STUDIA INFORMATICA 2009
Volume 30 Number 1 (82)

Marcin GORAWSKI, Aleksander CHRÓSZCZ
Politechnika Śląska, Instytut Informatyki

SYSTEM PRZETWARZANIA STRUMIENIOWEGO: STREAMAPAS V5.0

Streszczenie. Przedstawiony zostanie prototyp systemu przetwarzania strumieniowego StreamAPAS v5.0. Składnia języka zapytań tego systemu jest utworzona z myślą o zastosowaniach analitycznych, które wymagają obsługi struktur indeksujących oraz możliwości prostego dodawania nowej funkcjonalności. Omówiono implementacje węzłów wyliczających agregaty oraz ich proces definiowania przez kompilator języka zapytań. Połączenie zalet drzewa atrybutów oraz interfejsu funkcji sprawia, że zbudowany system StreamAPAS v5.0 łatwo dostosować do zmieniających się potrzeb aplikacji.

Słowa kluczowe: drzewa atrybutów, podejście funkcjonalne, przetwarzanie strumieniowe, CQL, synchronizacja strumieni, okna czasowe, algebra temporalnych operatorów

STREAM PROCESSING SYSTEM: STREAMAPAS V5.0

Summary. This paper introduces the prototype of the stream processing system StreamAPAS v5.0. The main goal of the engine and the query language is offering the general-purpose stream processing platform for data analysis. The language syntax simplify embedding new indexes and a new functionality. In this paper we focus on the implementation of the nodes calculating aggregates and the compiler algorithms used to define the aggregates. As it is further shown, the combination of hierarchical data structures and user aggregate defined functions makes continuous processing applications easier to develop and maintain.

Keywords: attributes tree, functional approach, stream processing, CQL, streams synchronization, time windows, temporal logical operator algebra

1. Wprowadzenie

Rozszerzenie definicji danych jako strumieni pozwala skorzystać z potokowości przetwarzania, która zwiększa skalowalność rozwiązań. Elementem wyróżniającym tę klasę systemów jest polityka zarządzania danymi historycznymi. Ograniczenie potrzeby archiwizacji jest koniecznością w przypadku systemów zasilanych dużą liczbą danych. Systemy przetwarzania strumieniowego (ang. *Stream Processing System*) (w skrócie: systemy strumieniowe) służą przetwarzaniu strumieni danych, a nie ich gromadzeniu, powyższa własność sprzyja ich zastosowaniu w analityce i systemach wspierających podejmowanie decyzji (DSS). Dodatkowo uzyskiwane wyniki odzwierciedlają bieżący stan systemu, co jest oczekiwane w zastosowaniach, takich jak: nadzorowanie ruchu ulicznego, monitorowanie notowań giełdowych. Przyjęcie koncepcji przetwarzania strumieniowego wiąże się z większą złożonością procesu uruchamiania zapytania, która może wymagać zasilenia danymi historycznymi, słownikami oraz mapami. Ponadto, w trakcie działania zapytania należy monitorować węzły przetwarzające, aby w przypadku awarii poprawnie wyłączyć sieć przetwarzania [2].

Utworzony system jest kontynuacją prac badawczych nad językiem CQL[1] prowadzonych przez Zespół Algorytmów, Programowania i Systemów Autonomicznych (APAS) w Zakładzie Teorii Informatyki Instytutu Informatyki Politechniki Śląskiej.

Podczas definiowania składni języka zapytań strumieniowych naszego prototypowego systemu StreamAPAS v5.0 wzięto pod uwagę następujące cele:

- ograniczenie listy funkcji dostępnych poprzez słowa kluczowe zaszyte w składni języka, z czym spotykamy się w SQL [4],
- rozszerzenie definicji struktur danych tak, aby w prosty sposób posługiwać się danymi przestrzennymi i hierarchicznymi,
- użycie algebry dla temporalnych danych strumieniowych [3], która pozwala stworzyć czytelny język zapytań.

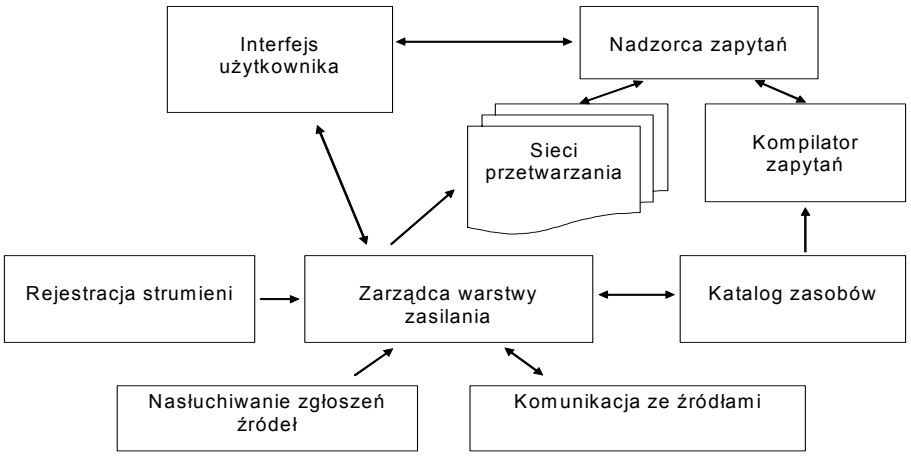
W utworzonym systemie przetwarzania strumieniowego zastosowano podejście funkcjonalne wzorowane na językach programowania, gdzie zbiór dostępnych funkcji zależy od dołączonych bibliotek. Rozwiązanie takie pozwala uniknąć sytuacji, w której na potrzebę dodania nowych funkcji należy zmieniać gramatykę języka.

Aby uprościć posługiwanie się złożonymi strukturami danych, wprowadzono drzewo atrybutów.

Zbudowany silnik StreamAPAS v5.0 bazuje na zaproponowanej w [3] algebrze operatorów temporalnych dla strumieni danych. Jej autorzy wprowadzili znaczniki początku i końca życia obiektów, co pozwoliło zdefiniować reguły optymalizacji planu produkcji dla zapytań strumieniowych.

2. Architektura systemu przetwarzania strumieniowego

Na rys. 1 zamieszczono poglądowy schemat zbudowanego systemu przetwarzania strumieniowego.



Rys. 1. Architektura systemu przetwarzania strumieniowego
Fig. 1. The architecture of the stream processing system

Poniżej przedstawiono opis ważniejszych elementów składowych systemu.

Zarządca warstwy zasilania – moduł ten aktualizuje informacje o metaschemacie strumieni w ramach katalogu zasobów oraz zasila danymi uruchomione strumieniowe sieci przetwarzania. Aby podłączyć do systemu nowe źródło danych, należy najpierw zarejestrować nazwę, pod którą dany zasób będzie dostępny. W ramach rejestracji nie podajemy definicji metaschematu strumienia, dostarcza go źródło danych. Po dodaniu nowego strumienia jest on widoczny w katalogu zasobów jako niepodłączony oraz z pustym metaschematem. Jeżeli użytkownik zbuduje zapytanie zasilane takim strumieniem, kompilator zgłosi błąd informujący o braku atrybutów, do których zdefiniowano odwołania.

Procedura przyjęcia zgłoszenia źródła danych:

- pobranie adresu zgłaszanego źródła,
- utworzenie wątku obsługującego wskazany kanał komunikacji,
- oczekiwanie na krotkę definiującą metaschemat strumienia,
- odbiór krotek z danymi.

W trakcie pracy strumienia możliwa jest zmiana metaschematu. W tym celu źródło przesyła krotkę systemową z nową definicją. Zarządca warstwy zasilania aktualizuje wtedy zawartość katalogu zasobów oraz przekazuje tę krotkę do strumieni wewnątrz systemu. Dzięki temu uruchomione zapytania zasilane tym źródłem informowane są o aktualizacji.

Katalog zasobów – repozytorium dostępnych w systemie strumieni.

Nadzorca zapytań – zapytanie strumieniowe to długookresowy proces, dlatego rolą nadzorcy zapytań jest nie tylko kontrolowanie przebiegu kompilacji, ale również nadzór uru-

chomionej sieci przetwarzania. Aby uprościć zarządzanie uruchomionymi sieciami przetwarzania, przyjęto, że zapytania są rejestrowane pod wybraną przez użytkownika nazwą. Dzięki temu w chwili wystąpienia awarii łatwo odnaleźć jej przyczynę na podstawie logów historii gromadzonych dla każdego zapytania oddzielnie. Przetwarzanie strumieniowe jest procesem wielowątkowym, dlatego moduł ten implementuje procedurę bezpiecznego wyłączenia zapytania, która gwarantuje zwolnienie wszystkich zasobów systemowych zajętych przez daną sieć przetwarzania. Kolejnym zadaniem nadzorcy zapytań jest obsługa komunikatów specjalnych generowanych przez sieć przetwarzania.

Sygnalizują one:

- wykrycie zmiany metaschematu jednego ze strumieni źródłowych,
- usunięcie strumienia źródłowego z katalogu strumieni,
- odbiór wyjątku w trakcie pracy węzła przetwarzającego.

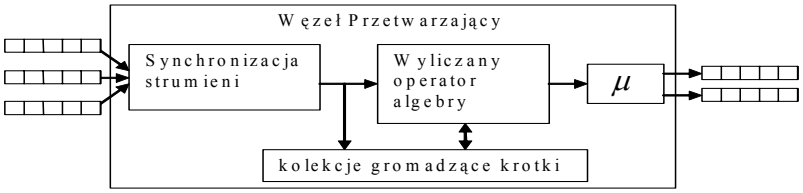
Jeżeli zapytanie nie definiuje metody obsługi komunikatu, nadzorca przystępuje do wyłączenia sieci przetwarzania.

Kompilator zapytania – kompiluje zapytanie do postaci sieci przetwarzania strumieniowego. Potrzebne informacje o strukturach strumieni pobiera z katalogu zasobów, z kolei wykryte błędy w zapytaniu kieruje do nadzorcy zapytania.

Sieć przetwarzania – przy użyciu operatorów fizycznych implementuje operacje logiczne zdefiniowane poprzez zapytanie w postaci tekstowej. Sieć przetwarzania to węzły przetwarzające połączone strumieniami danych. Węzeł przetwarzający jest platformą, na której uruchamiane są operatory fizyczne. Architektura taka w przejrzysty sposób rozdziela funkcjonalność wykorzystywaną przez większość operatów fizycznych (taką jak porządkowanie krotek strumieni wejściowych, obsługa komunikatów błędów) od konkretnej realizacji operatora fizycznego. Ponieważ przetwarzanie strumieniowe jest nieblokowalane, każdy z węzłów przetwarzających pracuje w ramach oddzielnego wątku. Przetwarzanie strumieniowe jest nieblokowalne, tzn. dany operator nie przetwarza całego zbioru danych wejściowych, zanim przekaże wyliczone wyniki do kolejnego operatora w planie produkcji. Proces przetwarzania strumieniowego dzięki zastosowaniu okien czasowych jest nieblokowalny, czyli dla każdej krotki wejściowej operator potrafi natychmiast wyliczyć wynik.

2.1. Architektura węzła przetwarzającego

Aby strumienie wewnątrz pojedynczego zapytania działały wydajnie, każdy z nich transportuje minimalny zbiór atrybutów koniecznych do ukończenia przetwarzania zapytania.



Rys. 2. Przepływ krotek w węźle przetwarzającym
Fig. 2. Tuples data flow in the processing node

Cel ten jest osiągany przez wprowadzenie operacji mapującej na wyjściu każdego węzła przetwarzającego.

Na rys. 2. przedstawiono przepływ krotek w trakcie wyliczania zapytania. Scenariusz działania jest następujący:

1. Uporządkowanie leksykograficzne krotek, jeżeli operator posiada kilka strumieni wejściowych.
2. Reorganizacja kolekcji oparta na znaczniku czasu krotki przekazanej do przetworzenia.
3. Wstawienie krotki do kolekcji.
4. Realizacja operacji algebry.
5. Wyznaczenie krotki wynikowej (wstawienie do strumienia wynikowego krotki ze zredukowaną liczbą atrybutów).

Jeżeli dana kolekcja zasila dodatkowo indeks, zamiany zachodzące w wyniku wstawienia krotki i reorganizacji struktury zgłaszane są przy użyciu wydarzeń: insert, update, delete.

2.2. Strumienie danych

Niech $I := \{[t_s, t_e) \in T \times T \mid t_s \le t_e\}$ oznacza zbiór przedziałów czasowych włącznie z punktami czasowymi, gdzie T jest dziedziną czasu, wprowadzamy również Ω, która oznacza przestrzeń dostępnych atrybutów.

Definicja 1.

Strumień opisuje para symboli $S = (M, \le_{t_s,t_e})$, gdzie:

M – nieskończony strumień krotek $(type, e, [t_s, t_e))$, gdzie: *type* – typ krotki, *e* – dane transportowane przez krotkę $[t_s, t_e) \in I$,

$\le_{t_s,t_e}$ – uporządkowanie leksykograficzne elementów strumienia *M* (najpierw po t_s, potem po t_e).

W systemie rozróżniamy kilka typów krotek (tab. 1), przez co możliwe jest zarządzanie siecią przetwarzania przy użyciu strumieni źródłowych.

Tabela 1

Typy krotek w systemie

Typ	Opis
HEADER	definiuje metaschemat krotek INSERTION
REMOVE	informuje o usunięciu strumienia z systemu
STOP	wstrzymuje działanie operatorów
INSERTION	służy transmisji wartości atrybutów
BOUNDARY	znacznik upływu czasu

2.3. Operatory

Logika operatorów algebry bazuje na architekturze zaproponowanej w [3]. Aktualnie system wspiera operatory: filtracji (σ), mapowania (μ), łączenia ($\triangleright\triangleleft$), okien czasowych ($\omega$), unii ($\bigcup$), agregacji ($\alpha$).

2.4. Operacja reorganizacji

Reorganizacja wyzwalana jest przez kolejne krotki przekazywane węzłowi na wejście, zanim zostaną jeszcze przetworzone. Jej zadaniem jest oczyszczanie operatora z elementów, których czas życia upłynął [3].

Niech $S_1,...,S_n \in S$, gdzie $n \in N$ są strumieniami wejściowymi operatora. Na potrzebę operacji reorganizacji przechowujemy najstarsze znaczniki czasu t_{s_j}, gdzie $j \in \{1,...,n\}$ krotek kolejnych strumieni wejściowych oczekujących na przetworzenie. Każda krotka $(e,[t_s,t_e))$ może zostać bezpiecznie usunięta ze struktur wewnętrznych operatora, jeżeli jej t_e jest mniejsze lub równe $\min\{t_{s_j} \mid j \in \{1,...,n\}\}$.

Wywołanie reorganizacji przed wykonaniem właściwej operacji węzła pozwala przyspieszyć wyliczania wyników, ponieważ operator nie musi wówczas sprawdzać krotek, co do których mamy pewność, że nie będą tworzyły wyniku.

2.5. Kolekcje krotek

Kolekcje służą przechowywaniu elementów pochodzących z jednego strumienia. Występują one tylko w węzłach reprezentujących operatory stanowe. Po uwzględnieniu faktu, że każda krotka ma ustalony czas życia oraz kolekcje mogą zasilać indeksy, obiekt kolekcji krotek udostępnia następującą funkcjonalność:

1) przeglądanie zawartości zgodnie z porządkiem leksykograficznym,
2) wstawianie krotek,
3) realizacja operacji reorganizacji,
4) dostęp do krotek poprzez identyfikator numeryczny,
5) zgłaszanie wydarzeń o zmianie zawartości kolekcji:
   - usunięcie wpisu,
   - wstawienie wpisu,
   - aktualizacja wpisu,
   - reorganizacja.

Dodanie indeksów wyliczających relacje wiąże się z potrzebą zdefiniowania atrybutu kluczowego. Jest on wartością numeryczną, jeżeli krotka nie definiuje takiego atrybutu, wówczas kolekcja automatycznie nadaje wartość unikalną kluczowi. Aby zapewnić spójność danych w węźle, aktualizacja indeksu jest wywoływana natychmiast po aktualizacji kolekcji, na której dany indeks jest zbudowany.

Prezentowany system wspiera kolekcje:
1) czasowe,
2) czasowo-tabelaryczne.

Kolekcja czasowa:
- wstawianym krotką przypisywane są automatycznie unikalne wartości klucza,
- czas życia obiektów zależy od wartości atrybutów t_s, t_e krotek,
- elementy kolekcji są usuwane w wyniku wywołania operacji reorganizacji.

Kolekcja czasowo-tabelaryczna:
- czas życia obiektów zależy od wartości atrybutów t_s, t_e krotek,
- elementy kolekcji są usuwane w wyniku wywołania operacji reorganizacji,
- krotka zawiera atrybut kluczowy, który steruje funkcjami: wstawiania, aktualizacji, zmiany czasu życia krotki.

Przekazanie do kolekcji czasowo-tabelarycznej krotki o atrybucie kluczowym *id* skutkuje operacją:
- id krotki nie istnieje w zbiorze – krotka jest dodana do zbioru,
- id krotki istnieje w zbiorze – krotka poprzednia jest zastąpiona nową,
- id krotki jest liczbą ujemną – atrybut tej krotki będącej w kolekcji jest zastępowany atrybutem tej nowej krotki.

Importowanie rekordów z tabel można zrealizować przez skopiowanie wartości atrybutów do krotki, ustawienie znacznika t_s na bieżący czas oraz $t_e = \infty$. Operacja usuwania rekordu odpowiada wysłaniu krotki o ujemnym identyfikatorze oraz przekazaniu informacji o czasie usunięcia przez atrybut t_e. Właściwe usunięcie elementu z kolekcji ma miejsce dopiero podczas fazy reorganizacji. Jeżeli zostanie wysłana krotka o identyfikatorze istniejącym

już w kolekcji, zastąpiony zostanie poprzedni wpis nowym, włącznie z aktualizacją znaczników czasu. Każda operacja na krotce kolekcji wiąże się z aktualizacją znacznika t_s, co ma na celu zagwarantowanie niezmienności prefiksu krotek przetworzonych przez operator (zmiany nie obejmują historii).

2.6. Uporządkowanie krotek w strumieniach

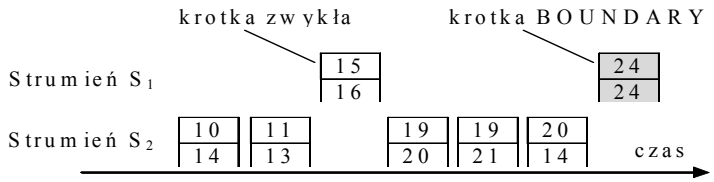
Definicja 2.

Prefiks sekwencji krotek: Podsekwencja krotek rozpoczyna się od najstarszej, a kończy na krotce ze znacznikiem $[t_s,t_e)$ (korzystamy z porządku leksykograficznego).

Operatory zasilane kilkoma strumieniami wejściowymi wymagają przetwarzania krotek w porządku leksykograficznym. Powyższa własność gwarantuje finalność generowanych wyników, ponieważ nie modyfikujemy prefiksu krotek dla $[t_s,t_e)$.

Aby uniknąć konieczności przesyłania replik krotek w strumieniach reprezentujących zjawiska o małej zmienności oraz w celu utrzymania płynności przetwarzania, wprowadzono krotkę typu BOUNDARY [2]. Pełni ona rolę mechanizmu „bicia serca". Znacznik czasowy krotki tego typu $(e,[t_s,t_s))$ informuje, że kolejne elementy w strumieniu nie będą posiadały dat wcześniejszych. Podobnie jak dla operacji reorganizacji, przechowujemy najstarsze znaczniki $[t_{s_j},t_{e_j})$, gdzie $j \in \{1,...,n\}$ krotek kolejnych strumieni wejściowych oczekujących na przetworzenie. Kolejna krotka pobrana ze strumienia wejściowego i przekazana do dalszego przetwarzania spełnia warunek:

$$[t_s,t_e) \leq_{t_s,t_e} \min\{[t_{s_j},t_{e_j}) \mid j \in \{1,...,n\}\}$$



Rys. 3. Porządkowanie krotek strumieni wejściowych. Liczby w wierszach oznaczają odpowiednio *ts*, t_e

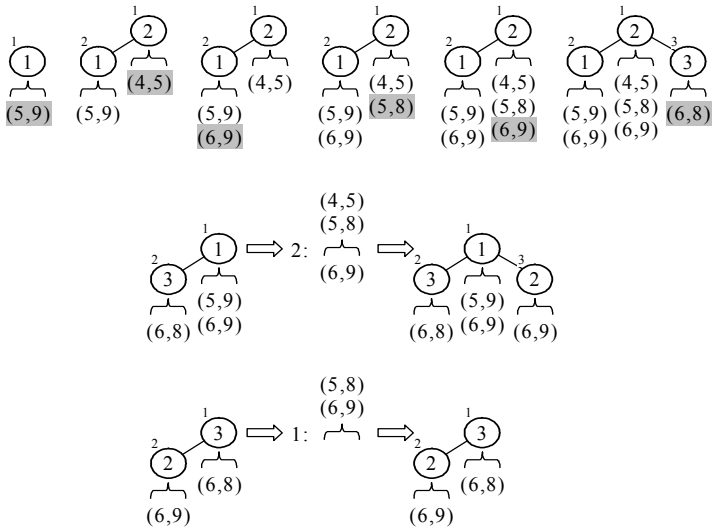
Fig. 3. Sorting the input streams tuples. The numbers in boxes denotes *ts*, t_e

Po przetworzeniu kolejek wejściowych z rys. 3 pozostaną w kolejkach ostatnie obiekty. Krotka BOUNDARY pozwoliła na przekazanie do dalszego przetworzenia obiekty o $t_s = 19$, 19, 20. Mechanizm ten rozwiązuje problem zachowania płynności przetwarzania, w przypadku gdy strumienie rzadko nadsyłają krotki z danymi.

2.6.1. Realizacja sortowania krotek strumieni wejściowych

Ponieważ liczba strumieni wejściowych węzłów nie ulega zmianie w trakcie działania zapytania, a krotki są uporządkowane w ramach pojedynczego strumienia, implementację synchronizacji strumieni oparto na sortowaniu przez kopcowanie.

Na rysunku 4 zaprezentowano kolejne kroki algorytmu porządkującego leksykograficznie krotki wejściowe. Pojedynczy element strumienia opisany jest dwoma liczbami oznaczającymi znacznik początku i końca życia. Liczba w węźle kopca reprezentuje numer strumienia, z którym jest związany, z kolei indeksy nad węzłem położenie w tabeli kopca. Przedstawiony scenariusz działania utworzony został dla operatora o trzech wejściach. Dokładny opis realizacji sortowania został umieszczony w algorytmie 1.



Rys. 4. Przykład sortowania krotek przy użyciu kopca
Fig. 4. Example for input streams tuples heap sorting

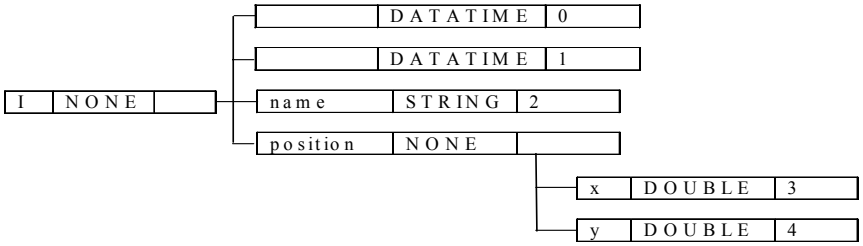
Algorytm 1. Sortowanie krotek strumieni wejściowych:

```
put(aSlot)
{
    int lastElem = mActiveNum;
    mHeap[lastElem] = aSlot;
    ++mActiveNum;

    fixUp(lastElem);
    while(mActiveNum == mSNum)//dopóki wszystkie strumienie zawierają dane
    {
          //źródło z najstarszymi krotkami
          tsSource = mInputStreams[mHeap[0]];
          //uporządkowanie kopca dla mActiveNum-1 elementów
          exch(0, lastElem);
          fixDown(0, lastElem-1);

          Tuple uptill = mInputStreams[mHeap[0]].peekNextTupleToProcess();
          //przetworzenie elementów stabilnych
          Tuple tuple = tsSource.peekNextTupleToProcess();
          while((tuple != null) && (tuple.compareDate(uptill) <= 0))
          {
                reorganise(tuple.getBeginDate());
```

```
                mOperator.mDomain.updateTuple(tuple.getSlot()+1, tuple);
                Tuple nextTuple = tsSource.nextTupleToProcess();

                //przetworzenie krotki
                if(tuple.getType() == TupleTypes.BOUNDARY)
                        mOperator.broadcast(tuple);
                else
                        mOperator.excecute(tuple);

                tuple = nextTuple;
        }

        if(tuple == null)
                --mActiveNum;
        else
                fixUp(lastElem);
    }
}
```

2.7. Drzewa atrybutów

Dane w języku zapytań reprezentowane są jako drzewa atrybutów. Etykieta korzenia wskazuje nazwę strumienia. Każdy z węzłów drzewa posiada nazwę, typ oraz identyfikator atrybutu krotki. Nie każdy węzeł struktury musi posiadać zdefiniowany typ atrybutu. Takie puste węzły tworzą strukturę danych czytelniejszą przez wprowadzenie zgrupowań. W poniższym przykładzie rolę tę pełni węzeł *position*.



Rys. 5. Przykład struktury drzewa atrybutów
Fig. 5. Example for attributes tree

Ponieważ każda krotka posiada znacznik początku oraz końca życia, ich identyfikatory mają wartości stałe kolejno 0, 1. Chcąc odczytać atrybut *I.position.x* (rys. 5), należy najpierw odczytać wartość identyfikatora atrybutu, następnie pobrać atrybut o wskazanym identyfikatorze z krotki strumienia.

Drzewo atrybutów jest reprezentowane przez zagnieżdżone listy atrybutów (podobnie jak w XML). Skorzystanie z definicji, w której rozpatrywalibyśmy drzewo jako zagnieżdżone wielozbiory [6], nie pozwoliłoby odwoływać się do atrybutów przy użyciu numeru porządkowego. W konsekwencji funkcje operujące na drzewach informacji mogłyby odwoływać się do atrybutów tylko przez etykietę atrybutu. Przyjęcie koncepcji zagnieżdżonych list atrybutów pozwala rozpoznawać atrybuty funkcji, zgodnie z numerem porządkowym (tak jak ma to miejsce w językach C/C++) lub na podstawie etykiety(analogicznie do XML).

Dla danej zagnieżdżonej listy etykiet *A* definiujemy listę *LT* atrybutów indeksowanych poprzez *I*, tak że:

- pusta lista etykiet {} należy do *LT*,
- jeśli *m* występuje w *A* oraz *I* w *LT*, wtedy istnieje para {<*m,I*>} w *LT*,
- *LT* jest unią zagnieżdżonych list $\bigcup_{j \in J} M(j)$, gdzie *J* jest zbiorem indeksów i $M \in J \rightarrow LT$.

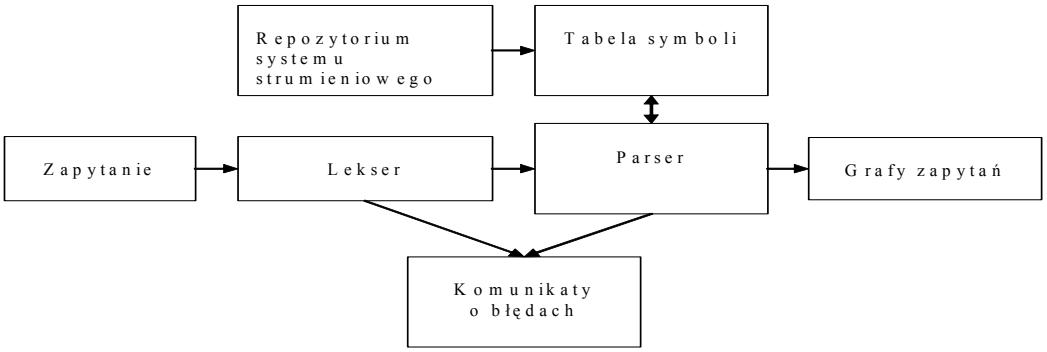
Przyjmując powyższą definicję drzewa etykiet, utworzono funkcję mapującą wyrażenie *A* na atrybuty krotki strumienia.

η ::=	etykieta węzła drzewa,
$nazwa	etykieta korzenia,
nazwa	etykieta węzła w bieżącym kontekście,
A, B ::=	wyrażenie,
T	poddrzewo wskazywane przez bieżący kontekst,
$\eta[A]$	lokacja bieżącego kontekstu na węźle η,
A , B	kompozycja drzew atrybutów.

Aby uprościć odwoływanie się do pojedynczych węzłów np. *O*[*z*[*x* = 1.1]], następujące po sobie klamry można zastąpić „.”. Otrzymamy wtedy postać *O.z.x* = 1.1.

3. Kompilator języka zapytań

Na rys. 6 przedstawiono przepływ sterowania podczas kompilacji zapytania. W odróżnieniu do języków programowania, języki zapytań odwołują się również do atrybutów niezdefiniowanych wewnątrz treści zapytania. Są nimi zmienne przechowywane w repozytorium systemu strumieniowego (np. lista dostępnych strumieni, metaschemat krotek strumieni, aktywność strumieni). Ponieważ w trakcie kompilacji zapytania informacje przechowywane w repozytorium systemowym mogą ulegać zmianą, aby zachować spójność tabeli symboli kompilatora, przyjęto, że na życzenie parsera kopiowane są kolejne struktury opisujące strumienie w repozytorium systemowym. Podejście takie wymaga podczas uruchamiania sieci przetwarzania dodatkowego zbadania, czy użyte metaschematy nie uległy zmianie od chwili przystąpienia do kompilacji.

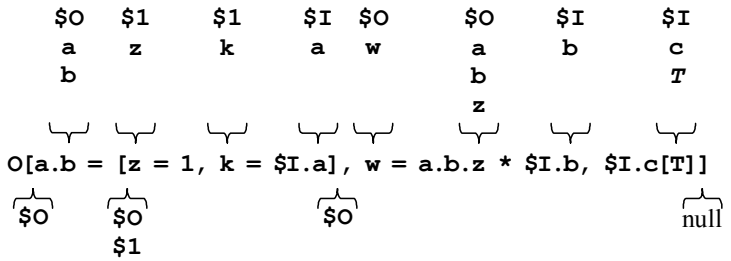


Rys. 6. Przebieg kompilacji zapytania
Fig. 6. Compilation process

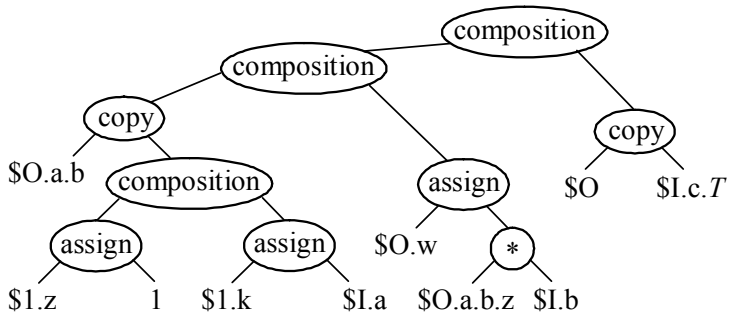
Zbudowany język korzysta z atrybutów ułożonych w hierarchie, wiąże się to z rozbudową parsera o dodatkowy stos, na który odkładana jest bieżąca pozycja kontekstu zmiennych. Każdy atrybut musi istnieć w tablicy symboli, zanim zostanie zdefiniowany operator realizujący na nim obliczenia. Aby to zapewnić, akcje semantyczne sterujące działaniem stosu kontekstu zostały umieszczone wewnątrz produkcji gramatyki, jak przedstawiono poniżej:

```
NodeDecl: QualifiedPath {wstaw ścieżkę} '=' Exp {zdejmij ścieżkę}
        …

 Level: QualifiedPath '[' {wstaw ścieżkę} NodesDecl ']'
 {zdejmij ścieżkę}
     | '[' {wstaw tymczasową zmienną} NodesDecl ']'{zdejmij ścieżkę}
     …
```

Składnia taka sprawia, że tabela symboli jest zasilana tylko przez parser (rys. 6). W języku zapytań przyjęto zasadę, że atrybut otrzymuje typ zgodny z wyliczeniem leżącym po prawej stronie wyrażenia. Po pierwszym zdefiniowaniu typu atrybutu nie dopuszcza się jego zmiany w dalszej części zapytania. Powyżej przedstawione produkcje gramatyki sprawiają, że znana jest pełna nazwa atrybutu hierarchicznego, co pozwala przeprowadzać na bieżąco kontrolę typów wyliczeń. Aby przybliżyć działanie parsera, na rys. 7 zilustrowano zawartość stosu kontekstu (kubełki pod wyrażeniem) wraz z przesuwaniem się miejsca analizy wyrażenia. Kubełki u góry oznaczają pełną nazwę zmiennych. Na rys. 8 umieszczono drzewo rozbioru wyrażenia z rys. 7.



Rys. 7. Zawartość stosu kontekstu drzewa atrybutów dla kolejnych momentów rozbioru wyrażenia
Fig. 7. The history of attribute tree context stack during the analysis of the example



Rys. 8. Drzewo rozbioru dla O[a.b = [z = 1, k = $I.a], w = a.b.z * $I.b, $I.c[T]]
Fig. 8. Abstract syntax tree for example O[a.b = [z = 1, k = $I.a], w = a.b.z * $I.b, $I.c[T]]

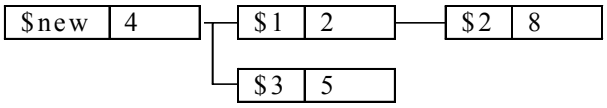
Dodatkowego omówienia wymagają struktury tymczasowe (np. [z=1, k=$I.a]). Aby kompilator mógł jednoznacznie odwoływać się do ich atrybutów, posiadają one unikalne nazwy. W momencie rozpoznania składni definiującej strukturę tymczasową zostaje skierowane zapytanie do zarządcy tabeli symboli o podanie nowej unikatowej nazwy. Ponieważ nazwy atrybutów nie mogą rozpoczynać się od cyfry, przyjęto, że kolejne liczby całkowite będą używane jako nazwy korzeni struktur tworzonych przez kompilator.

3.1. Zarządzanie identyfikatorami atrybutów

Aby móc przeprowadzić obliczenia zgodnie z otrzymanym drzewem rozbioru, należy zdefiniować politykę zarządzania pamięcią atrybutów. Odczyt/zapis atrybutów krotek wejściowych i wyjściowych realizowany jest bezpośrednio na elementach składowych krotek. Z kolei atrybuty struktur pomocniczych są alokowane w krotce wynikowej, a następnie usuwane przed przekazaniem jej do strumienia wyjściowego. W krotce wynikowej wyróżniamy wartość ostatniego identyfikatora wchodzącego w skład wyniku, wartości wyższe są dostępne dla atrybutów tymczasowych. Podział taki pozwala zaimplementować szybki mechanizm oczyszczania. Zmienne pomocnicze mogą być zagnieżdżane, co dodatkowo komplikuje mechanizm nadawania identyfikatorów. Dlatego utworzono strukturę reprezentującą hierarchię drzew atrybutów. Na rys. 9 umieszczono przykładowe drzewo, gdzie krawędzie reprezentują wzajemną relację zagnieżdżenia drzew atrybutów.

Jak widzimy, struktura tymczasowa *$3* nie jest związana z *$1* i *$2*. Czyli w trakcie wyliczania atrybutów struktury *$3* nie będą przechowywane wyniki cząstkowe atrybutów struktur *$1* i *$2*. Obserwacja ta skutkuje utworzeniem mechanizmu nadawania identyfikatorów według następującego scenariusza. Każda zmienna na tym samym poziomie otrzymuje pule identyfikatorów od pierwszego wolnego numeru następującego po wypełnieniu struktury ojca. Dla powyższego przykładu struktura *new* otrzyma pulę identyfikatorów 1..4, struktura *$1*: 5,6; *$2*: 7,...,14; *$3*: 5,...,9. Taka procedura numeracji pozwala zredukować tablicę atrybutów potrzebną do rozwiązania danego wyrażenia. Ponadto, chcąc oczyścić strukturę z atrybutów tymczasowych, wystarczy usunąć atrybuty o identyfikatorach wyższych od wskazanego

numeru. Należy zauważyć, że nadawanie identyfikatorów zachodzi na zakończenie działania kompilacji wyrażenia, ponieważ dopiero wtedy znana jest pełna definicja drzewa atrybutów.



Rys. 9. Drzewo odwołań do struktur. (pierwsza kolumna reprezentuje nazwę struktury, druga kolumna reprezentuje liczbę atrybutów zdefiniowanych wewnątrz struktury)
Fig. 9. The tree of the attributes tree. (the node's elements denote the attribute tree name and the number of attributes into it)

Algorytm 2. Nadawanie identyfikatorów drzewom atrybutów:

```
integer przypiszIdentyfikatory ( integer wartość_id, węzeł_drzewa node )
for element in node.lista_węzłów_potomnych
{
   if( element.typ_atrybutu != brak)
   {
         element.identyfikator = wartość_id
         ++wartość_id
   }
   wartość_id  = przypiszIdentyfikatory (wartość_id, node)
}
return wartość_id
```

Algorytm 3. Sterowanie nadawania identyfikatorów poprzez drzewo hierarchii:

```
nadajIdentyfikatory(integer wartość_id, węzeł_hierarchi node )
wartość_id = przypiszIdentyfikatory(wartość_id, node.drzewo_atrybutów)
for element in node.lista_węzłów_potomnych
{
   nadajIdentyfikatory(wartość_id, element)
}
return wartość_id
```

3.2. Kompilacja wyrażeń arytmetycznych

Zbudowane przez parser drzewo rozbioru nie definiuje w pełni realizacji operatorów. Pełną definicję otrzymujemy po uwzględnieniu typów danych źródłowych, dostępnych rzutowań oraz informacji, czy dana operacja jest zdefiniowana dla danego typu. Istotne jest, aby proces ten można wykonać przyrostowo, co upraszcza implementację mechanizmu identyfikacji i zgłaszania błędów kompilacji. Do osiągnięcia tego celu wystarczy, aby konstrukcja drzewa rozbioru przebiegała od dołu w górę, wtedy kolejno dodany operator dysponuje w chwili utworzenia definicją operatorów źródłowych. Właściwość tę spełnia zaproponowana gramatyka języka zapytań.

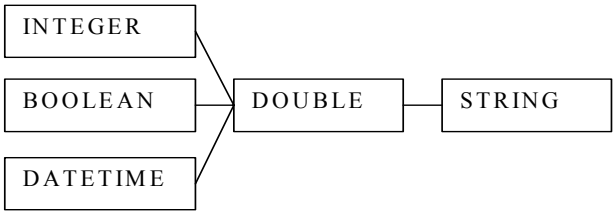
Algorytm 4. Scenariusz kompilacji operatorów arytmetycznych:

```
1. Pobranie typów wyliczeniowych atrybutów.
1.1 Jeżeli atrybutem operatora jest operacja odczytu z krotki, następuje
    sprawdzenie, czy atrybut, do którego się odwołuje, ma zdefiniowany typ
    wyliczeniowy. Jeżeli brak: poinformowanie o błędzie.
1.2 Utworzenie operatorów odczytu atrybutów z krotek.
```

```
3. Wyznaczenie typu wyliczeniowego dla kompilowanego operatora.
4. Sprawdzenie, czy są dostępne rzutowania typów atrybutów wejściowych na typ
   wyliczeniowy operatora. Jeżeli brak: powiadomienie o braku operatora dla
   danych argumentów.
5. Utworzenie i wstawienie operatorów realizujących operację rzutowania typów.
6. Utworzenie operatora realizującego zadaną operację i połączenie z operatorami
   atrybutów.
7. Zapisanie informacji o typie wynikowym operatora.
```

Wyznaczenie typu wyliczeniowego operatora polega na ustaleniu typu operatora arytmetycznego. Dla wyrażenia:*12.2*2* mamy argument DOUBLE i INTEGER. Ponieważ DOUBLE reprezentuje typ ogólniejszy od INTEGER, typem wyliczeniowym operatora będzie DOUBLE.

Do ustalenia typu wyliczeniowego operatora służy drzewo rzutowań domyślnych (rys. 10):



Rys. 10. Definicja rzutowań domyślnych
Fig. 10. Default cast definition

Wyznaczenie typu wyliczeniowego polega na ustaleniu najbliższego wspólnego typu przesuwając się po drzewie w prawą stronę, np. dla typów INTEGER i BOOLEAN najbliższym wspólnym typem jest DOUBLE. Drzewo to nie oznacza, że rzutowanie typu BOOLEAN na STRING to złożenie BOOLEAN->DOUBLE->STRING. Język nie łączy kaskadowo rzutowań, ponieważ mogłoby to prowadzać do nieoczekiwanych przez użytkownika wyników oraz niejasności. Dla rozważanego rzutowania BOOLEAN na STRING kompilator skorzysta z bezpośredniej konwersji typów.

Tabela 2
Wartościowanie dla rzutowania BOOLEAN -> STRING

BOOLEAN	STRING
True	True
False	False

Tabela 3
Wartościowanie dla rzutowania BOOLEAN -> DOUBLE

BOOLEAN	DOUBLE
True	1.0
False	0.0

Analogicznie utworzone jest rzutowanie BOOLEAN -> INTEGER

Dla rzutowania DATATIME-> DOUBLE przyjęto, że jest to liczba milisekund czasu UTC od początku epoki. Po skompilowaniu operatora następuje zapamiętanie typu wyniko-

wego, który może być różny od typu wyliczeniowego, np. dla *12 < 13* typem wyliczeniowym jest INTEGER, a typem wynikowym BOOLEAN.

Tabela 4

Zdefiniowane operatory

Operator	Zapis operatora oraz opis
VAL	odczyt wartości (służy realizacji rzutowania typów)
EQ	==
GE|EQ	>=
GE	>
LO|EQ	<=
LO	<
LO|GE	!=
ADD	+
SUB	-
MUL	*
DIV	/
AND	AND
OR	OR
NOT	NOT
XOR	XOR

Tabela 5

Dostępność operacji dla wskazanych typów wyliczeniowych

Operator	INTEGER	DATATIME	BOOLEAN	DOUBLE	STRING
VAL	●	●	●	●	●
EQ	●	●	●	●	●
GE|EQ	●	●		●	●
GE	●	●		●	●
LO|EQ	●	●		●	●
LO	●	●		●	●
LO|GE	●	●		●	●
ADD	●			●	●
SUB	●			●	
MUL	●			●	
DIV	●			●	
AND	●		●		
OR	●		●		
NOT	●		●		
XOR	●		●		

3.3. Funkcje agregacyjne w zapytaniach

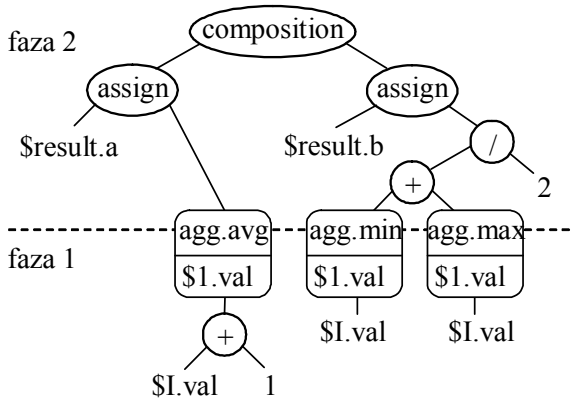
W frazie *select* można definiować agregaty algebraiczne, tzn. wartość agregatu jest wyliczana jako skalarna funkcja agregatów cząstkowych, przykładem jest wyliczenie średniej AVG=SUM/COUNT. Fraza *select* zawierająca funkcje agregacyjne wyliczana jest dwueta-

powo. Najpierw wyznaczane są wartości agregatów cząstkowych, następnie agregaty algebraiczne. W najprostszym przypadku druga faza to operacja przypisania wartości agregatu cząstkowego do atrybutu wynikowego [7].

W celu wyróżnienia operacji składających się na pierwszą i drugą fazę zdefiniowano algorytm cięcia drzewa rozbioru. Zasadę działania ilustruje poniższy przykład.

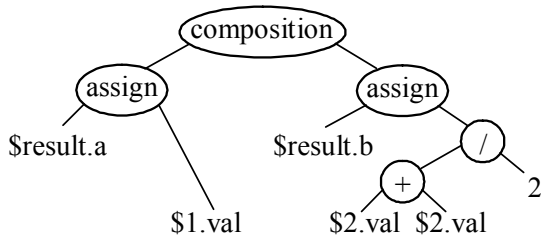
```
result[a = agg.avg($I.val + 1.0), b = (agg.min($I.val) + agg.max($I.val))/2]
```

Na rys. 11 zamieszczono drzewo rozbioru dla rozpatrywanego wyrażenia. Linia przerywana przechodzi przez operatory realizujące operacje agregacji, jej bieg wskazuje podział na operacje wchodzące w skład fazy pierwszej (dół) i drugiej (góra). Aby przeprowadzić kompilację takiego wyrażenia, w chwili wstawiania operatora agregacji tworzona jest równocześnie struktura tymczasowa, która będzie przechowywała wartości agregatów cząstkowych. Po zdefiniowaniu drzewa rozbioru następuje identyfikacja operatorów agregacji, po czym zostają one odcięte wraz z poddrzewami, a na ich miejsce wstawione operacje odczytu wartości ze struktur tymczasowych przechowujących wartości agregatów cząstkowych. Na rys. 12 przedstawiono wynik podziału drzewa rozbioru z rys. 11 uzupełniony o operacje odczytu ze struktur tymczasowych. Odcięte poddrzewa umieszczane są w liście definiującej operator agregacji, każde z nich jest rozszerzone o operację zapisu wyniku agregatu cząstkowego (rys. 13).

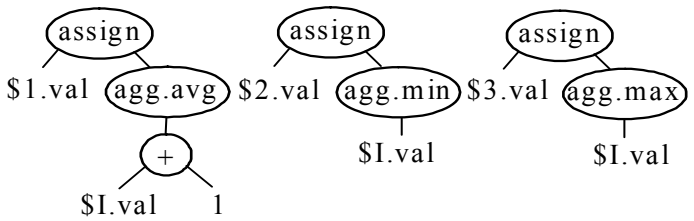


Rys. 11. Drzewo rozbioru dla result[a = agg.avg($I.val + 1.0), b = (agg.min($I.val) +agg.max($I.val))/2]

Fig. 11. Abstract syntax tree for example result[a = agg.avg($I.val + 1.0), b = (agg.min($I.val) +agg.max($I.val))/2]



Rys. 12. Drzewo rozbioru dla fazy II
Fig. 12. Abstract syntax tree for the phase II

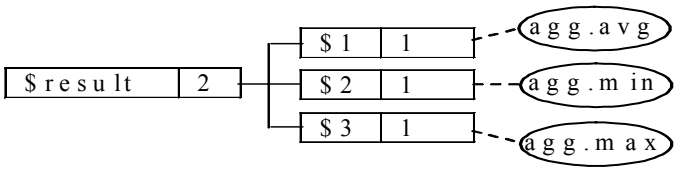


Rys. 13. Drzewo rozbioru dla fazy I
Fig. 13. Abstract syntax tree for the phase I

Jak widzimy na przykładzie, argumentem funkcji agregacyjnej może być wyliczenie. Sprawia to, że należy rozważyć przypadek, kiedy wewnątrz wyliczenia korzystalibyśmy ze struktur tymczasowych. Użycie ich wiąże się z nadawaniem identyfikatorów, zgodnie z algorytmem zarządzania identyfikatorami atrybutów. Do jego uruchomienia konieczna jest znajomość drzewa hierarchii struktur, dlatego operatory agregacji zawierają odnośniki do węzłów hierarchii struktur reprezentujących ich struktury wynikowe. Na rys. 14 przedstawiono drzewo hierarchii struktur atrybutów dla analizowanego przykładu. Dzięki temu, że każda funkcja dysponuje powyższym odnośnikiem, operację podziału drzewa rozbioru oraz reorganizację hierarchii drzew atrybutów można przeprowadzić jednocześnie. Wystarczy podczas przenoszenia operatora agregacji z drzewa rozbioru wyrejestrować skojarzoną z nim hierarchię atrybutów wynikowych i przenieść ją do kontekstu operatora wyliczającego agregaty. O ile nadanie identyfikatorów atrybutom wynikowym dla fazy II jest proste, ponieważ dysponujemy drzewem hierarchii struktur, o tyle dla fazy I należy je dopiero zbudować.

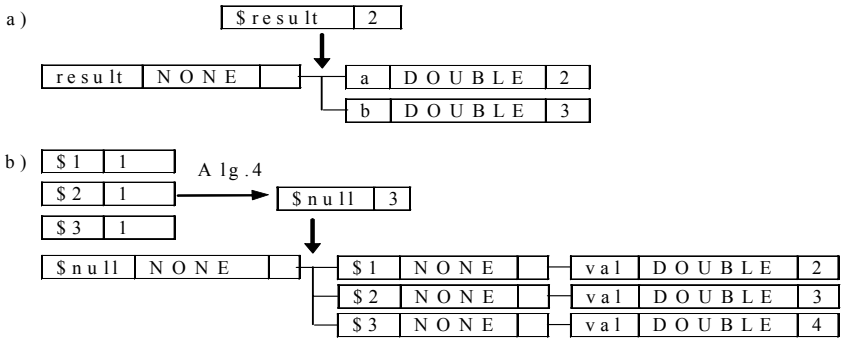
Algorytm 5. Definiowanie metaschematu strumienia wyjściowego dla operatora agregacji oraz nadanie identyfikatorów atrybutom:

```
1. Utwórz puste drzewo atrybutów ATREE.
2. Utwórz korzeń hierarchii drzew atrybutów HROOT wskazujący na ATREE.
3. Pobierz pierwszy/kolejny operator agregacji OP; jeżeli brak, przejdź
   do punktu 8.
4. Pobierz korzeń hierarchii drzew atrybutów OP_HROOT  dla OP.
5. Wstaw do ATREE drzewo atrybutów skojarzone z OP_HROOT.
6. Wstaw do  HROOT węzły potomne OP_HROOT, przejdź do punktu 3.
6. Nadaj identyfikatory drzewom atrybutów zawartych w HROOT, zgodnie
   z algorytmem.
```



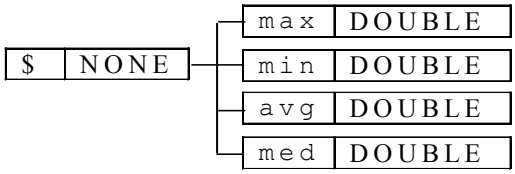
Rys. 14. Hierarchia drzew atrybutów dla analizowanego przykładu
Fig. 14. Hierarchy of attributes trees for the example

Na rys. 14 przedstawiono kolejne etapy tworzenia metaschematów operatorów oraz nadawania identyfikatorów atrybutom. Ponieważ w wyrażeniach zasilających funkcje agregacyjne nie ma struktur tymczasowych, więc hierarchia struktur (rys. 14b) dla operatora agregacji składa się z pojedynczego węzła.



Rys. 15. Definiowanie metaschematu strumieni wyjściowych: a) dla operatora wynikowego, b) dla operatora agregacji
Fig. 15. The process of defining stream's meta-schema for: a) final mapping operator, b) intermediate aggregation operator

Zaproponowaną reprezentację wyników funkcji agregacyjnych wyróżnia to, że w prosty sposób można przekazać grupę atrybutów do struktury wynikowej. Zaprezentowane funkcje umieszczały wynik w pojedynczym węźle o strukturze <*nazwa korzenia struktury*>.*val*. Jeżeli istniałaby potrzeba utworzenia bardziej złożonej funkcji agregacyjnej, np. wyliczającej kilka wartości: minimum, maksimum mediana i wartość średnią w celu prowadzenia analizy statystycznej jakiegoś wskaźnika, zbudowana na tę potrzebę funkcja mogłaby definiować strukturę wynikową o postaci (rys. 16):



Rys. 16. Struktura wynikowa złożonej funkcji agregacyjnej
Fig. 16. Structure of more complicated aggregation function

Korzystanie z drzewa atrybutów agregatu wiąże się z potrzebą jego wstępnego skopiowania do wynikowego drzewa atrybutów, jest to konieczne w celu zdefiniowania pełnej ścieżki

do struktury. Następnie odczyt atrybutów agregatu odbywa się tak samo, jak ma to miejsce dla zwykłych atrybutów w ramach krotki wynikowej.

3.4. Indeks agregatów

Zaimplementowany indeks agregatów to lista agregatów, aktualizowana inkrementacyjnie kolejnymi krotkami strumienia wejściowego. Każdy z agregatów ma ze sobą skojarzony interwał czasu, przez który obowiązuje. Lista agregatów jest systematycznie czyszczona z elementów wygasłych, które są następnie przekazywane do strumienia wynikowego.

Niech $S = (M, \leq_{t_s,t_e})$ jest strumieniem wejściowym operatora. Zdefiniujmy funkcję: $f : \Omega \cup \{\perp\} \times M \to \Omega$, której argumentami są: bieżąca wartość agregatu $\breve{s} \in \Omega \cup \{\perp\}$ oraz wartość, o jaką następuje aktualizacja $s = (e, [t_s, t_e)) \in M$, gdzie $\{\perp\}$ służy inicjalizacji pustego agregatu. Na początku następuje identyfikacja, które agregaty w indeksie nachodzą na interwał $[t_s, t_e)$. W przypadku częściowego pokrycia wykonany jest podział elementu na dwa interwały: część pokrytą i wolną, której wartość pozostaje bez zmian. Następnie aktualizowane zostają wartości elementów nachodzących na interwał $[t_s, t_e)$. Dodatkowo każdy podinterwał $r \in [t_s, t_e)$, dla którego nie odnaleziono struktury z bieżącą wartością agregatu, zostaje wstawiony do indeksu z wartością równą $f(\perp, s)$ i czasie życia *r*.

W trakcie czyszczenia listy agregatów (operacja reorganizacji) usuwane są wszystkie elementy, których znaczniki końca życia są mniejsze lub równe t_s aktualnie przetwarzanej krotki wejściowej, po czym usunięte elementy przekazywane są do listy wynikowej. Elementy indeksu agregatów są uporządkowane zgodnie z $\leq_{t_s,t_e}$, dzięki temu otrzymana lista agregatów w wyniku operacji reorganizacji może zasilić strumień wynikowy bez potrzeby dodatkowego sortowania (wymóg porządku leksykograficznego dla krotek strumieni).

Przedstawiony algorytm nie gwarantuje postaci znormalizowanej generowanych wyników. Oznacza to, że mogą po sobie następować krotki o tych samych wartościach agregatów aniżeli pojedynczy agregat. Aby uniknąć takiego zjawiska, które zbędnie obciąża strumienie systemu, w otrzymanej liście agregatów następuje łączenie następujących po sobie elementów o tych samych wartościach, po czym znormalizowana lista agregatów jest przesyłana do strumienia wynikowego.

Prezentowany indeks nie przechowuje krotek źródłowych, co uniemożliwia zdefiniowanie operacji usuwania wartości dla agregatów typu min, max. W przypadku agregatów count, sum i avg realizacja funkcji usuwania polega na wstawianiu elementu o wartości przeciwnej. Ponieważ zastosowanie okna typu czasowo-tabelarycznego pozwala na wystąpienie operacji usuwania krotki, należało zdefiniować działanie agregatów typu min/max dla powyższej sy-

tuacji. Przyjęto, że wywołanie operacji usuwania wpisu dla takich agregatów nie powoduje zmiany zawartości indeksu.

Aby łatwiej zrozumieć wyliczanie agregatów przy użyciu alg. 6 i 7, poniżej zamieszczono przebieg wypełniania i oczyszczania listy agregatów przy użyciu powyższych algorytmów. Dodatkowo, na rys. 18 przedstawiono graficzny przebieg aktualizacji listy agregatów.

wejście: (<1,4), -1)

reorganise: 1

Interwał	Agregat

wyjście:

change: (<1,4), -1)

Interwał	Agregat
<1,4)	-1

wejście: (<1,6), -4)

reorganise: 1

Interwał	Agregat
<1,4)	1

wyjście:

change: (<1,6), -4)

Interwał	Agregat
<1,4)	-5
<4,6)	-4

wejście: (<1,8), 7)

reorganise: 1

Interwał	Agregat
<1,4)	-5
<4,6)	-4

wyjście:

change: (<1,8), 7)

Interwał	Agregat
<1,4)	2
<4,6)	3
<6,8)	7

wejście: (<3,7), 2)

reorganise: 3

Interwał	Agregat
<3,4)	2
<4,6)	3
<6,8)	7

wyjście: (<1,3),2)

change: (<3,7), 2)

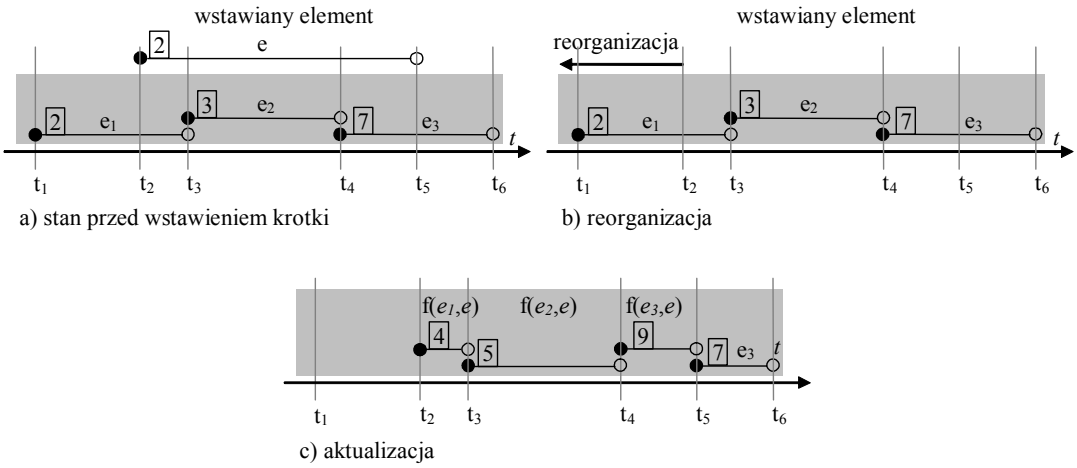
Interwał	Agregat
<3,4)	4
<4,6)	5
<6,7)	9
<6,8)	7

Rys. 17. Wyznaczanie agregatów SUM dla krotek wejściowych: (<1,4),-1), (<1,6),-4), (<1,8),7), (<3,7),2).

Fig. 17. The history of calculation SUM aggregates for example input tuples: (<1,4),-1), (<1,6),-4), (<1,8),7), (<3,7),2).

Algorytm 6. Reorganizacja indeksu agregatów:

```
lista_agregatów reorganise(time now )

element = lista_agregatów.pierwszy_element()
last_time = element.end
while(element != null and last_time <= now)
{
   lista_agregatów.usuń(element)
   wynik.wstaw(element)
   last_time = element.end
   element = lista_agregatów.pierwszy_element()
}
if(element != null and element.contain(now))
{
   new_element = copy(element)
   new_element.end = now
   wynik.wstaw(new_element)
   element.start = now
}
if(last_time < now)
{
   agg = wartość początkowa agregatu
   agg.interwał(last_time, now)
   wynik.wstaw(agg)
}
return wynik
```



Rys. 18. Przykład aktualizacji agregatów
Fig. 18. Example for aggregates updates

Algorytm 7. Wstawianie do indeksu nowej wartości:

```
change(aggVal aAggVal, interval aRange)

last_time = wartość_minimalna
for element in lista_agregatów
{
   last_time = element.end
   newAgg = wylicz_agregat(aAggVal, element.aggVal)
   if(aRange.intersects(elem.range))
   {
         if(aRange.contains(element.range))
               element.aggVal = newAgg;
         else
         {
               if(element.start < aRange.start){
                     interwał.start = aRange.start
                     interwał.end= elem.end
                     element.end =  aRange.start
                     element.wstaw_za(interwał, newAgg)
               }else{
                     interwał.start = aRange.end
                     interwał.end = element.end
                     lastAggVal = element.aggVal

                     element.end =  aRange.end
                     element.aggVal = newAgg
                     element.wstaw_za(interwał, lastAggVal)
               }
         }
   }
if(element.end >= aRange.end)
   break;
}
if(last_time < aRange.end)
{
   If(last_time >= aRange.start)
         aRange.start = last_time
   lista_agregatów.wstawNaKoniec(aRange, aAggVal)
}
```

4. Przykładowe zapytanie

Przyjmijmy, że na danym odcinku drogi mamy kilka punktów pomiaru szybkości, chcemy wskazać sektory, na których jest duże prawdopodobieństwo spotkania kierowców przekraczających szybkość. Każde urządzenie monitorujące nadaje znacznik początku czasu życia pomiaru oraz znacznik końca, który odpowiada czasowi pobytu monitorowanego obiektu we wskazanym sektorze. W zaproponowanym zapytaniu wyliczamy średnią wartość szybkości samochodów dla kolejnych obszarów monitorowanych. Następnie łączymy wyliczone strumienie w pojedynczy, po czym odfiltrowujemy te sektory, których wartość średnia przekracza ustalony limit. Dodatkowo wyliczamy wartość równą połowie między najmniejszą a największą szybkością odnotowaną w danym sektorze. Wskaźnik ten rozszerza informację o zarys rozkładu szybkości dla monitorowanego sektora. Wynikowy strumień można użyć do podjęcia decyzji, gdzie należy umieścić patrole.

Strumienie I i II reprezentują pomiary szybkości samochodów w dwóch sektorach. Pomiar składa się z atrybutu val oraz interwału czasu, przez który jest aktywny.

W tabeli 6 zamieszczono przykładową zawartość strumieni dla poniżej zdefiniowanego zapytania realizującego przedstawione zadanie. Przyjęto, że średnia szybkość 70 kwalifikuje sektor do kontroli. Dla uproszczenia zapisu kolumna *result* informuje, które krotki ze strumienia *facts* zostaną przekazane do strumienia *result*. Należy zwrócić uwagę na opóźnienie obsługi krotek przez operator util.union, wynika to z potrzeby uporządkowania krotek wejściowych, do czego potrzebana jest co najmniej jedna krotka w każdym ze strumieni wejściowych.

Tabela 6

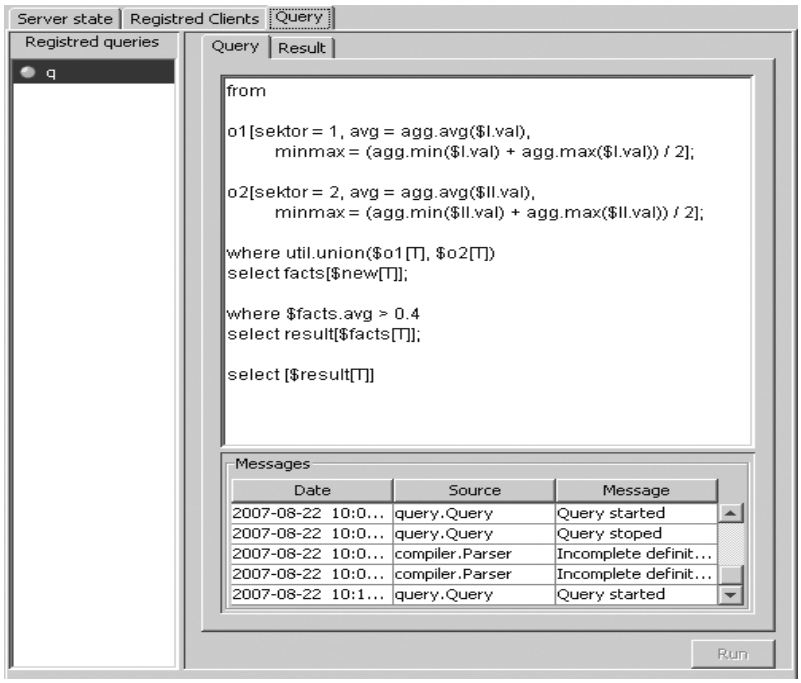
Przykładowa zawartość strumieni w trakcie wyliczania zapytania

I		o1				II		o2				facts				result
(interval)	val	(interval)	slot	avg	minmax	(interval)	val	(interval)	Slot	avg	minmax	(interval)	slot	avg	minmax	
<2,10)	40															
						<3,7)	90									
<4,10)	70	<2,4)	1	40	40											
						<5,9)	70	<3,5)	2	90	90	<2,4)	1	40	40	
						<7,14)	50	<5,7)	2	80	80					
<8,14)	70	<4,8)	1	55	55							<3,5)	2	90	90	<-
												<4,8)	1	55	55	
<9,17)	80	<8,9)	1	60	55							<5,7)	2	80	80	<-
						<9,18)	100	<7,9)	2	60	60	<7,9)	2	60	60	

Poniżej zamieszczono listing zapytania. Powyższe zadanie zostało uruchomione na platformie przetwarzania strumieniowego. Wartości pomiarów były generowane przez losowe źródło wartości z zakresu <0, 2.8).

```
from
o1[sektor = 1, avg = agg.avg($I.val),
          minmax = (agg.min($I.val) + agg.max($I.val)) / 2];
o2[sektor = 2, avg = agg.avg($II.val),
          minmax = (agg.min($II.val) + agg.max($II.val)) / 2];
```

```
where util.union($o1[T], $o2[T])
select facts[$new[T]];

where $facts.avg > 0.4
select result[$facts[T]];
select [$result[T]]
```



Rys. 19. Definicja dla zapytania o miejsca wykroczeń
Fig. 19. Definition of example query

Operator *util.union* łączy strumienie wejściowe w jeden strumień wyjściowy o nazwie *new*. Zapis *facts[$new[T]]* oznacza, że całe poddrzewo atrybutów dla strumienia *$new* jest kopiowane do strumienia *facts* .

Na rys. 20 przedstawiono uzyskane wyniki, a na rys. 19 postać formatki definiującej zapytanie. Jak widać na zamieszczonym rys. 20, oba z monitorowanych sektorów odnotowują wartości przekraczające limit. Praca systemu jest płynna, co wskazuje atrybut *sektor*, ponieważ wartości nadchodzą w zasadzie na przemian.

Wyobraźmy sobie sytuację, że chcemy wyliczyć średnią szybkość samochodów, które przebywały w danym sektorze w przeciągu ostatniej godziny. Wówczas należy zdefiniować okno zakresowe o rozmiarze 1h na strumieniach zasilających. W odniesieniu do poprzedniego zapytania zmianie ulegnie poniższy fragment.

```
…
where window.range($I, 1h)
select o1[sektor = 1, avg = agg.avg($I.val),
          minmax = (agg.min($I.val) + agg.max($I.val)) / 2];

where window.range($II, 1h)
select o2[sektor = 2, avg = agg.avg($II.val),
          minmax = (agg.min($II.val) + agg.max($II.val)) / 2];
…
```

Query Result

1.sektor	1.avg	1.minmax
1	2.0464729581130...	1.1439975553430...
2	2.8205983337129...	1.4164475228620...
1	1.6291050116539...	0.9353135821135...
1	1.7010914515620...	0.9713068020675...
2	2.8205983337129...	1.4164475228620...
1	1.9363737087535...	1.0889479306632...
2	2.8205983337129...	1.4164475228620...
1	1.9363737087535...	0.9955886488288...
2	2.8205983337129...	1.4164475228620...
1	1.9363737087535...	0.9955886488288...
2	2.8205983337129...	1.4164475228620...
1	1.9363737087535...	0.9955886488288...
2	2.8205983337129...	1.4164475228620...
1	1.9363737087535...	0.9955886488288...
2	2.8205983337129...	1.4164475228620...
1	1.9363737087535...	0.9955886488288...
2	2.8205983337129...	1.5986786914590...
2	2.8205983337129...	1.5986786914590...
1	1.9363737087535...	0.9955886488288...
2	2.8205983337129...	1.5057935549443...
1	1.9363737087535...	0.9955886488288...
2	2.8205983337129...	1.5057935549443...
1	2.1182879303018...	1.0865457596029...
2	2.8205983337129...	1.5057935549443...
1	2.5468575305205...	1.300830559712336

Rys. 20. Wyniki dla zapytania o miejsca wykroczeń
Fig. 20. Results for example query

Odnosząc się do przykładowych wyników z tab.6, zaproponowana modyfikacja poskutkuje następującymi wartościami dla strumienia o2. Przyjęto, że okno zakresowe ma rozmiar 10 jednostek. Zalecane jest, aby krotki z pomiarami źródłowymi, które nie definiują czasu życia, ustawiały tę wartość na 0, dopiero operator okna czasowego ustala czas wygasania krotek. Wówczas, jeżeli popełnimy błąd podczas definiowania zapytania, chronimy przed sytuacją przepełnienia kolekcji krotek.

Tabela 7

Przykładowa zawartość strumieni po użyciu okna zakresowego

II			o2			
(interval)	New(interval)	val	(interval)	Slot	avg	minmax
<3,3)	<3,13)	90				
<5,5)	<5,15)	70	<3,5)	2	90	90
<7,7)	<7,17)	50	<5,7)	2	80	80
<9,9)	<9,19)	100	<7,9)	2	70	70

5. Wnioski

W artykule zaprezentowano system przetwarzania strumieniowego StreamAPAS v5.0 oraz jego kompilator języka zapytań pod kątem realizacji operacji agregacji oraz sposobu zarządzania danymi hierarchicznymi.

Zastosowany w tym języku zapytań model danych poskutkował potrzebą rozbudowy analizatora leksykalnego o dodatkowy stos kontekstu drzewa atrybutów, z drugiej strony otrzymano strukturę, która w czytelny sposób ilustruje dane analityczne oraz przestrzenne.

Zaproponowany mechanizm kompilacji zapytań korzystających z funkcji agregacyjnych udostępnia możliwość generowania wyniku będącego drzewem atrybutów, co ułatwia utworzenie w systemie funkcji dedykowanych dla konkretnych dziedzin zastosowania tej platformy.

Zaproponowana algebra w [3] została rozszerzona o możliwość wyboru typu kolekcji krotek. Dodana nowa kolekcja czasowo-tabelaryczna upraszcza przenoszenie rekordów z tradycyjnych baz danych do systemu strumieniowego.

Wprowadzony wydajniejszy algorytm synchronizacji przetwarzania skrócił czas oczekiwania krotek na przetworzenie.

Dalszy rozwój tego systemu można podzielić na dwa kierunki. Pierwszy z nich obejmuje rozbudowę procesora strumieniowego, tak aby pozwalał na umieszczanie węzłów przetwarzających w systemie wielokomputerowym oraz wspierał większą liczbę operatorów. Drugi kierunek badań dotyczy rozwoju języka oraz architektury sieci przetwarzania. Ważny element, który należy rozwinąć w dalszych pracach, stanowi sposób definiowania źródeł danych, które dotąd są postrzegane jako proste strumienie. Rozwiązanie takie utrudnia uruchamianie zapytań, ponieważ ogranicza możliwości definiowania mechanizmów synchronizacji źródeł danych dla warunków początkowych, co zmusza użytkownika do implementacji tego mechanizmu oddzielnie.

6. Podsumowanie

Zaprezentowany został system przetwarzania strumieniowego wraz z językiem zapytań. Utworzony silnik przetwarzania obsługuje nową algebrę dla danych strumieniowych temporalnych [3], rozwiązanie takie upraszcza postać zapytań, ponieważ nie ma potrzeby definiowania operatorów IStream, DStream, RStream istniejących w języku CQL. Zastosowany silnik udostępnia operatory typu strumień-strumień, co gwarantuje wyższą wydajność pracy niż popularny model strumień-relacja-strumień, w którym konieczne są dodatkowe konwersje danych.

Zaproponowana składnia funkcji agregacyjnych oraz reprezentacja wyników tworzy jako całość wygodne narzędzie do umieszczania w języku funkcjonalności dedykowanych konkretnym zastosowaniom.

Przedstawiony mechanizm synchronizacji krotek w strumieniach pozwala na płynne ich przetwarzanie, nawet gdy częstość nadchodzących danych w jednym ze strumieni operatora jest niska, co stanowiło słabą stronę systemu, na którym się wzorowano [3].

BIBLIOGRAFIA

1. Gorawski M., Skierski A.: Okienkowy język zapytań. II Krajowa Konferencja Naukowa „Technologie Przetwarzania Danych", 24-26 września 2007, Poznań.
2. Balazinska M.: Fault-tolerance and load management in a Distributed Stream Processing System. PhD dissertation, Massachusetts institute of technology, February 2006.
3. Krämer J., Seeger B.: A Temporal Foundation for Continuous Queries over Data Streams. Department of Mathematics and Computer Science PhilippsUniversity, Marburg, Germany, 11th International Conference on Management of Data (COMAD 2005).
4. Arasu A., Babu S., Widom J.: The CQL Continuous Query Language Semantic Foundations and Query Execution. Stanford University, the International Journal on Very Large Databases (VLDB Journal) June 2006, 15:2, s. 121÷142.
5. Yu S., Atluri V., Adam N.: Optimizing View Materialization Cost in Spatial Data Warehouses. Lecture Notes in Computer Science 4081 Springer 2006, s. 45÷54.
6. Ardelli L., Ghelli G.: TQL: A Query Language for Semistructured Data Based on the Ambient logic. Mathematical structures in computer science 2004, Vol. 14; part 3, s. 285÷328.
7. Johannes, D. Leijen P.: The λ Abroad – A Functional Approach To Software Components. Daan Leijen. PhD thesis, Department of Computer Science, Utrecht University, November 2003.

Recenzent: Dr hab. inż. Zygmunt Mazur, prof Pol. Wrocławskiej

Wpłynęło do Redakcji 25 listopada 2008 r.

Abstract

The paper describes the general-purpose stream processing platform with the query language. The stream processor is build upon the temporal operator algebra over data streams [3], so that it is not necessary to define operators IStream, DStream, RStream which

exist in CQL. Besides the proposed algebra defines the stream-to-stream operators which are faster than popular in CQL stream-to-relation-to-stream operators.

The proposed language syntax in comparison with CQL define the way of further development system functionality by means of functions interface. This and hierarchical data structures simplify maintaining analytical applications where development is permanent process.

The presented stream processing system combines mainly the temporal logical operator algebra[3] and streams synchronization from [2]. In the result the final system has well defined available optimization rules and can offer low latency for tuples processing even when the query consist of slow data streams.

Adresy

Marcin GORAWSKI: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-100 Gliwice, Polska, Marcin.Gorawski@polsl.pl.
Aleksander CHRÓSZCZ: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-100 Gliwice, Polska, Aleksander.Chroszcz@polsl.pl.