastrzeżonym.

W języku StreamAPAS wyróżnia się następujące słowa zastrzeżone: true, false, AND, OR, XOR, begin, end, select, from, where, group by, having, as, axis, ::, =, +, -, *, /, (,), [,], {, }, !, ., \, ;, &, ?, ==, <, >, <=, >=, !=.

Parser

Jednostka kompilacji

Zapytanie składa się z jednego lub kilku Unit będących jednostkami kompilacji.

```
1)
units: qUnit ';'
 | units qUnit ';'
 ;
2)
qUnit: qUnitDecl
 | qUnitDecl BEGIN qUnitDef END
 ;
3)
qUnitDecl: IDENTIFIER
 | IDENTIFIER qUnitStates
 ;
4)
qUnitStates: IDENTIFIER
 | qUnitStates ','
 ;
```

Definicja elementów Task.

```
5)
qUnitDef: task
 | qUnitDef task
 ;
6)
task: querySpecification
 | call
 ;
7)
queryTerm: '(' querySpecification ')'
 | IDENTIFIER '{''}'
 ;
8)
querySpecification: queryClause
 | datasetClause
 ;
9)
datasetClause: IDENTIFIER '{' call '}'
 ;
```

Definicja frazy Select-From

```
10)
queryClause: SELECT selectClause fromClause
 ;
11)
selectClause: objectDef
 ;
12)
fromClause : whereClause
 | FROM fromPhrase whereClause
 ;
13)
fromPhrase: querySource
 | fromPhrase ',' querySource
 ;
```

```
14)
querySource: queryTerm
 | IDENTIFIER
 ;
15)
whereClause : groupByClause
 | WHERE searchCondition groupByClause
 ;
16)
searchCondition: expr
 | searchConditionObjList
 | searchConditionObjList ',' expr
 ;
17)
searchConditionObjList: objectLink
 | searchConditionObjList ',' objectLink
 ;
18)
objectLink: IDENTIFIER '{' objectLinkType '}'
 ;
19)
objectLinkType: memberList {$$ = $1;}
 ;
20)
groupByClause : havingClause
 | GROUPBY groupingElementList havingClause
 ;
21)
groupingElementList: qualifiedIdentifier
 | groupingElementList ',' qualifiedIdentifier
 ;
22)
havingClause :
 | HAVING expr
 ;
```

Definicja drzewa atrybutów

```
23)
objectDef: objectReference
 ;
24)
objectReference: IDENTIFIER '{' memberDef '}'
 ;
25)
memberDef: memberList
 ;
26)
memberList: memberNode
 | memberList ',' memberNode
 ;
27)
memberNode: assignmentExpression
 | '[' memberList ']'
 | qualifiedIdentifier '[' memberList ']'
 ;
28)
assignmentExpression: conditionalExpression
 | qualifiedIdentifier '=' conditionalExpression
 ;
```

Definicja wyrażeń arytmetycznych

```
29)
conditionalExpression: expr
```

```
 ;
30)
expr: expr AND expr
 | expr OR expr
 | expr XOR expr
 | expr EQUALITY expr
 | expr INEQUALITY expr
 | expr '-' expr
 | expr '+' expr
 | expr '*' expr
 | expr '/' expr
 | prefixExpr
 ;
31)
prefixExpr: atom
 | '-' prefixExpr %prec UNARY
 | '!' prefixExpr %prec UNARY
 ;

atom: CONSTANT
 | STR
 | ambigiousAtom
 | call
 | '(' assignmentExpression ')'
 ;
32)
ambigiousAtom: qualifiedIdentifier
 ;
33)
call: qualifiedIdentifier '(' callArgList ')'
 | qualifiedIdentifier '(' ')'
 | qualifiedIdentifier DOTS IDENTIFIER '(' callArgList ')'
 | qualifiedIdentifier DOTS IDENTIFIER '(' ')'
 ;
34)
callArgList : callArgListObj
 | callArgList ',' callArgListObj
 ;
35)
callArgListObj: memberNode
 | queryTerm
 ;
36)
qualifiedIdentifier: IDENTIFIER
 | qualifiedIdentifier '.' IDENTIFIER
 ;
```

Spis symboli i skrótów

- DAG (ang. Directed Acyclic Graph) – skierowany acykliczny graf służący do reprezentacji zapytań w strumieniowej bazie danych. W zależności od kontekstu użycia graf reprezentuje operatory logiczne albo operatory fizyczne.
- DBMS (ang. Data Base Management System) – moduł odpowiedzialny za zarządzanie pracą relacyjnej bazy danych
- DSMS (ang. Data Stream Management System) – moduł odpowiedzialny za zarządzanie pracą strumieniowej bazy danych
- CQL (ang. Continuous Query Language) – deklaratywny język zapytań służący do tworzenia, modyfikacji i operacji na strumieniowej bazie danych STREAM
- SQL (ang. Structures Query Language) – deklaratywny strukturalny język służący do tworzenia, modyfikowania i operacji na relacyjnych bazach danych.

Spis ilustracji

RYS. 2.1. ZESTAWIENIE RÓŻNYCH DEFINICJI STRUMIENI 28
RYS. 2.2. PRZYKŁADOWY WYNIK OPERATORA ZŁĄCZENIA DANYCH TEMPORALNYCH 39
RYS. 2.3. REZULTAT ŁĄCZENIA STRUMIENI DLA RÓŻNYCH PODEJŚĆ 40
RYS. 2.4. PRZEBIEG PROCESU AGREGACJI DLA MODELU: A) TEMPORALNEGO, B) Z KROTKAMI POZYTYWNYMI I NEGATYWNYMI 46
RYS. 2.5. WYZNACZANIE AGREGATÓW DLA MODELU Z KROTKAMI TEMPORALNYMI I NEGATYWNYMI 48
RYS. 2.6. PRZETWARZANIE STRUMIENIA WEJŚCIOWEGO DLA OPERATORA ELIMINUJĄCEGO DUPLIKATY 54
RYS. 2.7. PRZETWARZANIE STRUMIENIA WEJŚCIOWEGO DLA OPERATORA RÓŻNICY 57
RYS. 2.8. ZAPYTANIE TESTUJĄCE KOMPRESJE STRUMIENIA AGREGACJI 63
RYS. 2.9. WYDAJNOŚĆ KOMPRESJI 64
RYS. 2.10. WPŁYW KOMPRESJI NA OPÓŹNIENIA 65
RYS. 3.1. PRZEBIEG KOMPILACJI ZAPYTAŃ 83
RYS. 3.2. PRZYKŁADOWE DRZEWO ATRYBUTÓW 87
RYS. 3.3. DRZEWO PARSOWANIA DLA PRZYKŁADOWEGO ZAPYTANIA 104
RYS. 3.4. DAG DLA PRZYKŁADOWEGO ZAPYTANIA 104
RYS. 3.5. DRZEWO PARSOWANIA DLA PRZYKŁADOWEGO ZAPYTANIA 105
RYS. 3.6. PRZYKŁADOWY SEGMENT AUTOSTRADY W LRB 106
RYS. 3.7. TABELA SYMBOLI I DRZEWO ROZBIORU SKŁADNIOWEGO DLA FRAGMENTU PROGRAMU W C 113
RYS. 3.8. FRAGMENT DRZEWA ATRYBUTÓW 116
RYS. 3.9. POCZĄTKOWA BUDOWA SYMBOLU 116
RYS. 3.10. BUDOWA SYMBOLU ZŁOŻONEGO 118
RYS. 3.11. TABELA SYMBOLI DLA ANALIZOWANEGO ZAPYTANIA 120
RYS. 3.12. PRZEKSZTAŁCANIE DRZEWA ROZBIORU SKŁADNIOWEGO 125
RYS. 4.1. PRZEBIEG KOMPILACJI ZAPYTAŃ 135
RYS. 4.2. ARCHITEKTURA SILNIKA STRUMIENIOWEGO 136
RYS. 4.3. MODUŁ WORKERÓW Z KOLEJKĄ OPERATORÓW I PULĄ WĄTKÓW 138
RYS. 4.4. DYSTRYBUCJA LOSOWA OPERATORÓW POMIĘDZY WĄTKI 139
RYS. 4.5. DYSTRYBUCJA KONTROLOWANA OPERATORÓW POMIĘDZY WĄTKI 140
RYS. 4.6. BUDOWA POJEDYNCZEGO WORKERA 141
RYS. 4.7. INTERFEJS OPERATORA FIZYCZNEGO 142
RYS. 4.8. INTERFEJS WEJŚCIA OPERATORA 143
RYS. 4.9. INTERFEJS WYJŚCIA OPERATORA 143
RYS. 4.10. WARSTWA ZASILAJĄCA OPERATOR 144
RYS. 4.11. KOLEJKA Z ELEMENTEM WARTOWNIKIEM NA POCZĄTKU LISTY 147
RYS. 4.12. DWUKROTNE ZMNIEJSZENIE ZAPOTRZEBOWANIA PAMIĘCIOWEGO DLA KOLEKCJI S_1 PRZEZ ZASTOSOWANIE PARTYCJI OPERATORÓW 152
RYS. 4.13. OPTYMALIZACJA PAMIĘCIOWA UZYSKANA DZIĘKI POŁĄCZENIU OPERATORÓW 153
RYS. 4.14. ZESTAWIENIE OPÓŹNIEŃ DLA PRZEPROWADZONYCH BADAŃ 154
RYS. 4.15. ZESTAWIENIE OPÓŹNIEŃ DLA PRZEPROWADZONYCH BADAŃ 155
RYS. 4.16. ILUSTRACJA OPERATORA Z DWOMA WEJŚCIAMI 159
RYS. 4.17. INTERPRETACJA GRAFICZNA OPÓŹNIENIA ORAZ ŚREDNIEGO ODSTĘPU POMIĘDZY GRUPAMI KROTEK 159
RYS. 4.18. PESYMISTYCZNY SCENARIUSZ DLA WEJŚCIA A 160
RYS. 4.19. METODA WYLICZANIA ŚREDNIEGO CZASU OPÓŹNIENIA 160
RYS. 4.20. TWORZENIE SIĘ NOWYCH GRUP KROTEK NA WYJŚCIU OPERATORA Z DWOMA WEJŚCIAMI 161
RYS. 4.21. ŚREDNI ODSTĘP POMIĘDZY GRUPAMI KROTEK NA WYJŚCIU OPERATORA DLA RÓŻNYCH PROPORCJI T_A / T_B 162
RYS. 4.22. PODEJŚCIE PODSTAWOWE DO NALICZANIA OPÓŹNIEŃ 163
RYS. 4.23. ZAPYTANIE DAG1 165
RYS. 4.24. ZAPYTANIE DAG2 169
RYS. 4.25. ZAPYTANIE DAG3 169

RYS. 4.26. ZAPYTANIE DAG4 169
RYS. 4.27. ZAPYTANIE DAG5 169
RYS. 4.28. ZAPYTANIE DAG6 169
RYS. 4.29. SPRAWDZENIE DOKŁADNOŚCI MODELU MIXED DLA PYTANIA DAG1 170
RYS. 4.30. PORÓWNANIE DOKŁADNOŚCI MODELI DLA DANYCH O ROZKŁADZIE POISSON DLA PYTANIA DAG1 170
RYS. 4.31. SPRAWDZENIE DOKŁADNOŚCI MODELU MIXED DLA DANYCH O ROZKŁADZIE RZECZYWISTYM DLA PYTANIA DAG2 171
RYS. 4.32. SPRAWDZENIE DOKŁADNOŚCI MODELU MIXED DLA DANYCH O ROZKŁADZIE POISSON DLA PYTANIA DAG2 171
RYS. 4.33. SPRAWDZENIE DOKŁADNOŚCI MODELU MIXED DLA PYTANIA DAG3 171
RYS. 4.34. SPRAWDZENIE DOKŁADNOŚCI MODELU BASIC DLA DANYCH O ROZKŁADZIE POISSON DLA PYTANIA DAG3 172
RYS. 4.35. SPRAWDZENIE DOKŁADNOŚCI MODELU MIXED DLA DANYCH O ROZKŁADZIE RZECZYWISTYM DLA PYTANIA DAG4 173
RYS. 4.36. PORÓWNANIE DOKŁADNOŚCI MODELI DLA DANYCH O ROZKŁADZIE POISSON DLA PYTANIA DAG4 173
RYS. 4.37. PORÓWNANIE DOKŁADNOŚCI MODELI DLA DANYCH O ROZKŁADZIE POISSON DLA PYTANIA DAG5 173
RYS. 4.38. PORÓWNANIE DOKŁADNOŚCI MODELI DLA DANYCH O ROZKŁADZIE RZECZYWISTYM DLA PYTANIA DAG5 173
RYS. 4.39. PORÓWNANIE DOKŁADNOŚCI MODELU MIXED DLA DANYCH O ROZKŁADZIE POISSON DLA PYTANIA DAG6 174
RYS. 4.40. PORÓWNANIE DOKŁADNOŚCI MODELU MIXED DLA DANYCH O ROZKŁADZIE RZECZYWISTYM DLA PYTANIA DAG6 174
RYS. 4.41. WSPÓŁDZIELENIE KROTEK Z JEDNEGO OPERATORA O_1 177
RYS. 4.42. WYKRESY FINALIZACJI PRZETWARZANIA DLA PRZYKŁADU Z RYS. 4.41 178
RYS. 4.43. ZAPYTANIE SKŁADAJĄCE SIĘ Z PROSTEGO CIĄGU OPERATORÓW 179
RYS. 4.44. WYKRES ZAPOTRZEBOWANIA NA PAMIĘĆ 179
RYS. 4.45. ZAPYTANIE ZAWIERAJĄCE OPERATOR WIELOWEJŚCIOWY 180
RYS. 4.46. WYKRES LICZBY KROTEK W SYSTEMIE W FUNKCJI CZASU 183
RYS. 4.47. WYKRES LICZBY KROTEK W SYSTEMIE W FUNKCJI CZASU PO WPROWADZENIU PARTYCJI O_1+O_3 183
RYS. 4.48. WYKRES ZMIANY LICZBY KROTEK W SYSTEMIE PO WPROWADZENIU PARTYCJI $O_1+O_2+O_3$ 184
RYS. 4.49. FRAGMENT ZAPYTANIA PRZEZNACZONY DO OPTYMALIZACJI PAMIĘCIOWEJ 185
RYS. 4.50. WYKRES OBCIĄŻENIA PAMIĘCIOWEGO W FUNKCJI CZASU 185
RYS. 4.51. OBCIĄŻENIE PAMIĘCIOWE W FUNKCJI OBCIĄŻENIA SYSTEMU 188
RYS. 4.52. CZASY LATENCJI W FUNKCJI OBCIĄŻENIA SYSTEMU 189
RYS. 4.53. MAKSYMALNE WARTOŚCI OBCIĄŻENIA PAMIĘCIOWEGO 189
RYS. 4.54. MAKSYMALNE WARTOŚCI LATENCJI 190

Spis tabel

TABELA 2.1. MONOTONICZNOŚĆ WYNIKU DLA Θ ZŁĄCZENIA 44
TABELA 3.1. ZBIÓR OPERATORÓW ARYTMETYCZNYCH I LOGICZNYCH DOSTĘPNYCH W STREAMAPAS 84
TABELA 3.2. DOMYŚLNE ROZSZERZANIE TYPÓW 85
TABELA 3.3. KONWERSJA TYPU BOOLEAN DO INTEGER 85
TABELA 3.4. TYPY NA KTÓRYCH SĄ ZDEFINIOWANE OPERATORY W STREAMAPAS 86
TABELA 4.1. KONFIGURACJA EKSPERYMENTÓW DLA RYS. 4.14 153