RetractorDB

Edge Signal Processing Engine

Michal Widera

2026-06-04

Przetwarzanie sygnałów w bazach danych

RetractorDB

Instrukcja Uzytkownika

@Muro/MichalWidera

Spis treści

1 RetractorDB **9**
1.1 Pięć sąsiednich dziedzin 9
1.2 1. Teoria liczb: sekwencje Beatty'ego i układy pokrywające 10
1.3 2. Szeregowanie zadań przez sekwencje Beatty'ego 10
1.4 3. Cyfrowe przetwarzanie sygnałów: próbkowanie niejednorodne i banki filtrów 11
1.5 4. Strumieniowe systemy zarządzania danymi (DSMS) 11
1.6 5. Systemy szeregów czasowych i DSP wewnątrz bazy 12
1.7 Biała plama: gdzie leży wkład 12
1.8 Zastrzeżenie metodologiczne 13

2 Podstawy matematyczne **14**
2.1 Podstawy matematyczne 14

3 Algebra regularnych serii czasowych **17**

4 Formalne podstawy i dowody **20**
4.1 Układy pokrywające jako fundament 20
4.2 Narzędzia: własności podłogi i sufitu 21
4.3 Operatory w zapisie formalnym 22
4.4 Twierdzenie 1: przeplot zapewnia pokrycie zbiorów 22
4.5 Twierdzenie 2: rozplątanie spełnia postulaty Fraenkela 23
4.6 Własności operatorów wykorzystywane w optymalizacji 26
4.7 Dlaczego to ma znaczenie 28

5 Wyrażenia algebraiczne **29**

6 Implementacja modelu **30**

7 Reprezentacja graficzna **36**

8 Podsumowanie **38**

9 Konstrukcja języka zapytań **39**

10 Polecenie DECLARE **40**
10.1 Typy pól 40

11 Opcje odczytu w DECLARE **43**
11.1 ONESHOT . . . 43
11.2 DISPOSABLE . . . 43
11.3 HOLD . . . 43
11.4 Tabela porównawcza . . . 44

12 Polecenie SELECT **45**
12.1 Operatory klauzuli FROM . . . 46

13 Sekwencjonowanie operacji sumowania **47**

14 Sekwencjonowanie operacji przeplotu **50**

15 Klauzula VOLATILE **52**
15.1 Działanie . . . 52
15.2 Różnica względem `STORAGE MEMORY` . . . 52
15.3 Przykład . . . 53

16 Typy STORAGE **54**
16.1 Tabela typów . . . 54
16.2 Kiedy używać . . . 55
16.3 Przykład . . . 55

17 Operatory agregujące i funkcje wyrażeń **56**
17.1 Agregaty okna (MIN, MAX, AVG, SUMC) . . . 56
17.2 Funkcja to_string . . . 57

18 Polecenie RULE **59**
18.1 Składnia polecenia RULE . . . 60
18.2 Akcja DO SYSTEM . . . 61
18.3 Akcja DO DUMP . . . 61
18.4 Wiele reguł dla jednego strumienia . . . 62

19 Konstrukcja mechanizmu **63**

20 Warunek logiczny w RULE **65**
20.1 Operatory porównania . . . 65
20.2 Spójniki logiczne . . . 65
20.3 Struktura wyrażenia . . . 65
20.4 Przykłady . . . 66
20.5 Dostęp do pól . . . 66

21 Przykład alarmowania **67**
21.1 Przykład 2: zapis kontekstu zdarzenia (DO DUMP) . . . 67
21.2 Przykład 3: rotacja zrzutów (DO DUMP z RETENTION) . . . 69
21.3 Przykład 4: wiele reguł na jednym strumieniu . . . 69

22 Dyrektywy konfiguracyjne **71**

23 Architektura systemu **73**

23.1 Przegląd poruszonych w rozdziale tematów 73

24 Perspektywa ogólna **76**

25 Przepływ danych i sterowania **78**
25.1 Zatrzymanie xretractor 79

26 Artefakty, Substraty, Efemerydy **82**

27 Format zapisu danych **84**
27.1 Zestaw plików artefaktu i substratu 85
27.2 Rozdziały 85

28 Pliki **88**
28.1 Plik deskryptora (.desc) 88
28.2 Plik danych binarnych 90
28.3 Plik metadanych (.meta) 91
28.4 Plik cienia (.shadow) 99
28.5 Plik cienia indeksu (.meta.shadow) 100
28.6 Relacja pomiędzy plikami 105
28.7 Punkt wyjścia — plik binarny bez metadanych 106
28.8 Co wnosi każdy plik 106

29 Mechanizm rotacji plików **109**
29.1 Domyślne zachowanie (bez dyrektywy `ROTATION`) 109
29.2 Dyrektywa `ROTATION` i licznik sesji 109
29.3 Przepływ sterowania w procesie rotacji 110
29.4 Co trafia do plików `.old<N>` 110
29.5 Przykład sekwencji trzech sesji 111
29.6 Otwieranie pliku rotowanego w `xtrdb` 111

30 Narzędzie inspekcji: `xtrdb -s` **112**
30.1 Cel i zastosowanie 112
30.2 Co pokazuje mapa 112
30.3 Przykład 1 — artefakt prosty 114
30.4 Przykład 2 — artefakt z przerwą w transmisji i modyfikacją 114
30.5 Przykład 3 — artefakt z retencją segmentową 115

31 Podsumowanie: uzasadnienie przyjętej struktury **117**
31.1 Zestaw plików i typy akcesorów 117
31.2 Pliki artefaktu 117
31.3 Mechanizm rotacji 118
31.4 Narzędzie inspekcji `xtrdb -s` 118
31.5 Porównanie podejść 118

32 Kompilacja i budowa planu **120**
32.1 Dostępne flagi xretractor 122

33 Przetwarzanie i dystrybucja danych **124**

34 Analiza artefaktów **129**
34.1 Inspekcja metadanych null/gap 131

35 Podsumowanie **133**

36 Kompilacja zapytań **134**
36.1 Wejście i wyjście kompilatora 134
36.2 Przegląd poruszonych w rozdziale tematów 135

37 Przebiegi kompilacji **137**
37.1 Przykład śledzący 137
37.2 Łańcuch etapów 138

38 Budowa drzewa zależności **140**

39 Substraty **143**
39.1 Redukcja substratów 145
39.2 Eliminacja duplikatów substratów 147
39.3 Wchłonięcie substratu przez jawny strumień 149
39.4 Aktualizacja schematu po wchłonięciu 151

40 Rozwijanie symbolu * **153**

41 Rozwiązywanie interwałów **155**
41.1 Algorytm 155
41.2 Równania operatorów 156
41.3 Dlaczego iteracja? 157

42 Wykrywanie pętli w kompilacji **158**
42.1 Przykład pętli 158
42.2 Efekt kompilacji 158
42.3 Mechanizm wykrywania 159
42.4 Jak naprawić 159

43 Aliasowanie **161**

44 Przetwarzanie symbolu _ **163**

45 Równanie typów w górę **165**

46 Debugowanie kompilacji **167**
46.1 Podstawowe narzędzie: flaga `-c` 167
46.2 Jak czytać plan kompilacji 167
46.3 Wizualizacja grafu zależności 169
46.4 Weryfikacja interwałów 169
46.5 Typowe błędy kompilacji 170

47 Realizacja zapytań **171**

48 Algorytm przeglądu drzewa zapytań **173**

48.1 Przegląd ogólny 173
48.2 Struktura danych: qTree 173
48.3 Minimalna siatka czasowa: TimeLine / CRSMath 173
48.4 Krok zerowy: `processZeroStep()` 176
48.5 Główna pętla: filtrowanie i przetwarzanie 177
48.6 Rozgłaszanie wyników: `broadcast()` 177
48.7 Pełny przykład: zapytania A, B, C, D dla delt {1/2, 1/3} 177

49 Zapytania Ad hoc **181**

50 Realizacja alarmowania **185**
50.1 Miejsce RULE w cyklu przetwarzania 185
50.2 Ewaluacja warunku WHEN 185
50.3 Akcja DO SYSTEM 186
50.4 Akcja DO DUMP — szczegółowy algorytm 186
50.5 Retencja (RETENTION N) 188
50.6 Format pliku zrzutu 190
50.7 Wiele reguł — kolejność ewaluacji 190
50.8 Ograniczenia i uwagi praktyczne 191

51 Ruchome okno danych AGSE **192**
51.1 Jak zmienia się interwał strumienia wyjściowego 192
51.2 Typowe wzorce użycia 193
51.3 Wizualizacja operatora @ 193
51.4 Przykłady 193

52 Przykład serializacji **195**

53 Przykład średniej ruchomej **199**
53.1 Dane źródłowe 199
53.2 Zapytanie RQL 199
53.3 Uruchomienie 200
53.4 Weryfikacja planu zapytania 200
53.5 Zależność między parametrami okna a opóźnieniem 200

54 Różne typy okien **202**
54.1 Strumień źródłowy 202
54.2 Tumbling window — okna bez nakładania 202
54.3 Sliding window — okna z nakładaniem 203
54.4 Próbkowanie — okna z przerwami 203
54.5 Okno lustrzane — odwrócona kolejność pól 203
54.6 Zestawienie wzorców 204
54.7 Plan realizacji zapytań 204

55 Odtwarzanie strumienia **205**

56 Przykłady zastosowań **207**

57 Implementacja filtru sygnałowego **208**

58 Wizualizacja EKG i Detekcja Arytmii — baza MIT-BIH **215**
58.1 Źródło danych — PhysioNet MIT-BIH Arrhythmia Database 215
58.2 Przygotowanie danych 216
58.3 Zapytanie RQL 217
58.4 Wizualizacja na ekranie 217
58.5 Detekcja QRS i identyfikacja arytmii 219

59 Załączniki **226**

60 Opcje wywołania **227**

61 xretractor **228**
61.1 Tryby pracy 228
61.2 Tryb przetwarzania (domyślny) 228
61.3 Tryb tylko kompilacja (`-c`) 230
61.4 Informacje o wersji 231

62 xqry **233**
62.1 Uruchomienie 233
62.2 Odbiór danych ze strumieni 234
62.3 Diagnostyka i sterowanie serwerem 234
62.4 Formaty wyjścia 235
62.5 Sterowanie trybem odbioru 235
62.6 Wzorzec uruchamiania w skryptach 235
62.7 Informacje o wersji 236

63 xtrdb **237**
63.1 Uruchomienie 237
63.2 Przegląd poleceń 238
63.3 Zarządzanie sesją 238
63.4 Konfiguracja środowiska 239
63.5 Otwieranie artefaktu 239
63.6 Odczyt i zapis rekordów 239
63.7 Przeglądanie zawartości 239
63.8 Edycja payload 240
63.9 Metadane null (`.meta`) 240
63.10 Pozostałe polecenia 241
63.11 Przykłady użycia 241

64 Geneza systemu **243**

65 Dlaczego wybrano taką nazwę dla systemu? **245**

66 Dalsze kierunki rozwoju **246**

67 Jeszcze inna matematyka **247**

68 Kolorowanie składni RQL **249**
68.1 Visual Studio Code 249
68.2 Vim 250

68.3 bat / batcat 251

69 Testy integracyjne **254**
69.1 Testy sekwencyjne — `IntegrationTest_serial` 254
69.2 Testy równoległe — `IntegrationTest_parallel` 256

70 Literatura **259**

Rozdział 1

RetractorDB

Ten rozdział jest mapą, nie katalogiem. Zamiast wyliczać wszystko, co kiedykolwiek napisano o strumieniach i sygnałach, pokazuję pięć nurtów recenzowanej literatury, na styku których leży RetractorDB, i dla każdego z nich odpowiadam na trzy pytania: co ten nurt już rozwiązał, w czym RetractorDB się od niego różni i czego ten nurt **nie** dotyka. Dopiero nałożenie tych pięciu warstw na siebie pokazuje lukę, którą ten projekt wypełnia.

> **Pobierz dokumentację jako PDF**
>
> retractordb.pdf — generowany automatycznie przy każdym `git push`.

Uwaga

Ten system to: Edge Signal Processing Engine (Brzegowy System Przetwarzania Sygnałów)

Info

Dlaczego umieściłem ten rozdział tak wcześnie? Bo uczciwa odpowiedź na pytanie „czy to jest potrzebne?” wymaga najpierw pokazania, co już istnieje. Większość pomysłów w informatyce została już raz pomyślana – wymyślanie koła na nowo to marnowanie cudzego wysiłku. Ten rozdział jest moją próbą udowodnienia, że akurat tego koła jeszcze nie wynaleziono.

1.1 Pięć sąsiednich dziedzin

Problem, który rozwiązuje RetractorDB, nie należy w całości do żadnej pojedynczej dyscypliny. Siedzi w szczelinie między pięcioma:

1. **Teoria liczb** – sekwencje Beatty’ego, twierdzenie Fraenkela, układy pokrywające. To dostarcza fundamentu formalnego.
2. **Szeregowanie zadań przez sekwencje Beatty’ego** – ta sama matematyka, inne zastosowanie. Najbliższy sąsiad aplikacyjny.

3. **Cyfrowe przetwarzanie sygnałów (DSP)** – próbkowanie niejednorodne i banki filtrów o wymiernych współczynnikach. To DSP-owy odpowiednik operacji przeplotu.
4. **Strumieniowe systemy zarządzania danymi (DSMS)** – algebry strumieni i semantyka zapytań ciągłych. To bazodanowy punkt odniesienia.
5. **Systemy szeregów czasowych (TSMS) i DSP wewnątrz bazy** – najwęższa, najsłabiej zaludniona nisza, najbliższa właściwemu celowi systemu.

Omawiam je kolejno, od fundamentu ku zastosowaniu.

1.2 1. Teoria liczb: sekwencje Beatty'ego i układy pokrywające

Cała algebra RetractorDB stoi na sekwencji Beatty'ego i jej uogólnieniu przez Fraenkela na liczby wymierne. Te wyniki przytaczam w Formalnych podstawach i dowodach. Tutaj interesuje mnie szersze tło: jak ta matematyka funkcjonuje we współczesnej literaturze i czy ktoś zastosował ją już tam, gdzie ja.

Sekwencje Beatty'ego mają bogatą literaturę kombinatoryczną oraz udokumentowane zastosowania w nieperiodycznych parkietażach (kwazikryształy), szeregowaniu okresowym, widzeniu komputerowym (linie cyfrowe) i teorii języków formalnych [11]. Nurt jest żywy: Schaeffer, Shallit i Zorcic (2024) wykazali, że niejednorodna sekwencja Beatty'ego jest synchronizowalna automatem skończonym, co prowadzi do rozstrzygalności teorii pierwszego rzędu tych sekwencji [12]. Dla mnie najistotniejsza jest jednak praca Bergera, Felzenbauma i Fraenkela (1986) o rozłącznych układach pokrywających opartych na **wymiernych** sekwencjach Beatty'ego [13] – to dokładnie ten wariant, na którym opieram rozplątanie, a którego w pierwotnej pracy nie przywołałem.

Czego ten nurt nie dotyka: teoria liczb bada te sekwencje jako obiekty matematyczne. Nie łączy ich z bazą danych, z modelem przetwarzania strumieni ani z przetwarzaniem sygnałów. Dostarcza cegieł, nie budowli.

1.3 2. Szeregowanie zadań przez sekwencje Beatty'ego

To jest nurt, który muszę omówić najuczciwiej, bo używa **tej samej maszynerii dowodowej** co moje twierdzenia – tyle że w innym celu. W problemie szeregowania okresowego (ang. *pinwheel scheduling*) zadania o różnych okresach powtarzania rozdziela się tak, że zadania o jednym czasie powtórzeń trafiają w sloty czasowe należące do pierwszej komplementarnej sekwencji Beatty'ego, a o drugim – do drugiej [14]. Świeże prace (2026) prowadzą dowody na podziale Rayleigha/Beatty'ego z tożsamościami na funkcjach podłogi i sufitu typu $\lceil (m+l)a \rceil - \lceil ma \rceil$ [15] – niemal kropka w kropkę aparat z mojego dowodu, że rozplątanie spełnia postulaty Fraenkela.

Wniosek jest dla mnie podwójny. Z jednej strony – to niezależne potwierdzenie, że podejście jest poprawne i naturalne; skoro ktoś dochodzi tą samą drogą do działającego szeregowania, fundament jest solidny. Z drugiej – to zawęża to, co mogę nazwać nowością. „Sekwencje Beatty'ego do szeregowania" już istnieją i są aktywnie publikowane.

Co ciekawe, mój system używa tej matematyki **wewnętrznie** właśnie do szeregowania zadań (patrz Realizacja zapytań) – ale to nie tu leży wkład oryginalny.

Czego ten nurt nie dotyka: szeregowanie traktuje sekwencje jako narzędzie przydziału slotów czasowych procesorom. Nie buduje na nich algebry danych, nie wyraża nimi operacji na sygnałach, nie tworzy języka zapytań.

1.4 3. Cyfrowe przetwarzanie sygnałów: próbkowanie niejednorodne i banki filtrów

Operacja przeplotu i rozplątania to – w języku DSP – konwersja częstotliwości próbkowania między strumieniami o różnych Δ. Tu istnieje rozległa, dojrzała literatura. Najbliższym pomostem jest praca Samadiego, Ahmada i Swamy'ego (2004), która formułuje warunek perfekcyjnej rekonstrukcji niejednorodnych banków filtrów na podstawie odpowiedzi układu na opóźnione sygnały skoku jednostkowego [16] – wprowadza więc maszynerię funkcji skoku (a pośrednio podłogi) do dziedziny wielotempowego DSP. Szerszy nurt to próbkowanie okresowo-niejednorodne sygnałów pasmowo ograniczonych [17] oraz – bezpośrednio adekwatne – banki filtrów o **wymiernych** współczynnikach decymacji (Kovačević i Vetterli) [18].

Pojawiają się tam nawet konstrukcje teorioliczbowe: banki filtrów Ramanujana wydobywają składowe okresowe sygnału [19]. Ale akurat sekwencji Beatty'ego ani twierdzenia Fraenkela w tej literaturze nie znalazłem – i to jest część luki.

Czego ten nurt nie dotyka: DSP operuje w dziedzinie z, dziedzinie częstotliwości, na ramkach i bazach. Nie ujmuje resamplingu jako deklaratywnego operatora algebraicznego ani nie osadza go w systemie bazodanowym. Współczynniki bywają wymierne, ale aparatem jest analiza, nie teoria liczb podziału zbioru.

1.5 4. Strumieniowe systemy zarządzania danymi (DSMS)

Po stronie bazodanowej kanonem jest CQL ze stanfordzkiego projektu STREAM (Arasu, Babu, Widom). W tym modelu strumień to potencjalnie nieskończony wielozbiór elementów ⟨s, τ⟩, gdzie s jest krotką, a τ stemplem czasowym [20]; semantykę zapytań buduje się na oknach i odwzorowaniach strumień↔relacja. Drugim bliskim sąsiadem jest temporalna algebra Krämera i Seegera (system PIPES), zapewniająca deterministyczne wyniki zapytań ciągłych oraz bogaty zbiór reguł transformacji stanowiących podstawę optymalizacji [21].

To jest właściwy punkt odniesienia dla mojej algebry i moich reguł przepisywania wyrażeń. Różnica jest jednak fundamentalna i dotyczy samego modelu danych. CQL i PIPES budują semantykę na modelu (s, τ) – każda krotka nosi własny stempel czasowy, a operatory działają przez okna. Ja przyjmuję model różnicowy (s_n, Δ) z wymierną, stałą wartością Δ na strumień, a operatory wyrównujące strumienie o różnych Δ wyprowadzam z teorii liczb. To nie jest kosmetyczna różnica w składni – to inny model danych, prowadzący do innej klasy operatorów (przeplot, rozplątanie) i innej metody optymalizacji.

Czego ten nurt nie dotyka: DSMS celują w przybliżone, skalowalne przetwarzanie nieograniczonych strumieni z tolerancją na nieuporządkowanie czasowe. Nie dążą do dokładnych, deterministycznych operacji DSP w rygorze twardego czasu rzeczywistego i nie sięgają po teorię liczb dla semantyki resamplingu.

1.6 5. Systemy szeregów czasowych i DSP wewnątrz bazy

To najwęższa nisza – i najbliższa właściwemu celowi RetractorDB. Kanoniczny przegląd to praca Jensena, Pedersena i Thomsena „Time Series Management Systems: A Survey" (IEEE TKDE, 2017) [22]. Opisany tam system Plato jest najbliższym prawdziwym „DSP wewnątrz bazy": łączy RDBMS z metodami przetwarzania sygnałów, eliminując potrzebę eksportu danych do narzędzi zewnętrznych typu R czy SPSS [22]. Pozostałe podejścia do „sygnałów w bazie" sprowadzają się do aproksymacji i kompresji – reprezentacje falkowe, słownikowe, kształtowe.

Wszystkie one traktują jednak DSP jako aproksymację albo analitykę po fakcie. Żaden nie czyni z operacji przetwarzania sygnałów **dokładnych, deterministycznych operatorów pierwszej klasy** wewnątrz algebry zapytań. To potwierdza, że nisza jest cienka, a mój kąt natarcia – dokładność na liczbach wymiernych – jest odrębny.

Czego ten nurt nie dotyka: TSMS optymalizują skalę ingestii, kompresję i retencję. DSP jest w nich obywatelem drugiej kategorii – dodatkiem analitycznym, nie rdzeniem semantyki.

1.7 Biała plama: gdzie leży wkład

Po nałożeniu pięciu warstw obraz staje się czytelny. Każda dziedzina dotyka jednej lub dwóch ścian problemu, ale **żadna nie zajmuje ich przecięcia**:

Dziedzina	Beatty/Fraenkel	Dokładny DSP	Algebra strumieni / język zapytań	Twardy czas rzeczywisty
Teoria liczb	✓	–	–	–
Szeregowanie (pinwheel)	✓	–	–	częściowo
DSP wielotempowy	–	✓	–	–
DSMS (CQL, PIPES)	–	–	✓	–
TSMS / DSP-w-bazie	–	częściowo	częściowo	–
RetractorDB	✓	✓	✓	✓

Wkład RetractorDB nie leży w żadnym pojedynczym składniku – leży w ich **syntezie**: w użyciu układów pokrywających (wymiernych sekwencji Beatty'ego i twierdzenia Fraenkela) jako semantycznego fundamentu deklaratywnej algebry strumieni, która realizuje dokładne operatory przetwarzania sygnałów wewnątrz systemu bazodanowego,

w rygorze twardego czasu rzeczywistego. Teoria liczb ma Beatty'ego i nawet szeregowanie, ale nie łączy ich z bazą ani z DSP. DSP ma multirate i wymierne banki filtrów, ale nie sięga po Fraenkela i nie ujmuje tego jako języka zapytań. DSMS ma algebry strumieni i reguły optymalizacji, ale na modelu okienkowym (s, τ), nie różnicowym (s_n, Δ). To przecięcie jest puste.

Ostrzeżenie

Stąd realne ryzyko, które wprost wskazuję: społeczność szeregowania publikuje tę samą maszynerię Beatty'ego/Fraenkela w latach 2023–2026. Pomost „układy pokrywające ↔ wyrównanie strumieni i DSP" postawiłem publikacją już w 2006 roku [3], lecz w miejscu o niskiej odnajdywalności. Jeśli ten wynik nie trafi do dobrze cytowanego obiegu, ten sam pomost może zostać niezależnie postawiony i przypisany komu innemu.

1.8 Zastrzeżenie metodologiczne

To przegląd ukierunkowany, nie systematyczny – oparty na wyszukiwaniu w pięciu nurtach, nie na pełnej analizie cytowań. Do pełnej, recenzowanej publikacji wymaga domknięcia o przegląd cytowań „w przód" prac Samadiego [16] i nurtu szeregowania [14], a także o weryfikację, czy ktokolwiek użył wprost twierdzenia Fraenkela w kontekście wielotempowego DSP. Z mojego przeszukania – nie znalazłem takiej pracy. Jeśli istnieje, zmienia to zakres roszczenia o nowość i należy ją tu uwzględnić.

Rozdział 2

Podstawy matematyczne

2.1 Podstawy matematyczne

Info

Czy wiesz co to jest medal Fieldsa? Jest to nagroda przyznawana wyłącznie wybitnym matematykom w wieku poniżej 40 lat. Nazywana jest matematycznym Noblem. Co ciekawe żaden matematyk nie otrzyma nagrody Nobla – zgodnie z życzeniem fundatora. Sam John Charles Fields (1863-1932) był Kanadyjskim matematykiem. John Charles Fields miał jednego doktoranta – Samuela Beatty (1881-1970).

Samuel Beatty w 1926 roku opublikował następujące twierdzenie [1]:

Jeśli p, q są dodatnimi liczbami niewymiernymi i zachodzi pomiędzy nimi zależność

$$\frac{1}{p} + \frac{1}{q} = 1$$

to sekwencje

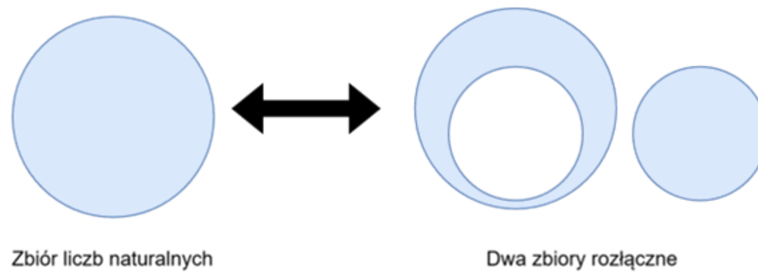
$$\{\lfloor np \rfloor\}_{n=1}^{\infty} = \lfloor p \rfloor, \lfloor 2p \rfloor, \lfloor 3p \rfloor, ...$$

oraz

$$\{\lfloor nq \rfloor\}_{n=1}^{\infty} = \lfloor q \rfloor, \lfloor 2q \rfloor, \lfloor 3q \rfloor, ...$$

oraz dokonują podziału zbioru dodatnich liczb całkowitych.

Te dwie sekwencje dokonują podziału zbioru liczb naturalnych. Oznacza to że dysponując dwoma liczbami niewymiernymi, pomiędzy którymi wskazana w twierdzeniu zależność – będziemy mogli podzielić zbiór wszystkich liczb naturalnych na dwa rozłączne zbiory (Rys. 1).



Rys. 1. Repreznatacja graficzna pojęcia zbiorów rozłącznych

Twierdzenie Beaty samo w sobie jest bardzo ciekawą obserwacją – jednak w przypadku systemów komputerowych mamy pewien problem z liczbami niewymiernymi. Liczby rzeczywiste – pomimo faktu że w niektórych językach programowania pojawia się czasem słowo Real lub Float jako reprezentanta typu liczby rzeczywistej, z liczbami rzeczywistymi nie mają wiele wspólnego. Fundamentalny problem polega na tym że ich nie mamy i zapewne nigdy mieć nie będziemy.

I tu nasza podróż gwałtownie by się skończyła gdyby nie powstało kolejne twierdzenie. Sytuacja diametralnie uległa zmianie za sprawą matematyka – Aviezri Siegmund Fraenkel (1926) specjalizującego się w kombinatorycznych aspektach teorii gier.

Przedstawił on w 1969 roku następujące twierdzenie [2]:

Sekwencje

$$\mathcal{B}(\alpha, \alpha') := \left(\left\lfloor \frac{n - \alpha'}{\alpha} \right\rfloor\right)_{n=1}^{\infty}$$

oraz

$$\mathcal{B}^{(c)}(\alpha, \alpha') := \left(\left\lceil \frac{n - \alpha'}{\alpha} \right\rceil\right)_{n=1}^{\infty}$$

dokonują podziału zbioru $\mathbb{N}$ wtedy i tylko wtedy gdy następujące pięć warunków zostanie spełnionych:

1.

$$0 < \alpha < 1$$

2.

$$\alpha + \beta = 1$$

3.

$$0 \leq \alpha + \alpha' \leq 1$$

4. Jeśli α jest liczbą niewymierną, wtedy:

$$\alpha' + \beta' = 0$$

i

$$k\alpha + \alpha' \notin \mathbb{Z}$$

dla

$$2 \leq k \in \mathbb{N}$$

5. Jeśli α jest liczbą wymierną, (niech q∈N będzie najmniejszą liczbą taką że qα∈N), wtedy

$$\frac{1}{q} \leq \alpha + \alpha'$$

i

$$\lceil q\alpha' \rceil + \lceil q\beta' \rceil = 1$$

No i to jest to czego potrzebujemy! Liczb niewymiernych co prawda nie mamy, ale liczby wymierne rozumiane jako stosunek dwóch liczb naturalnych to jest temat do ogarnięcia za pomocą komputera.

W naszym przypadku najpierw stworzyłem prototypy równań w języku Python a następnie zacząłem poszukiwać podstaw matematycznych, które wyglądały podobnie i można było się oprzeć na nich jako dobrze udokumentowanych równaniach popartych formalnymi dowodami. Dowodami oczywiście przeprowadzonymi przez bardziej doświadczonych matematyków. Skromne umiejętności pozwoliły jednak na identyfikację tych dwóch publikacji w aspekcie moich pomysłów.

W tym dokumencie nie umieściłem formalnych dowodów. Dlatego przestawiłem tutaj jedynie stosowane w systemie równania i twierdzenia. Po formalne dowody odsyłam do moich publikacji naukowych [3].

Rozdział 3

Algebra regularnych serii czasowych

Algebra – rozumiana jako konstrukcja w postaci zdefiniowanego zbioru i zdefiniowanych operacji na nim, stanowi podstawę dla opracowanego deklaratywnego język zapytań. W dalszej części pracy odnosząc się do Algebry (bez dodatkowego przymiotnika) będę ją rozumiał jako Algebrę regularnych serii czasowych. Jeśli będę chciał odwołać się do Algebry Relacji – jasno wyspecyfikuję przymiotnik.

Zaproponowałem [3] następującą definicje regularnej serii czasowej (tzw. Modelu danych) oraz następujące operacje i definicje.

Uwaga

Przez strumień danych rozumiemy uporządkowaną parę S := (sn,Δ) – gdzie pierwszy element to uporządkowania seria danych a drugi, oznaczony symbolem delty to regularny odstęp czasu pomiędzy kolejnymi elementami serii danych.

Tak zdefiniowaną serię danych w systemie określam jako strumień danych. Taki regularnie przepływający przez system zestaw danych, zazwyczaj opisany schematem danych zawiera pola różnych typów. Każdy odczyt występuje w równym odstępie czasu pomiędzy kolejnymi pomiarami. Taka konstrukcja bardziej przypomina sygnał cyfrowy niż nieregularny strumień danych – jednak oznaczenie jej jako strumień w dalszej części prac badawczych okaże się uzasadnione.

Info

Uwaga:
Pojęcie strumień i Seria czasowa w tej pracy używane są zamiennie i oznaczają to samo.
Formalnie w literaturze naukowej strumień oznaczany jest jako zbiór par (a,t) – gdzie a oznacza krotkę, a czast oznacza jej moment zarejestrowania lub wystąpienia.
W strumieniu dopuszczalne są krotki, których czas t pokrywa się dla różnych krotek. W przypadku serii czasowej rozróżniamy dwa typy serii – regularne i

nieregularne.
- W przypadku serii nieregularnych – seria to sekwencja uporządkowanych krotek w czasie – {at,tn}, gdzie czas tn jest unikalny w zbiorze dla każdej krotki.
- Natomiast seria regularnej serii czasowej może zostać opisana sekwencją krotek i regularnym odstępem czasu pomiędzy ich występowaniem – ({at},D) – i to ta ostatnia definicja jest bazą dalszych operacji w opracowanym systemie.

Operacje jakie możemy na takim zbiorze danych wykonać zdefiniowałem następująco:

- przeplot i rozplątanie
- suma i różnica
- przesunięcie sekwencji
- agregacja i serializacja

W operacji przeplotu biorą udział dwa różne strumienie danych.

Definiujemy ją następująco:

$$c_n = \begin{cases} b_{n-\lfloor nz \rfloor} & \lfloor nz \rfloor = \lfloor (n+1)\, z \rfloor \\ a_{\lfloor nz \rfloor} & \lfloor nz \rfloor \neq \lfloor (n+1)\, z \rfloor \end{cases}, z = \frac{\Delta_b}{\Delta_a + \Delta_b}, \Delta_c = \frac{\Delta_a \Delta_b}{\Delta_a + \Delta_b}$$

Argumentem operacji splątania (przeplotu) są dwa strumienie danych A i B, każdy z własną szybkością napływu danych. Wynikiem jest strumień wynikowy C – z nową różną od dwóch poprzednich szybkością napływu wyznaczoną wzorem powyżej.

Operację będziemy oznaczać symbolem #.

Operację rozplątania definiujemy poprzez dwie operacje.

1. Rozplątanie lewostronne jako strumień A w postaci:

$$a_n = c_{n+\left\lceil \frac{(n+1)\Delta_a}{\Delta_b} \right\rceil},\ \Delta_a = \frac{\Delta_c \Delta_b}{|\Delta_c - \Delta_b|}$$

2. Rozplątanie prawostronne jako strumień B w postaci:

$$b_n = c_{n+\left\lfloor \frac{n\Delta_b}{\Delta_a} \right\rfloor},\ \Delta_b = \frac{\Delta_c \Delta_a}{|\Delta_c - \Delta_a|}$$

Operacje rozplątania 1 i 2 będziemy oznaczać symbolami & i %.

Argumentem operacji rozplątania jest splątany strumień danych oraz wymierna liczba określająca szybkość napływu odplątywanego strumienia danych. W wyniku operacji otrzymujemy strumień danych z wyznaczoną szybkością wzorem powyżej.

Operacje splątania i rozplątania są komplementarne. Oznacza to że przypominają operacje mnożenia i dzielenia w zbiorze liczb naturalnych. W wyniku mnożenia otrzymujemy pewien wynik natomiast w wyniku dzielenia – czasem dochodzi reszta, istotne jest również to co przez co dzielimy i w jakiej kolejności.

Operacje sumy zdefiniowałem następująco:

$$c_n = \begin{cases} a_n | b_{\lfloor \frac{n\Delta_a}{\Delta_b} \rfloor} & \Delta_c = \Delta_a \\ a_{\lfloor \frac{n\Delta_b}{\Delta_a} \rfloor} | b_n & \Delta_c = \Delta_b \end{cases}, \Delta_c = \min\left(\Delta_a, \Delta_b\right)$$

Natomiast różnicę opisuje wzór:

$$a_n = \begin{cases} c_n & \Delta_b \geqslant \Delta_a \\ c_{\lceil \frac{n\Delta_a}{\Delta_b} \rceil} & \Delta_b < \Delta_a \end{cases}$$

Te operacje oznaczać będziemy znakami + oraz -.

Operacja przesunięcia sekwencji zawiera argument w postaci opóźnienia dostępu do danych o daną ilość odstępów czasu pomiędzy kolejnymi elementami. I tak np. dane napływające co sekundę ze strumienia źródłowego po wykonaniu operacji przesunięcia o 3 – pojawią się jako wynik opóźnione o 3 sekundy.

Operację przesunięcia oznaczać będę za pomocą >.

Ostatnią operacją w ramach zdefiniowanej algebry jest operacja agregacji i serializacji – w skrócie Agse. O ile wydaje się że to dwie oddzielne operacje, zdefiniowałem dwuargumentowy operator implementujący logikę ruchomego okna danych. Pierwszym argumentem jest skok okna, drugim jest jego szerokość. Skok jest liczbą naturalną o ile ruchome okno danych należy przesunąć nad strumieniem. Zakładamy że źródłowy strumień danych rozbity zostaje względem schematu danych, modyfikując jego szybkość napływu. Szerokość okna jest liczbą całkowitą, różną od zera. Wartości ujemne szerokości przenoszą kolejność tworzonych elementów w odbiciu lustrzanym. Wartości dodatnie – zachowują sekwencyjny charakter tworzonych ruchomych okien danych.

Operację Agse oznaczać będę znakiem @.

Podsumowując, algebra będąca podstawą dla deklaratywnego języka zapytań prezentuje się następująco:

$$A_{rql} ::= ((s_n, \Delta_s), (\#, \&, \%, +, -, >, @))$$

Gdzie pierwszy element pary definiującej algebrę to model danych (s_n — seria danych, Δ_s — jej regularny odstęp czasu) a drugi to zdefiniowane formalnie na tym modelu danych operacje.

Rozdział 4

Formalne podstawy i dowody

W rozdziale o algebrze regularnych serii czasowych przedstawiłem zbiór operatorów i opisujące je równania. Świadomie pominąłem tam formalne dowody – chciałem najpierw pokazać *co* system robi, zanim wyjaśnię *dlaczego* wolno mu to robić. Ta strona uzupełnia tę lukę. Zebrałem tu formalny szkielet algebry: powiązanie operatorów strumieniowych z teorią układów pokrywających oraz dowody twierdzeń, na których opiera się poprawność i optymalizacja planów zapytań.

Info

Cała poniższa konstrukcja trzyma się w jednej dziedzinie – liczb wymiernych. To nie jest ozdobnik. To jest cały sens. Twierdzenie Beatty potrzebuje liczb niewymiernych, których w komputerze nie ma. Twierdzenie Fraenkela pozwala zejść do liczb wymiernych. Dowody na tej stronie pokazują, że operacje przeplotu i rozplątania są szczególnym przypadkiem sekwencji Beatty spełniającym postulaty Fraenkela – a więc są realizowalne wyłącznie na liczbach wymiernych.

4.1 Układy pokrywające jako fundament

Literatura dotycząca układów pokrywających (ang. *Covering Systems*) [4] związana jest z kombinatoryką i kryptoanalizą w obszarze teorii liczb. Rozważanym problemem jest sposób wyznaczania podziału zbioru dodatnich liczb naturalnych. Mówimy, że dwie sekwencje dokonują podziału zbioru dodatnich liczb naturalnych, jeśli zbiory powstałe z elementów tych sekwencji po operacji przecięcia tworzą zbiór pusty, a ich suma tworzy zbiór dodatnich liczb naturalnych.

Podstawą rozważań jest sekwencja nazywana sekwencją Beatty. W postaci ogólnej zapisujemy ją w wariancie z funkcją podłogi:

$$\mathcal{B}(\alpha, \alpha') := \left(\left\lfloor \frac{n - \alpha'}{\alpha} \right\rfloor \right)_{n=1}^{\infty}$$

lub w wariancie z funkcją sufitu:

$$\mathcal{B}^{(c)}(\alpha, \alpha') := \left(\left\lceil \frac{n - \alpha'}{\alpha} \right\rceil \right)_{n=1}^{\infty}$$

Parametry tej sekwencji mają czytelną interpretację geometryczną:

- α oznacza gęstość sekwencji,
- $1/\alpha$ oznacza nachylenie,
- α' oznacza przesunięcie,
- $-\alpha'/\alpha$ oznacza y-przechwycenie (punkt przecięcia z osią rzędnych).

Twierdzenie Beatty gwarantuje podział zbioru dla liczb niewymiernych. Twierdzenie Fraenkela jest uogólnieniem, które – co dla nas kluczowe – dopuszcza również liczby wymierne, pod warunkiem spełnienia pięciu postulatów (przytoczonych w rozdziale wstępnym). Przystępny dowód twierdzenia Fraenkela można odnaleźć w pracy K. O'Bryanta *„Fraenkel's partition and Brown's decomposition"*.

Cała dalsza część tej strony sprowadza się do jednej myśli: pokazania, że operatory strumieniowe są w istocie maszynami generującymi sekwencje Beatty, które dokonują podziału (pokrycia) zbioru liczb naturalnych.

4.2 Narzędzia: własności podłogi i sufitu

Dowody operują niemal wyłącznie na funkcjach podłogi ($\lfloor x \rfloor$ – część całkowita) i sufitu ($\lceil x \rceil$ – najmniejsza liczba całkowita nie mniejsza od x). Przytaczam więc najpierw zestaw tożsamości, które będą wielokrotnie wykorzystywane. Niech $x \in \mathbb{R}$, a C oznacza liczbę całkowitą:

$$\lfloor x \rfloor = \lceil x \rceil \iff x \in \mathbb{N}$$

$$\lfloor x \rfloor + 1 = \lceil x \rceil \iff x \in \mathbb{R} - \mathbb{N}$$

$$\lfloor x + C \rfloor = \lfloor x \rfloor + C \iff C \in \mathbb{N}$$

Dodatkowo, w analizie residuum sekwencji rozplątania wykorzystamy zależności wiążące największy wspólny dzielnik (nwd) z dziedziną ilorazu a/b:

$$\text{nwd}(a, b) = b \iff \frac{a}{b} = c \in \mathbb{N}$$

$$1 \leq \text{nwd}(a, b) \leq a \iff 0 < \frac{a}{b} < 1$$

Te dwa przypadki rozłącznie pokrywają całą interesującą nas dziedzinę – co pozwoli przeprowadzić dowód „przez przypadki".

4.3 Operatory w zapisie formalnym

Operatory wprowadzone w języku zapytań mają swoje formalne odpowiedniki. Poniższa tabela wiąże zapis formalny (stosowany w dowodach) z symbolami spotykanymi w języku zapytań:

Operacja	Symbol formalny	Symbol w języku zapytań
Rzutowanie	π	lista pól po `SELECT`
Selekcja	σ	warunek logiczny
Suma	Σ	`+`
Różnica	δ	`-`
Przeplot (splątanie)	φ	`#`
Rozplątanie i jego dopełnienie	Θ, ~Θ	`&` , `%`
Agregacja i serializacja (AGSE)	Ψ	`@`
Przesunięcie	τ	`>`

Dla samodzielności dowodów przytaczam dwie definicje, do których będę się bezpośrednio odwoływał.

Przeplot φ(A, B) tworzy strumień wynikowy, którego kolejne krotki wyznacza reguła:

$$c_n = \begin{cases} b_{n-\lfloor nz \rfloor} & \lfloor nz \rfloor = \lfloor (n+1)\, z \rfloor \\ a_{\lfloor nz \rfloor} & \lfloor nz \rfloor \neq \lfloor (n+1)\, z \rfloor \end{cases}, \; z = \frac{\Delta_b}{\Delta_a + \Delta_b}, \; \Delta_c = \frac{\Delta_a \Delta_b}{\Delta_a + \Delta_b}$$

Rozplątanie definiują dwa komplementarne wzory – operator Θ odtwarzający pierwotny strumień oraz operator ~Θ wyznaczający „resztę” rozplątania:

$$a_n = c_{n + \left\lceil \frac{(n+1)\Delta_a}{\Delta_b} \right\rceil}, \; \Delta_a = \frac{\Delta_c \Delta_b}{|\Delta_c - \Delta_b|}$$

$$b_n = c_{n + \left\lfloor \frac{n\Delta_b}{\Delta_a} \right\rfloor}, \; \Delta_b = \frac{\Delta_c \Delta_a}{|\Delta_c - \Delta_a|}$$

4.4 Twierdzenie 1: przeplot zapewnia pokrycie zbiorów

Uwaga

Twierdzenie. Operacja splątania (przeplotu) zapewnia sekwencyjne pokrycie obu zbiorów zawierających elementy strumieni danych będących jej argumentami.

Dowód. Dowód rozpoczynamy od analizy pierwszego warunku (warunku równości) w równaniu przeplotu. Oznaczmy ten warunek jako (∗):

$$(*): \quad \lfloor nz \rfloor = \lfloor (n+1)\, z \rfloor$$

Dla każdego n spełniającego warunek (∗) kolejne wartości wyrażenia n − ⌊nz⌋ tworzą drugi, skojarzony ciąg liczb naturalnych, wybierający kolejne elementy z ciągu b. Oznacza to, że dla każdego n spełniającego (∗) wyrażenie x = n − ⌊nz⌋ podlega zależności $x_n = x_{n+1} - 1$. Formalnie:

$$n - \lfloor nz \rfloor = (n+1) - \lfloor (n+1)z \rfloor - 1$$

Podstawiając warunek (∗) do prawej strony otrzymujemy:

$$n - \lfloor (n+1)z \rfloor = (n+1) - \lfloor (n+1)z \rfloor - 1$$

Po prostym uproszczeniu algebraicznym dochodzimy do tożsamości n = (n + 1) − 1, która jest prawdziwa. Tym samym indeksy wybierające elementy ze strumienia b następują po sobie kolejno, bez przerw i powtórzeń. Drugą część dowodu, opartą na warunku nierówności (wybór elementów ze strumienia a), prowadzi się analogicznie.
□

4.5 Twierdzenie 2: rozplątanie spełnia postulaty Fraenkela

To jest centralne twierdzenie tej strony. Dowodzi, że obie sekwencje opisujące operację rozplątania są szczególnym przypadkiem sekwencji Beatty spełniającym postulaty twierdzenia Fraenkela dla liczb wymiernych. Bez tego twierdzenia cały system pozostaje jedynie obietnicą.

Uwaga

Twierdzenie. Operacja rozplątania spełnia postulaty twierdzenia Fraenkela.

Dowód – część pierwsza (sprowadzenie do postaci Beatty). Poszukujemy sposobu przedstawienia sekwencji wyboru kolejnych krotek w operacji rozplątania jako sekwencji Beatty. Sekwencja opisująca wybór krotek ma postać:

$$\left(n + \left\lfloor \frac{nb}{a} \right\rfloor\right)_{n=1}^{\infty}$$

Dla n ∈ ℕ, na mocy własności ⌊x + C⌋ = ⌊x⌋ + C, powyższe równanie można przyrównać do ogólnej postaci sekwencji Beatty:

$$\left(\left\lfloor \frac{n-\alpha'}{\alpha} \right\rfloor\right)_{n=1}^{\infty} = \left(\left\lfloor n + \frac{nb}{a} \right\rfloor\right)_{n=1}^{\infty}$$

Upraszczając lewą stronę i grupując prawą:

$$\left(\left\lfloor n\alpha^{-1} - \frac{\alpha'}{\alpha}\right\rfloor\right)_{n=1}^{\infty} = \left(\left\lfloor n\frac{a+b}{a}\right\rfloor\right)_{n=1}^{\infty}$$

Symbol $-\alpha'/\alpha$ oznacza y-przechwycenie. Jeśli przesunięcie sekwencji $\alpha' = 0$, to $\alpha = a/(a+b)$, a sekwencja przyjmuje postać:

$$\mathcal{B}\left(\frac{a}{a+b}, 0\right) := \left(\left\lfloor n\frac{a+b}{a}\right\rfloor\right)_{n=1}^{\infty}$$

W ten sposób, poprzez kilka prostych przekształceń algebraicznych, otrzymaliśmy postać sekwencji Beatty z sekwencji opisującej wybór kolejnych krotek w operacji rozplątania.

Dowód – część druga (weryfikacja pięciu postulatów i wyznaczenie residuum). Sprawdzamy kolejno postulaty twierdzenia Fraenkela dla wyznaczonej sekwencji:

1. Wartość $\alpha = a/(a+b)$ dla $a, b > 0$ jest większa od zera i mniejsza od jedności.
2. Warunek $\alpha + \beta = 1$ jest spełniony dla $\beta = b/(a+b)$.
3. Dla $\alpha' = 0$ postulat jest równoważny postulatowi 1.
4. Rozwiązań poszukujemy w zbiorze liczb wymiernych (przypadek α wymiernego).
5. Jeśli $q\alpha \in \mathbb{N}$ oraz $q \in \mathbb{N}$ i zachodzi $1/q \leq \alpha + \alpha'$, to – skoro $\alpha' = 0$ – warunek ten jest prawdziwy dla $q \leq (a+b)/\mathrm{nwd}(a,b)$. Wynika stąd, że $\lceil((a+b)/\mathrm{nwd}(a,b)) \cdot \beta'\rceil = 1$, czyli $\beta' = \mathrm{nwd}(a,b)/(a+b)$.

Postać ciągu residuum (ciągu dopełniającego) dla sekwencji $\mathcal{B}(a/(a+b), 0)$, spełniająca postulaty twierdzenia Fraenkela, przedstawia się więc następująco:

$$\mathcal{B}\left(\frac{b}{a+b}, \frac{\mathrm{nwd}(a,b)}{a+b}\right)$$

Przyjmujemy, że podział zbioru liczb naturalnych następuje w oparciu o tę sekwencję:

$$\mathcal{B}\left(\frac{b}{a+b}, \frac{\mathrm{nwd}(a,b)}{a+b}\right) = \left(\left\lfloor \frac{(n+1) - \frac{\mathrm{nwd}(a,b)}{a+b}}{\frac{b}{a+b}} \right\rfloor\right)_{n=1}^{\infty}$$

Po opuszczeniu nawiasów opisujących sekwencję i wykonaniu kilku prostych przekształceń można wykazać, że:

$$\left\lfloor \frac{(n+1) - \frac{\mathrm{nwd}(a,b)}{a+b}}{\frac{b}{a+b}} \right\rfloor := \left\lfloor n\frac{a}{b} + n + \frac{a}{b} + 1 - \frac{\mathrm{nwd}(a,b)}{b} \right\rfloor$$

Poszukiwane równanie opisujące proces tworzenia sekwencji wyboru krotek przedstawia się następująco:

$$\left\lfloor n\frac{a}{b} + n + \frac{a}{b} + 1 - \frac{\text{nwd}(a,b)}{b} \right\rfloor = n + \left\lceil \frac{(n+1)a}{b} \right\rceil$$

Stąd, po wydzieleniu części całkowitej, otrzymujemy:

$$\left\lfloor \frac{(n+1)a}{b} - \frac{\text{nwd}(a,b)}{b} \right\rfloor + 1 = \left\lceil \frac{(n+1)a}{b} \right\rceil$$

Podstawiając za n + 1 liczbę naturalną n, otrzymujemy równość, którą należy udowodnić:

$$\left\lfloor n\frac{a}{b} - \frac{\text{nwd}(a,b)}{b} \right\rfloor + 1 = \left\lceil n\frac{a}{b} \right\rceil$$

Dowód – część trzecia (analiza przypadków). Korzystając z własności współczynnika nwd(a, b), rozważamy dwa rozłączne przypadki pokrywające całą dziedzinę.

Przypadek 1: nwd(a, b) = b, czyli a/b = c ∈ ▯. Dowodzone równanie przyjmuje postać:

$$\left\lfloor \frac{(n+1)a}{b} - \frac{b}{b} \right\rfloor + 1 = \left\lceil \frac{(n+1)a}{b} \right\rceil$$

Uwzględniając tożsamości $\lfloor x \rfloor = \lceil x \rceil \iff x \in \mathbb{N}$ oraz $\lfloor x + C \rfloor = \lfloor x \rfloor + C$, a także dziedzinę tego przypadku, stwierdzamy, że obie sekwencje tworzą te same elementy.

Przypadek 2: $1 \leq$ nwd(a, b) $\leq a$, czyli $0 < a/b < 1$. Załóżmy, że istnieją takie dwie liczby a i b, dla których dowodzone równanie nie jest prawdziwe, tzn. dla wartości $n \cdot a/b - \text{nwd}(a,b)/b$ oraz $n \cdot a/b$ należących do $\mathbb{N}$ nie zachodzi:

$$\left\lfloor n\frac{a}{b} - \frac{\text{nwd}(a,b)}{b} + 1 \right\rfloor \neq \left\lceil n\frac{a}{b} \right\rceil$$

Korzystając z własności podłogi i sufitu, poszukujemy takich a i b, że:

$$n\frac{a}{b} - \frac{\text{nwd}(a,b)}{b} + 1 \neq n\frac{a}{b}$$

Równanie to jest spełnione jedynie dla nwd(a, b) = b, a w rozważanej dziedzinie $1 \leq$ nwd(a, b) $\leq a$ nie ma ono rozwiązań. Nie istnieją zatem takie a i b należące do tej dziedziny, które przeczyłyby dowodzonemu równaniu.

Rozpatrzmy jeszcze drugą własność ($\lfloor x \rfloor + 1 = \lceil x \rceil \iff x \in \mathbb{R} - \mathbb{N}$). Załóżmy, że istnieją dwie liczby a i b, dla których równanie nie jest spełnione, czyli dla $n \cdot a/b - \text{nwd}(a,b)/b$ oraz $n \cdot a/b$ należących do ℝ▯ℕ powinna zawsze zachodzić zależność:

$$n\frac{a}{b} - \frac{\text{nwd}(a,b)}{b} \neq n\frac{a}{b}$$

Nie istnieją jednak dwie takie liczby, dla których nwd(a, b) = 0. Czyli dla $a/b \in \mathbb{R} - \mathbb{N}$ równanie to jest zawsze prawdziwe.

Tak więc oba równania opisujące operację rozplątania są przypadkiem sekwencji Beatty spełniającym postulaty twierdzenia Fraenkela dla liczb wymiernych. □

Ostrzeżenie

Praktyczny morał z tego dowodu: w implementacji nie wolno opuszczać dziedziny liczb wymiernych nawet na chwilę. Niejawne rzutowanie wyniku pośredniego na liczbę zmiennoprzecinkową łamie założenia powyższego twierdzenia. Materializację do postaci zmiennoprzecinkowej należy odłożyć do momentu jawnego zastosowania operacji podłogi lub sufitu.

4.6 Własności operatorów wykorzystywane w optymalizacji

W oparciu o przedstawioną algebrę można wykazać szereg własności strumieni danych. Mają one bezpośrednie zastosowanie w systemie zarządzania danymi – w trakcie optymalizacji planów zapytań oraz interpretacji wyników.

Zaburzenie kolejności zdarzeń

Uwaga

Twierdzenie. Kolejność elementów w strumieniu nie odzwierciedla faktycznej kolejności występowania elementów w świecie rzeczywistym.

Dowód (przez kontrprzykład). Rozważmy dwa strumienie:

```
Alfa(znak),2:    {1,2,3,4,5,6,...}
Epsilon(znak),3: {a,b,c,d,e,f,...}
```

Wyrażenie φ(Epsilon, Alfa) tworzy strumień wynikowy:

```
Tau(znak),6/5:   {1,2,a,3,b,4,5,c,6,d,...}
```

W strumieniu Tau krotka oznaczona literą `c` występuje po krotce oznaczonej cyfrą `5`. Tymczasem krotka `c` pojawia się w strumieniu Epsilon w 9. sekundzie, a krotka `5` w strumieniu Alfa – w 10. sekundzie. Naturalny porządek zdarzeń został w strumieniu wynikowym naruszony. Wniosek: prowadząc analizę względem czasu zawartego w strumieniach, konieczne jest zastosowanie operacji rozplątania w celu uzyskania pierwotnej postaci strumieni danych. □

Przemienność sumowania

Uwaga

Twierdzenie. Operacja sumowania strumieni danych, z pominięciem kolejności atrybutów, jest przemienna.

Dowód. Załóżmy, że C = Σ(A, B) oraz D = Σ(S, A). Korzystając ze wzoru na sumę strumieni danych, zapisujemy obie zależności i pomijamy kolejność atrybutów wynikającą z operacji połączenia krotek. Zmieniając kolejność warunków w definicji D oraz podstawiając za symbol S symbol B, otrzymujemy wzór tożsamy ze wzorem na C. Przypadek równych wartości Δ obu strumieni jest trywialny i został pominięty. Dowodzi to przemienności operacji sumowania. ∎

Metoda dopasowania przeplotu

Operacja przeplotu nie jest przemienna (co pokazano w rozdziale o algebrze). Istnieje jednak algebraiczna metoda umożliwiająca zmianę kolejności jej argumentów przy określonych założeniach – co jest cenne w optymalizacji planów zapytań.

Uwaga

Twierdzenie. Jeśli wybierzemy dwie liczby naturalne i, k, których stosunek jest równy stosunkowi wartości Δ strumieni łączonych przeplotem, to przeplot strumieni przesuniętych względem tych wartości tworzy strumień równy strumieniowi powstałemu przez przeplot z zamienioną kolejnością argumentów i przesunięciem o sumę tych liczb.

Formalnie:

$$\varphi\left(\tau_i(A), \tau_k(B)\right) = \tau_{i+k}\left(\varphi(B, A)\right), \quad \frac{i}{k} = \frac{\Delta_a}{\Delta_b}, \quad i, k \in \mathbb{N}$$

Dowód. Analizując lewą stronę równania i korzystając z definicji przeplotu, otrzymujemy:

$$\varphi\left(\tau_i(A), \tau_k(B)\right): \quad c_n = \begin{cases} b_{(n-\lfloor nz \rfloor)+i} & \lfloor nz \rfloor = \lfloor (n+1)\, z \rfloor \\ a_{\lfloor nz \rfloor + k} & \lfloor nz \rfloor \neq \lfloor (n+1)\, z \rfloor \end{cases}$$

Analizując prawą stronę równania, otrzymujemy:

$$\tau_{i+k}\left(\varphi(B, A)\right): \quad c_n = \begin{cases} a_{\lfloor (n+i+k) z \rfloor} & \lfloor nz \rfloor = \lfloor (n+1)\, z \rfloor \\ b_{n+i+k-\lfloor (n+i+k) z \rfloor} & \lfloor nz \rfloor \neq \lfloor (n+1)\, z \rfloor \end{cases}$$

Porównując warunki, dla których oba równania wybierają próbki ze strumienia B, oraz zakładając poprawność tezy, stwierdzamy, że $-\lfloor nz \rfloor = k - \lfloor (n+i+k)z \rfloor$. Jednocześnie, z założenia o stosunku liczb:

$$i + k = \frac{\Delta_a}{\Delta_b} k + k = k\left(\frac{\Delta_a}{\Delta_b} + 1\right) = \frac{k}{z}$$

Łącząc obie zależności, dochodzimy do równania:

$$-\lfloor nz \rfloor = k - \lfloor k + nz \rfloor$$

Ponieważ z założenia $k \in \mathbb{N}$, na mocy własności $\lfloor x + C \rfloor = \lfloor x \rfloor + C$ powyższe równanie jest spełnione. Druga część dowodu, prowadzona w oparciu o warunek nierówności, jest analogiczna. □

4.7 Dlaczego to ma znaczenie

Przedstawione twierdzenia nie są formalnością dla samej formalności. Każde z nich pełni konkretną rolę w działającym systemie:

- **Twierdzenie 1 i 2** gwarantują, że pary operacji przeplot/rozplątanie oraz suma/różnica są komplementarne – dane nie giną i nie powielają się w sposób niekontrolowany. To one pozwalają traktować te operacje jak mnożenie/dzielenie oraz dodawanie/odejmowanie w zbiorze regularnych serii czasowych.
- **Twierdzenie 2** w szczególności udowadnia, że całą konstrukcję da się zrealizować wyłącznie na liczbach wymiernych – a więc deterministycznie i dokładnie na komputerze. To jest warunek, bez którego system RetractorDB nie mógłby istnieć.
- **Twierdzenia o własnościach operatorów** (przemienność sumowania, dopasowanie przeplotu, zaburzenie kolejności) dostarczają reguł przepisywania wyrażeń strumieniowych. Optymalizator planów zapytań korzysta z nich, aby przekształcać plany do postaci tańszej w realizacji, nie zmieniając wyniku.

Dział matematyki, w którym osadzone są te równania, to teoria układów pokrywających [4] w obszarze teorii liczb. Pełny formalizm wraz z kompletem dowodów przedstawiłem w pracy Deterministyczna metoda przetwarzania ciągów danych [3].

Info

Numeryczna weryfikacja powyższych równań – prototypy w języku Python operujące na liczbach wymiernych (biblioteka `Fraction`) – znajduje się na stronie Implementacja modelu oraz w repozytorium github.com/michalwidera/equations.

Rozdział 5

Wyrażenia algebraiczne

Zdefiniowana algebra pociąga za sobą możliwość definicji wyrażeń algebraicznych. Typowe wyrażenia algebraiczne w zbiorze liczb wymiernych to materiał przerabiany w szkole podstawowej. Wyrażenia algebraiczne w systemie RetractorDB występują w dwóch formach. Na liście pól polecenia SELECT – mamy wyrażenia typowe, znane ze szkoły podstawowej. Na liście argumentów polecenia SELECT w klauzuli FROM mamy wyrażenie algebraiczne zbudowane w oparciu o nową, zdefiniowaną algebrę.

Oznacza to że na liście pól po klauzuli SELECT operator plus oznacza jedno a w klauzuli FROM – oznacza zupełnie coś innego. Niewinnie wyglądające zapytanie z definicji łączy dwa zupełnie inne światy i pojęcia. Jeden algebry opartej na liczbach drugiej opartej na regularnych seriach czasowych.

Przykład. Jako przykład przedstawione zostanie wyrażenie algebraiczne zbudowane w zbiorze regularnych serii czasowych (zwanych dalej strumieniami). Zakładając istnienie dwóch strumieni: A(a1 int, a2 int),1 oraz B(b1 int),½ – gdzie,

- A oznacza strumień zawierający w każdym rekordzie dwa pola o wartościach typu int – a1 oraz a2, napływające raz na sekundę, oraz
- B zawierający w każdym rekordzie pole typu int o nazwie b1 napływające dwa razy na sekundę.

To wyrażenie algebraiczne postaci C=A+B stworzy strumień danych o polach C(a1 int, a2 int, b1 int),½.

Aby dokonać przeplotu strumienia danych zbiory A i B powinny posiadać te same schematy danych. Załóżmy więc że istnieje strumień D(d1 int),1 – napływający podobnie jak strumień A – raz na sekundę.

To wyrażenie algebraiczne postaci E=B#D stworzy strumień: E(e1 int),⅓. Szybkość ⅓ bierze się ze wzoru (1*½)/(1+½). Wzór znajdziesz przy definicji operacji przeplotu.

W tak zdefiniowanych strumieniach nadal poprawne jest wyrażenie:

```
F=((B#D)+A)>2
```

I takie wyrażenia mogą się pojawić jako poprawne względem opracowanej algebry szeregów czasowych w treści zapytania.

Rozdział 6

Implementacja modelu

Opracowane równania algebry zaimplementowano pierwotnie w języku Python. Jest to znany mi najbardziej efektywny sposób modelowania i numerycznego weryfikowania hipotez. Każdy z operatorów został zaimplementowany wewnątrz osobnej funkcji. Operacje realizowane na zmiennych wymiernych (biblioteka Fraction). Wyniki prezentowane są w postaci ograniczonych tablic. Operatory te jednak w końcowej implementacji realizują operacje na nieskończonych strukturach danych.

Operacja przeplotu

Na początku zbudujmy operację przeplotu:

Kod źródłowy

```
# Operacja splątania (hash) dwóch list z określonymi krokami (delta).
from fractions import Fraction
from math import floor, ceil

A = range(1, 24)
deltaA = Fraction(1, 2)
B = list(map(chr, range(ord('a'), ord('z')+1)))
deltaB = Fraction(1, 2)

def hash(A: list, deltaA: Fraction, B: list, deltaB: Fraction):
  result = []
  delta = deltaB / (deltaA + deltaB)
  for i in range(0, 20):
      if floor(i*delta) == floor((i+1)*delta):
          result.append(B[i-int(floor((i+1)*delta))])
      else:
          result.append(A[int(floor(i*delta))])
  deltaC = (deltaA*deltaB)/(deltaA+deltaB)
  return result, deltaC

def main():
```

```
    print("A:", A[0:10], " deltaA:", deltaA)
    print("B:", B[0:10], " deltaB:", deltaB)
    hash_result1, delta_hash1 = hash(A, deltaA, B, deltaB)
    hash_result2, delta_hash2 = hash(B, deltaB, A, deltaA)
    print("Hash(A,B):", hash_result1[0:10], " deltaHash:", delta_hash1)
    print("Hash(B,A):", hash_result2[0:10], " deltaHash:", delta_hash2)

if __name__ == '__main__':
    main()
```

Efekt uruchomienia

```
$ python hash.py
A: range(1, 11) deltaA: 1/2
B: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'] deltaB: 1/2
Hash(A,B): ['a', 1, 'b', 2, 'c', 3, 'd', 4, 'e', 5] deltaHash: 1/4
Hash(B,A): [1, 'a', 2, 'b', 3, 'c', 4, 'd', 5, 'e'] deltaHash: 1/4
```

Kod po uruchomieniu przedstawi dane wejściowe A oraz B - oraz wyniki operacji A#B oraz B#A. Jak widać operacja przeplotu nie jest przemienna.

Operacja rozplątania

Operacja rozplątania wymaga zaimplementowania dwóch komplementarnych operacji.

Kod źródłowy - even

```
# Operacja rozplątania (dehash) even.
from fractions import Fraction
from math import floor, ceil

A = range(1, 24)
deltaA = Fraction(1, 2)
B = list(map(chr, range(ord('a'), ord('z')+1)))
deltaB = Fraction(1, 2)

def hash(A: list, deltaA: Fraction, B: list, deltaB: Fraction):
  result = []
  delta = deltaB / (deltaA + deltaB)
  for i in range(0, 20):
      if floor(i*delta) == floor((i+1)*delta):
          result.append(B[i-int(floor((i+1)*delta))])
      else:
          result.append(A[int(floor(i*delta))])
  deltaC = (deltaA*deltaB)/(deltaA+deltaB)
  return result, deltaC

def dehasheven(C: list, deltaC: Fraction, deltaA: Fraction):
```

```
  result = []
  deltaB = deltaA*deltaC / (deltaA - deltaC)

  for i in range(0, 6):
      result.append(C[i+int(ceil((i+1)*deltaA/deltaB))])
  return result, deltaB

def main():
    hash_result, delta_hash = hash(B, deltaB, A, deltaA)
    print("Hash(A,B):", hash_result[0:10], " deltaHash:", delta_hash)
    mod_result, delta_mod = dehasheven(hash_result, delta_hash, deltaA)
    print("Mod(Hash):", mod_result[0:10], " deltaMod:", delta_mod)

if __name__ == '__main__':
    main()
```

---

wynik - even

```
$ python dehash_even.py
Hash(A,B): [1, 'a', 2, 'b', 3, 'c', 4, 'd', 5, 'e'] deltaHash: 1/4
Mod(Hash): ['a', 'b', 'c', 'd', 'e', 'f'] deltaMod: 1/2
```

Kod źródłowy - odd

```
# Operacja rozplątania (dehash) odd.
from fractions import Fraction
from math import floor, ceil

A = range(1, 24)
deltaA = Fraction(1, 2)
B = list(map(chr, range(ord('a'), ord('z')+1)))
deltaB = Fraction(1, 2)

def hash(A: list, deltaA: Fraction, B: list, deltaB: Fraction):
  result = []
  delta = deltaB / (deltaA + deltaB)
  for i in range(0, 20):
      if floor(i*delta) == floor((i+1)*delta):
          result.append(B[i-int(floor((i+1)*delta))])
      else:
          result.append(A[int(floor(i*delta))])
  deltaC = (deltaA*deltaB)/(deltaA+deltaB)
  return result, deltaC

def dehashodd(C: list, deltaC: Fraction, deltaB: Fraction):
```

```python
  result = []
  deltaA = deltaB*deltaC / (deltaB - deltaC)

  for i in range(0, 6):
      result.append(C[i+int(i*deltaB/deltaA)])
  return result, deltaA

def main():
    hash_result, delta_hash = hash(B, deltaB, A, deltaA)
    print("Hash(A,B):", hash_result[0:10], " deltaHash:", delta_hash)
    div_result, delta_div = dehashodd(hash_result, delta_hash, deltaB)
    print("Div(Hash):", div_result[0:10], " deltaDiv:", delta_div)

if __name__ == '__main__':
    main()
```

---

wynik - odd

```
$ python dehash_odd.py
Hash(A,B): [1, 'a', 2, 'b', 3, 'c', 4, 'd', 5, 'e']  deltaHash: 1/4
Div(Hash): [1, 2, 3, 4, 5, 6]  deltaDiv: 1/2
```

Tak zbudowany kod najpierw łączy dwa strumienie a następnie wyciąga dane źródłowe.

Operacja sumy

Sumowanie łączy dwa strumienie danych napływające z różną czestotliwością.

Kod źródłowy

```python
# operacja sumowania dwóch list z określonymi krokami (delta).
from fractions import Fraction
from math import floor, ceil

A = range(1, 24)
deltaA = Fraction(1, 2)
B = list(map(chr, range(ord('a'), ord('z')+1)))
deltaB = Fraction(1)

def sum(A: list, deltaA: Fraction, B: list, deltaB: Fraction):
  result = []
  deltaC = min(deltaA, deltaB)
  for i in range(0, 20):
      if deltaC == deltaA:
          result.append(str(A[i])+B[int(i*deltaA/deltaB)]),
      else:
          result.append(str(A[int(i*deltaB/deltaA)])+B[i]),
  return result, deltaC
```

```
def main():
    print("A:", A[0:10], " deltaA:", deltaA)
    print("B:", B[0:10], " deltaB:", deltaB)
    sum_result, delta_sum = sum(A, deltaA, B, deltaB)
    print("Sum:", sum_result[0:10], " deltaSum:", delta_sum)

if __name__ == '__main__':
    main()
```

---

wynik

```
$  python sum.py
A: range(1, 11)  deltaA: 1/2
B: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']  deltaB: 1
Sum: ['1a', '2a', '3b', '4b', '5c', '6c', '7d', '8d', '9e', '10e']  deltaSum: 1/2
```

Operacja różnicy

Komplementarną operacją dla sumy jest operacja różnicy.

Kod źródłowy

```
# Operacja różnicy (diff) dwóch list z określonymi krokami (delta).
from fractions import Fraction
from math import floor, ceil

A = range(1, 24)
deltaA = Fraction(1, 2)
B = list(map(chr, range(ord('a'), ord('z')+1)))
deltaB = Fraction(1)

def sum(A: list, deltaA: Fraction, B: list, deltaB: Fraction):
  result = []
  deltaC = min(deltaA, deltaB)
  for i in range(0, 20):
      if deltaC == deltaA:
          result.append(str(A[i])+B[int(i*deltaA/deltaB)]),
      else:
          result.append(str(A[int(i*deltaB/deltaA)])+B[i]),
  return result, deltaC

def diff(C: list, deltaA: Fraction, deltaB: Fraction):
  result = []
  deltaC = min(deltaA, deltaB)
  for i in range(0, 10):
      if deltaA > deltaB:
          result.append(C[int(ceil(i*deltaA/deltaB))])
```

```
        else:
            result.append(C[i])
    return result, deltaC

def main():
    sum_result, delta_sum = sum(A, deltaA, B, deltaB)
    diff_result, delta_diff = diff(sum_result, deltaA, deltaB)
    print("Sum:", sum_result[0:10], " deltaSum:", delta_sum)
    print("Diff(Sum):", diff_result[0:10], " deltaDiff:", delta_diff)

if __name__ == '__main__':
    main()
```

---

wynik

```
$ python diff.py
Sum: ['1a', '2a', '3b', '4b', '5c', '6c', '7d', '8d', '9e', '10e']  deltaSum: 1/2
Diff(Sum): ['1a', '2a', '3b', '4b', '5c', '6c', '7d', '8d', '9e', '10e']  deltaDiff: 1/2
```

Kody źródłowe

Kody źródłowe przedstawionych przykładów zjadują się w repozytorium projektu w katalogu /examples/python-model/

Implementacja w języku javascript możliwa do przetestowania bezpośrednio na sto-nie:

https://retractordb.com/assets/interlace.html

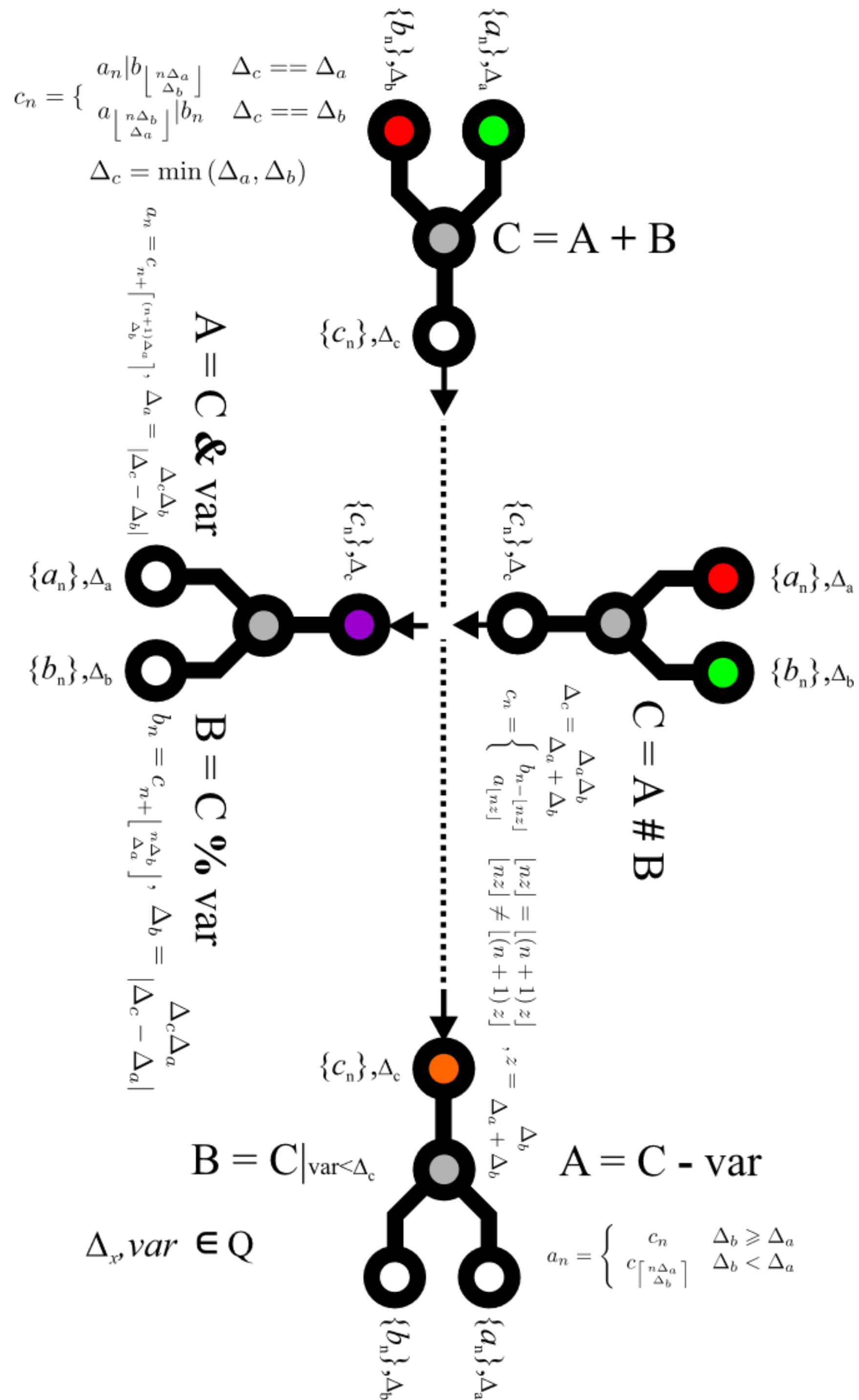
https://retractordb.com/assets/sum.html

Rozdział 7

Reprezentacja graficzna

Na Rys. 2 przedstawiono schematycznie zależności pomiędzy opracowanymi operatorami algebry serii czasowych. Na rysunku połączyłem zależności opracowanych operatorów, ich symboliczne oznaczenia stosowane w języku zapytań oraz kierunki przetwarzania danych.

Przedstawiony rysunek stanowi też graficzne podsumowanie treści zaprezentowanych w tym rozdziale. Przedstawiony graficzny sposób reprezentacji mam nadzieję ułatwi przyswojenie zasad panujących pomiędzy wprowadzonymi operatorami. Dla czytelności pominięte zostały operatory agregacji i serializacji oraz przesunięcia czasowego. Należy mieć świadomość że do pełnego obrazu brakuje ich na tym schemacie.



Rys. 2. Zależności pomiędzy operatorami algebry

Rozdział 8

Podsumowanie

Równania te z początku modelowałem w postaci programów w języku Python. Przedstawioną formalną formę przyjęły na sam koniec procesu poszukiwań. Dowodząc numerycznie poprawności opracowanych równań konstruowałem sekwencje operacji na strumieniach. Jeśli jakieś elementy gubiły się w trakcie realizacji przedstawionych operacji – oznaczało to że popełniłem błąd. Okazuje się np. ze istotne jest w implementacji aby nie opuszczać nawet na chwilę dziedziny liczb wymiernych. Błąd można popełnić przypadkiem, niejawnie rzutując wynik na liczbę zmiennoprzecinkową. Materializację wyniku w formie zmiennoprzecinkowej należy w obliczeniach odłożyć do momentu jawnego przeniesienia wyniku operacją podłogi lub sufitu. Jeśli program w Pytonie złożymy w sekwencję operacji na nieskończonych strumieniach i żadne dane w wyniku tej operacji nie znikną – mamy obiekt do dalszych badań i analizy formalnej, gotowy do formalnego dowodu matematycznego poprawności. Formalny dowód (formalizm matematyczny) znajdziemy w pracy pt. Deterministyczna metoda przetwarzania ciagow danych [3].

Dział matematyki który zawiera prace badawcze związane z tymi równaniami nosi nazwę systemów pokrywających [4] w obszarze teorii liczb.

Info

Przedstawienie podstaw matematycznych systemu jest konieczne w celu zrozumienia dalszych technicznych aspektów rozwiązania. Przedstawione metody wybiegają poza standardowy materiał prezentowany obecnie na studiach z zakresu nauk technicznych. Wynika to z faktu, że podstawy matematyczne wydobyłem z obszaru dotychczas niemającego zastosowań w znanej mi technice. Są to metody umożliwiające zbudowanie nowego sposobu przetwarzania danych. Na tym polega jeden z aspektów różniących RetractorDB od reszty podobnych rozwiązań.

Rozdział 9

Konstrukcja języka zapytań

Komunikacja pomiędzy opracowanym systemem a użytkownikiem odbywa się za pomocą opracowanego, deklaratywnego języka zapytań. Konstrukcja języka oparta jest na przedstawionej w poprzednim rozdziale algebrze. Podobnie jak w przypadku systemów relacyjnych, gdzie algebra relacji tworzy podstawę dla języka SQL – w moim przypadku opracowana algebra tworzy podstawę dla języka zapytań RQL.

RQL to skrót od *RetractorDB Query Language*. Jego składnia jest bardzo podobna do składni języka SQL. Należy mieć jednak na uwadze, że właściwym określeniem w tym przypadku jest ang. *False Friend*. Czyli wygląda to jak SQL, ale nie ma z nim zbyt wiele wspólnego.

Poprawne zdania w języku RQL na chwilę obecną zaczynają się od kilku słów kluczowych. Najbardziej rozpoznawalne to polecenie zaczynające się od słowa kluczowego SELECT za którym występuje lista atrybutów w postaci wyrażeń algebraicznych. Algebry opartej na liczbach rzeczywistych.

Polecenia zapisuje się w pliku tekstowym. Jego rozszerzenie to zwyczajowo .rql ale dowolne inne też zostanie przyjęte i przetworzone. Plik tekstowy języka RQL zawiera ciąg poleceń zaczynających się od zdefiniowanych słów kluczowych. Komentarze poprzedza się znakiem #.

Język zapytań został zaimplementowany przy pomocy generatora parserów Antlr4 [5]. Gramatyka języka RQL została zapisana, zdefiniowana i po każdej modyfikacji jest kompilowana do języka w którym stworzono system RetractorDB. Każde zdanie pliku zbioru zapytań nie będące komentarzem jest kompilowane, przetwarzane i modyfikuje wewnętrzny stan systemu.

Rozdział 10

Polecenie DECLARE

Polecenie DECLARE służy do zadeklarowania źródła danych.

Jego składnia opisana jest następująco:

```
DECLARE pole typ[N] [, pole typ[N]]
STREAM nazwa, szybkość
FILE źródło
[DISPOSABLE]
[ONESHOT]
[HOLD]
```

10.1 Typy pól

Każde pole ma nazwę i typ. Dostępne typy:

Typ	Rozmiar	Opis
`BYTE`	1 B	liczba całkowita bez znaku 8-bit
`INTEGER`	4 B	liczba całkowita ze znakiem 32-bit
`UINT`	4 B	liczba całkowita bez znaku 32-bit
`FLOAT`	4 B	liczba zmiennoprzecinkowa 32-bit
`DOUBLE`	8 B	liczba zmiennoprzecinkowa 64-bit
`STRING`	N B	ciąg bajtów o stałej długości N

Tablice pól (`typ[N]`)

Do każdego pola można dodać mnożnik tablicowy `[N]` — pole zajmuje `N × rozmiar_typu` bajtów i tworzy `N` kolejnych pozycji w schemacie rekordu:

```
DECLARE coef INTEGER[25]
STREAM filter, 1
FILE 'coefficients.txt'
```

Pole `coef INTEGER[25]` tworzy rekord o rozmiarze 25 × 4 = 100 bajtów i daje dostęp do indeksów `filter[0]` ... `filter[24]`. Jest to standardowy sposób przekazywania tablic współczynników (np. filtry FIR) do systemu.

Wiele pól różnych typów można łączyć w jednym rekordzie:

```
DECLARE id UINT, wartosc FLOAT, nazwa STRING[16]
STREAM pomiar, 0.1
FILE 'czujnik.dat'
```

Rozmiar rekordu: 4 + 4 + 16 = 24 bajty.

System RetractorDB działając pod kontrolą systemu Linux pobiera i zapisuje dane do plików. W systemie Linux dostęp do większości zasobów jest realizowany za pomocą dostępu do różnego rodzaju plików. Takie rozwiązanie ujednolica sposób dostępu do danych.

Przykładem polecenia tworzącego w systemie RetractorDB obiekt zwracający wartości przypadkowe ze strumienia /dev/random 10 razy na sekundę o wartościach typu int wygląda następująco

```
DECLARE pole_przypadkowe INTEGER
STREAM random_stream, 0.1
FILE ‘/dev/random’
```

Wspominane w poleceniu źródło, jeśli zostanie zadeklarowane jako plik tekstowy z rozszerzeniem .txt zostanie zinterpretowane przez system jako ciągły i nieskończony plik danych czytany wiersz po wierszu. Po napotkaniu końca pliku, odczyt danych zaczyna się od początku. Ta funkcjonalność została wbudowana w system RetractorDB. Zapewnione jest podstawowe wsparcie dla formatu – jeśli podamy dwa pola całkowite w deklaracji a w pliku po spacji podamy dwie wartości całkowite – wartości te trafią jako kolejne elementy czytanego rekordu.

```
DECLARE pole_1 INTEGER
STREAM cykliczny_stream, 0.1
FILE ‘plik.txt’
```

Aby parsowanie pliku nastąpiło automatycznie, plik musi nosić rozszerzenie .txt. Na chwilę ta funkcjonalność została zaimplementowana na stałe i nie podlega parametryzacji. Planuję to zmienić w przyszłości.

Uwaga

Opisana funkcjonalność ma pokrycie w teście: `Pattern7` opisanym w załączniku pt. Testy Integracyjne.

Jeśli plik danych wejściowych będzie nosić rozszerzenie .dat – plik ten zostanie potraktowany jako plik binarny a odczyt danych z niego zostanie również zapętlony. Zapętlenie polega na tym że po przeczytaniu ostatniej wartości z pliku źródłowego, pozycja odczytu pliku kierowana jest na początek. Dane z takiego pliku czytane są w nieskończonej pętli, po zakończeniu wracając do początku.

Trzy opcjonalne dyrektywy (`ONESHOT`, `DISPOSABLE`, `HOLD`) sterują cyklem życia źródła danych — szczegółowy opis i tabela porównawcza znajdują się w rozdziale Opcje odczytu.

Info

Obsługa wartości NULL (per-pole) jest zaimplementowana w systemie RetractorDB. Metadane null przechowywane są w pliku `.meta` obok danych binarnych, zarządzanym przez klasę `metaDataStream`.

Rozdział 11

Opcje odczytu w DECLARE

Polecenie `DECLARE` przyjmuje trzy opcjonalne dyrektywy wpływające na sposób odczytu i cykl życia zadeklarowanego źródła:

```
DECLARE pole typ STREAM nazwa, szybkość FILE źródło
    [DISPOSABLE]
    [ONESHOT]
    [HOLD]
```

11.1 ONESHOT

Bez `ONESHOT` źródło danych czytane jest w nieskończonej pętli — po osiągnięciu końca pliku pozycja odczytu wraca na początek. `ONESHOT` wyłącza pętlę: plik czytany jest dokładnie raz, a po jego wyczerpaniu strumień zwraca wartości zerowe lub puste.

```
DECLARE pomiar INTEGER STREAM burst, 0.1 FILE 'dane.dat' ONESHOT
```

Zastosowanie: jednorazowe załadowanie danych historycznych do systemu.

11.2 DISPOSABLE

Po zakończeniu przesyłania danych ze źródła system usuwa plik danych, plik deskryptora (`.desc`) i plik metadanych (`.meta`). Dyrektywa działa przy destrukcji obiektu `storage`.

```
DECLARE temp INTEGER STREAM jednorazowy, 0.1 FILE 'temp.dat' ONESHOT DISPOSABLE
```

`DISPOSABLE` używa się razem z `ONESHOT` — dane wczytane raz, po wczytaniu usunięte. Kombinacja przydatna do tymczasowych plików danych wejściowych.

11.3 HOLD

Zadeklarowane źródło nie inicjuje odczytu od razu po starcie systemu. Fizyczny odczyt danych uruchamia się dopiero przy pierwszym zapytaniu wymagającym danych

z tego strumienia (np. zapytanie Ad Hoc). Dopóki strumień nie zostanie odpytany — w systemie widoczne są wartości zerowe lub puste.

```
DECLARE rzadkie INTEGER STREAM opcjonalny, 1.0 FILE 'rzadkie.dat' HOLD
```

Zastosowanie: źródła danych aktywowane warunkowo, np. na żądanie użytkownika przez `xqry`.

11.4 Tabela porównawcza

Dyrektywa	Pętla odczytu	Usuwa pliki po odczycie	Opóźniony start odczytu
(domyślnie)	tak	nie	nie
`ONESHOT`	nie	nie	nie
`DISPOSABLE`	tak	tak	nie
`HOLD`	tak	nie	tak

Rozdział 12

Polecenie SELECT

Każde polecenie SELECT w systemie RetractorDB tworzy ciągłe zapytania. Zapytania te realizowane są od momentu pojawienia się w systemie aż do zakończenia pracy systemu.

Składnia polecenia SELECT przedstawia się następująco:

```
SELECT wyrażenie_algebraiczne [, wyrażenie_algebraiczne]
STREAM nazwa_budowanego_strumienia
FROM strumieniowe_wyrażnie_algebraiczne
[FILE 'nazwa_pliku_artefaktu']
[RETENTION pojemność [segmenty]]
[VOLATILE]
[STORAGE profile]
```

Osoby posługujące się językiem SQL zauważą od razu że przedstawione powyżej polecenie odbiega znacząco od tego co znają z zakresu relacyjnych baz danych.

Pierwsza różnica poza składnią to fakt że polecenia te wprowadzone do systemu realizują się aż do zakończenia pracy systemu. Każde polecenie SELECT jest zapytaniem ciągłym. Klauzula STREAM wymaga nadania przez twórcę każdemu zapytaniu unikalnej nazwy. O ile wyrażenia algebraiczne na liście klauzuli SELECT nie odbiegają od formy znanej z systemów relacyjnych o tyle strumieniowe wyrażenie algebraiczne musi spełniać warunki przedstawione w poprzednim rozdziale dotyczącym wyrażeń algebraicznych. Opcjonalne klauzule FILE oraz RETENTION zapewniają procesy kierowania wyników i zarządzania formą ich retencji. Stare, podzielone pliki wynikowe mogą być usuwane na bieżąco zapewniając systemowi miejsce na nowe dane w ruchu ciągłym.

Przykładem zapytania tworzącego nowy strumień danych może być następujące polecenie w języku RQL.

```
SELECT str1[0]*10 + str1[1]*10, str1[2]
STREAM str1
FROM A+B
```

Tak zbudowane zapytanie zakłada że ktoś zadeklarował strumienie A i B. Operację tą mógł wykonać za pomocą słowa kluczowego DECLARE lub innego polecenia SELECT.

W oparciu tylko o wiersz zawierający zapytanie nie jesteśmy w stanie stwierdzić jak szybko dane strumienia str1 napływają. Ta informacja jest wyliczana na etapie kompilacji w oparciu o strumienie A i B i wyrażenie algebraiczne w klauzuli FROM.

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `simple`, `Pattern2` opisanych w załączniku pt. Testy Integracyjne.

Klauzula VOLATILE - tworzy ulotną formę zapytania. Zapytanie z tą klauzulą przechowują tylko jeden rekord w pamięci - na dysku pojawia się tylko deskryptor opisujący strukturę danych.

Klauzula STORAGE umożliwia wybór sposobu tworzenia i zarządzania tworzonymi artefaktami. Pełna tabela typów z opisem każdego z nich znajduje się w rozdziale Typy STORAGE.

12.1 Operatory klauzuli FROM

Strumieniowe wyrażenie algebraiczne w klauzuli `FROM` może zawierać:

Operator	Składnia	Opis
Suma	`A + B`	Połączenie dwóch strumieni — patrz Sekwencjonowanie sumowania
Przeplot	`A # B`	Interleaving dwóch strumieni — patrz Sekwencjonowanie przeplotu
Przesunięcie	`A > N`	Przesuwa okno odczytu o N próbek
Okno	`A @ (k, w)`	Ruchome okno danych — patrz Ruchome okno danych
AGSE		AGSE
Agregat	`A.min` / `A.max` / `A.avg` / `A.sumc`	Redukuje wielopolowy rekord do jednej wartości — patrz Operatory agregujące

Uwaga

Operator przesunięcia `A > N` ma pokrycie w teście: `issue56_timeshift` opisanym w załączniku pt. Testy Integracyjne.

Uwaga

Propagacja wartości null przez wyrażenia SELECT ma pokrycie w teście: `issue121_null_propagation` opisanym w załączniku pt. Testy Integracyjne.

Rozdział 13

Sekwencjonowanie operacji sumowania

Do systemu napływają i są przetwarzane w nim dane. Określenie kolejności ich napływu i przetwarzania możemy opisać terminem - sekwencjonowanie. Sposób w jakim zostaną dane połączone opisywany jest przez wyrażenie algebraiczne umieszczone w klauzuli FROM. Wyrażenia te zapisane są w formie szeregu operacji algebraicznych, podlegającym ścisłym regułom. Podobne reguły poznaliśmy w trakcie nauki w szkole podstawowej – były to reguły dotyczące operacji arytmetycznych w zbiorze liczb takich jak dodawanie, mnożenie dzielenie i odejmowanie.

Na początku przeanalizujmy następujące zapytanie:

```
DECLARE a BYTE STREAM A, 1 FILE 'data1.txt'
DECLARE a BYTE STREAM B, 2 FILE 'data2.txt'
SELECT * STREAM str1 FROM A+B
```

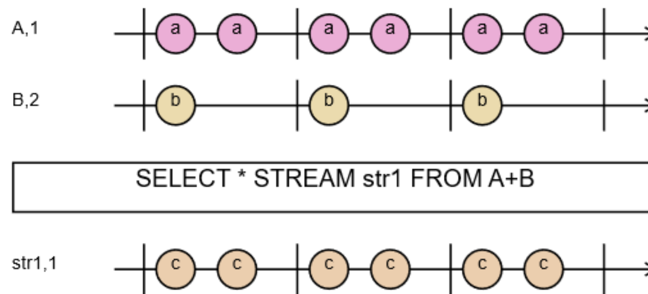
Zapytanie zapiszę w pliku qplan1.rql. Następnie wykonam następujące polecenia:

```
$ xretractor -c qplan1.rql -w 1:3 > out.txt
$ swirly out.txt -o out.svg
```

Program swirly zainstalowany został z repozytorium GitHub [6]. Program ten służy do generacji diagramów kulkowych stosowanych w wyjaśnianiu zachowania operacji asynchronicznych RxJs [7].

Modyfikację jaką zastosowałem w moim przypadku użycia to alternatywne znaczenie pionowych linii. W moim przypadku pionowe linia oddzielają jednolite interwały czasowe – prezentujące ilość cykli o które poprosiliśmy przy wywołaniu (w tym przypadku to 3 cykle). Wygenerowany obraz przedstawia Rys. 3:

W tym miejscu konieczne jest kilka słów wyjaśnienia dotyczące tego generatora oraz sposobu generacji wytycznych dla tego generatora. Wbudowałem w kompilator opcję wizualizacji realizacji sekwencji operacji. Diagramy tworzone przez program Swirly są jednym z wygodnych sposobów prezentacji zależności czasowych. Na wejściu program Swirly oczekuje pliku tekstowego z opisem diagramu. Generator symulujący wskazaną ilość cykli w argumencie i budujący plik dla Swirly został wbudowany w



Rys. 3 Schemat Kulkowy - Operacja sumy

kompilator.

Program xretractor po podaniu jako pierwszy parametr nazwy pliku z planem realizacji zapytania wymaga drugiego parametru (-w [–diagram]) – co jest wskazaniem że oczekujemy na wyjściu opisu diagramu kulkowego. Wymaganym argumentem parametru -w są dwie liczby oddzielone dwukropkiem. Pierwsza informuje czy program ma wstawić separatory czasowe na diagramie (to te pionowe linie oddzielające cykle), drugim parametrem jest ile cykli ma zostać zaprezentowane na diagramie.

Jeśli zajrzysz do wygenerowanego pliku out.txt zobaczysz następującą zawartość:

```
% Creating diagram output grid is on, cycle count:3
% Minimum interval is 1000ms
% Maximum interval is 2000ms
% Grid time is 500ms, divider:2
% Full cycle step count in grid is 4
-|a-a-|a-a-|a-a-|-
title = A,1

-|b---|b---|b---|-
title = B,2

> SELECT * STREAM str1 FROM A+B

-|c-c-|c-c-|c-c-|-
title = str1,1
```

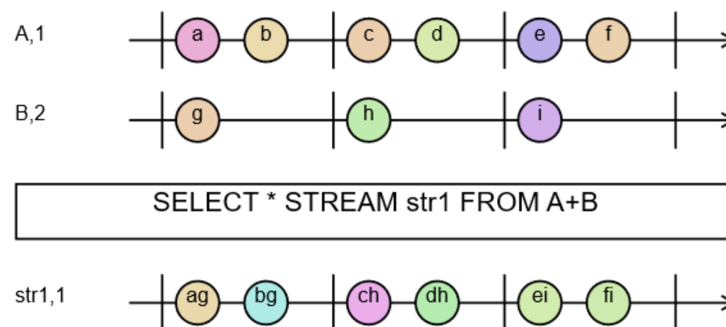
W tym pliku proszę zwrócić uwagę na dane przedstawione w komentarzach. Są to czasy wyznaczone w trakcie generowania schematu a odnoszące się do skali prezentowanej na schemacie kulkowym. Jak widać, dla naszego zapytania minimalny interwał okna to 1 sekunda, maksymalny to 2 sekundy. Siatka jaka została zidentyfikowana i wyznaczona na pół sekundy. Na schemacie każda litera lub myślnik to właśnie półsekundowy czasokres pomiędzy kolejnymi operacjami.

Wygenerowaną zawartość możemy zawartość zmienić ręcznie. Jeśli zamienimy tą zawartość w następujący sposób:

```
-|a-b-|c-d-|e-f-|-
```

```
title = A,1

-|g---|h---|i---|-
title = B,2

> SELECT * STREAM str1 FROM A+B

-|j-k-|l-m-|n-o-|-
title = str1,1
j:=ag
k:=bg
l:=ch
m:=dh
n:=ei
o:=fi
```

Wywołamy następnie ponownie program swirly zobaczymy bardziej dokładny rysunek przedstawiający sekwencję zdarzeń występujących w systemie.



Rys. 4 Schemat kulkowy - Suma, diagram zmodyfikowany

Na diagramie przedstawionym na rysunku Rys. 4 widać, które kulki zostały połączone i z których kulek powstały. Przypominam jednak że to obraz poprawiony ręcznie, dla celów tego opracowania – generator wbudowany w kompilator nie realizuje tej funkcjonalności.

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `Pattern1`, `issue167_triarg` opisanych w załączniku pt. Testy Integracyjne.

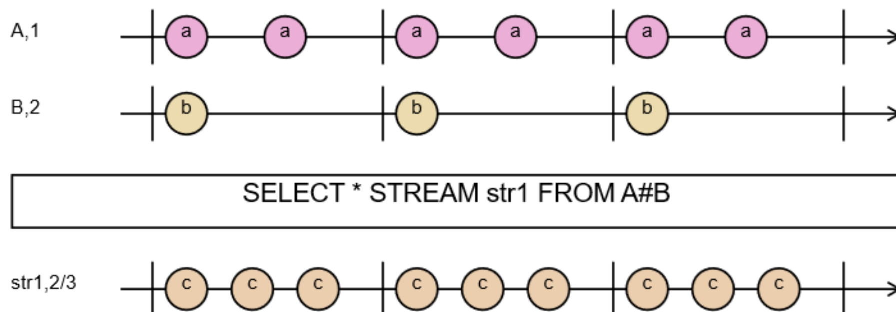
Rozdział 14

Sekwencjonowanie operacji przeplotu

Przeanalizujmy teraz operację przeplotu. Stwórzmy plik qplan2.rql o następującej treści:

```
DECLARE a BYTE STREAM A, 1 FILE 'data1.txt'
DECLARE a BYTE STREAM B, 2 FILE 'data2.txt'
SELECT * STREAM str1 FROM A#B
```

Oprócz znaku # zamiast znaku + w klauzuli from oba pliki się niczym nie różnią. Wywołajmy kompilację oraz program swirly. Plik graficzny prezentować się będzie następująco:



Rys. 5 Schemat kulkowy - operacja przeplotu

Na Rys. 5 widać zmianę. Kulki strumienia str1 zostały równomiernie uporządkowane w czasie. Zdarzenia występujące w zadeklarowanych strumieniach danych wejściowych nie uległy zmianie. Uległa natomiast zmianie zasada budowy strumienia wynikowego str1.

Jeśli zajrzymy do wygenerowanego schematu tekstowego – zobaczymy że wartości czasowe również uległy zmianie:

```
% Minimum interval is 666ms
```

```
% Maximum interval is 2000ms
% Grid time is 333ms, divider:2
% Full cycle step count in grid is 6
```

Zachęcam do dalszego eksperymentowania z tym sposobem prezentacji zdefiniowanych operacji na seriach czasowych.

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `operations`, `Pattern1` opisanych w załączniku pt. Testy Integracyjne.

Rozdział 15

Klauzula VOLATILE

Klauzula `VOLATILE` w poleceniu `SELECT` tworzy strumień przechowujący wyłącznie jeden rekord w pamięci. Na dysku pojawia się jedynie plik deskryptora `.desc` opisujący schemat danych — same dane nigdy nie są zapisywane.

15.1 Działanie

```
SELECT wyrażenie STREAM nazwa FROM źródło VOLATILE
```

Wewnętrznie kompilator ustawia typ przechowywania na `MEMORY` z pojemnością `1`:

```
if (ctx->VOLATILE()) {
    qry.policy = std::make_pair("MEMORY", 1);
}
```

Oznacza to, że:

- bufor w pamięci przechowuje zawsze tylko jeden, ostatni rekord,
- dane nie trafiają na dysk,
- deskryptor `.desc` jest tworzony — inne procesy mogą poznać schemat strumienia.

15.2 Różnica względem `STORAGE MEMORY`

Cecha	`VOLATILE`	`STORAGE MEMORY`
Pojemność bufora	zawsze 1 rekord	zależna od `RETENTION`
Klauzula `RETENTION`	ignorowana	stosowana
Deskryptor na dysku	tak	tak
Dane na dysku	nie	nie

`VOLATILE` przydaje się gdy wynik zapytania jest pobierany przez `xqry` na bieżąco i historia nie jest potrzebna — np. aktualna wartość czujnika udostępniana przez system operacyjny.

15.3 Przykład

```
DECLARE a INTEGER STREAM sensor, 0.1 FILE '/dev/sensor0'
```

```
SELECT sensor[0] * 100 STREAM scaled VOLATILE
```

Strumień `scaled` zawiera w każdej chwili jedną, aktualną wartość. Proces `xqry` może ją odczytać przez pamięć współdzieloną.

Rozdział 16

Typy STORAGE

Klauzula STORAGE w poleceniu SELECT oraz dyrektywa SUBSTRAT przyjmują jeden z następujących identyfikatorów. Każdy mapuje się na konkretną klasę akcesora danych w implementacji.

16.1 Tabela typów

Słowo kluczowe	Klasa C++	Retencja	Shadow	Przeznaczenie
DEFAULT	groupFile<posixBinaryFileWithShadow>	tak	tak	Domyślny tryb produkcyjny; plik .shadow chroni modyfikacje
DIRECT	groupFile<posixBinaryFile>	tak	nie	Retencja bez ochrony shadow
MEMORY	memoryFile	tak (RAM)	nie	Dane wyłącznie w pamięci; bufor kołowy bez zapisu na dysk
POSIX	posixBinaryFile	nie	nie	Pojedynczy plik binarny; bez retencji
POSIXSHD	posixBinaryFileWithShadow	nie	tak	Pojedynczy plik z ochroną shadow; bez retencji
GENERIC	genericBinaryFile	nie	nie	Generyczny plik binarny
DEVICE	binaryDeviceRO	nie	nie	Urządzenie binarne; tylko odczyt; pętla zależna od ONESHOT
TEXTSOURCE	textSourceRO	nie	nie	Plik tekstowy; tylko odczyt; pętla zależna od ONESHOT

Retencja — artefakty rotowane, starsze pliki usuwane automatycznie (wymaga `RETENTION` w `SELECT`).
Shadow — każda modyfikacja zapisywana jest do osobnego pliku `.shadow`; dane historyczne są chronione przed nadpisaniem.

W przypadku `MEMORY` retencja działa w pamięci jako bufor kołowy: kolejne dopisania nadpisują najstarszy slot (`index % capacity`). Dane nie są segmentowane do plików i nie trafiają na dysk.

Uwaga

Typ `MEMORY` (SUBSTRAT 'memory') ma pokrycie w testach: `issue61_tmpmem` (sekwencyjny i równoległy) opisanych w załączniku pt. Testy Integracyjne.

16.2 Kiedy używać

Wybór zależy od wymagań środowiska:

- **Środowisko produkcyjne, dane krytyczne** → `DEFAULT` (retencja + shadow)
- **Środowisko produkcyjne, dane nieistotne historycznie** → `MEMORY` (zero dysku, retencja w RAM)
- **Rozwój i debugowanie** → `DEFAULT` lub `DIRECT` (dane widoczne na dysku)
- **Odczyt z urządzenia lub pliku tekstowego** → `DEVICE` / `TEXTSOURCE` (odpowiednio)

16.3 Przykład

```
SELECT str1[0] STREAM str1 FROM core0 STORAGE MEMORY
SELECT str2[0] STREAM str2 FROM core0 RETENTION 100 STORAGE DIRECT
```

Dla substratów globalnie — dyrektywa `SUBSTRAT`:

```
SUBSTRAT 'memory'
```

Rozdział 17

Operatory agregujące i funkcje wyrażeń

17.1 Agregaty okna (MIN, MAX, AVG, SUMC)

Operatory agregujące działają na strumieniu posiadającym wiele pól — typowo strumieniu wynikowym operatora `@(k,w)` (okno danych). Redukują wszystkie pola rekordu do jednej wartości.

Składnia

```
FROM strumień.agregator
```

gdzie `agregator` to jedno z:

Słowo kluczowe	Działanie
`min` / `MIN`	minimum ze wszystkich pól rekordu
`max` / `MAX`	maksimum ze wszystkich pól rekordu
`avg` / `AVG`	średnia arytmetyczna pól rekordu
`sumc` / `SUMC`	suma wszystkich pól rekordu

Słowa kluczowe akceptowane są zarówno małymi, jak i wielkimi literami.

Interwał wyjściowy

Agregaty nie zmieniają częstotliwości strumienia — interwał wyniku jest taki sam jak źródła:

$$\Delta_{wynik} = \Delta_{strumie}$$

Przykład: średnia ruchoma

```
DECLARE val INTEGER STREAM src, 1 FILE 'data.txt'

-- okno 5-elementowe przesuwane o 1
SELECT * STREAM win5 FROM src@(1,5)

-- średnia z 5 ostatnich wartości
SELECT win5[0] STREAM ma5 FROM win5.avg
```

Strumień `ma5` zawiera w każdej chwili średnią z pięciu kolejnych próbek `src`.

Przykład: filtr sygnałowy (sumc)

Fragment z przykładu implementacji filtru sygnałowego:

```
SELECT signalRow[_] * filter[_] STREAM accRow FROM signalRow+filter
SELECT accRow[0] STREAM output FROM accRow.sumc
```

`accRow.sumc` sumuje wszystkie pola rekordu `accRow` (iloczyny próbek sygnału przez współczynniki filtru) produkując wyjście filtru FIR.

Przykład: MIN i MAX

```
DECLARE v INTEGER STREAM src, 0.1 FILE '/dev/urandom'
SELECT * STREAM win10 FROM src@(1,10)

SELECT win10[0] STREAM min10 FROM win10.min
SELECT win10[0] STREAM max10 FROM win10.max
```

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `simple_max`, `Pattern4` opisanych w załączniku pt. Testy Integracyjne.

17.2 Funkcja to_string

Funkcja `to_string` konwertuje wyrażenie liczbowe na ciąg tekstowy o zadanej szerokości. Wynik trafia do pola typu STRING w strumieniu wynikowym.

Składnia

```
to_string(wyrażenie : szerokość)
to_string(wyrażenie)
```

Parametr `szerokość` (liczba naturalna po dwukropku `:`) określa szerokość pola wyjściowego w bajtach. Pominięcie parametru daje domyślną szerokość 32 bajtów.

Info

Separator argumentów to dwukropek `:`, nie przecinek `,`. Przecinek jest separatorem listy SELECT — użycie przecinka w `to_string(x, n)` spowoduje błąd parsowania.

Przykład

```
DECLARE v INTEGER STREAM src, 1 FILE 'data.txt'
```

```
SELECT to_string(src[0]:10) STREAM labels FROM src
```

Strumień `labels` zawiera wartości `src` sformatowane jako tekst w polu 10-bajtowym.

Konkatenacja z literałem

Ciąg wynikowy można łączyć z literałem stringowym operatorem `+`:

```
SELECT to_string(src[0]:8) + '_ok' STREAM tagged FROM src
```

Rozmiar pola wynikowego: 8 (z `to_string`) + 3 (literal `_ok`) = 11 bajtów.

Zastosowanie

`to_string` przydaje się przy eksporcie do systemów przyjmujących dane tekstowe (Graphite, InfluxDB przez `xqry`) lub przy tworzeniu etykiet zdarzeń łączonych z wyjściem `DO DUMP`.

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `issue121_isnull`, `issue128_numeric_to_string`, `issue128_string_to_numeric` opisanych w załączniku pt. Testy Integracyjne.

Rozdział 18

Polecenie RULE

To polecenie to jedno z ostatnich opracowanych przeze mnie rozszerzeń systemu. Rozszerza ono funkcjonalność systemu o mechanizm alarmowania.

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `issue42_rule` opisanych w załączniku pt. Testy Integracyjne.

Składnia polecenia RULE przedstawia się następująco:

```
RULE nazwa_reguły
ON nazwa_strumienia_danych
WHEN warunek_logiczny
DO DUMP kroki_wstecz TO kroki_w_przód [RETENTION segmenty]
```

Lub w taki sposób:

```
RULE nazwa_reguły
ON nazwa_strumienia_danych
WHEN warunek_logiczny
DO SYSTEM polecenie_systemu
```

Tak zdefiniowane zdarzenia podpinają się do zdefiniowanych strumieni danych. Nazwa reguły powinna być unikalna. Strumień danych powinien zostać zdefiniowany przed pojawieniem się polecenia stworzenia reguły w pliku rql.

W obu wersjach polecenia RULE tworzona jest nazwa reguły, warunek logiczny oraz nazwa strumienia do którego proces uruchamiany poleceniem DO jest podłączany. Warunek logiczny powinien odwoływać się do zmiennych dostępnych w schemacie strumienia danych występującego po klauzuli ON.

W pierwszej wersji polecenia w której występuje klauzula DO DUMP definiujemy proces, który umożliwia zebranie danych, które napłyną w przyszłości. Jeśli pominiemy klauzulę RETENTION, zrzut nastąpi bezpośrednio do pliku z nazwą reguły poprzedzonej nazwą strumienia. Jeśli dołączymy klauzulę RETENTION, pliki będą podlegały retencji w zakresie zdefiniowanej w parametrze ‘segmenty’. Będą dołączane se-

kwencyjne numery na końcu każdego zrzutu. Zrzuty są binarne i zachowują schemat wszystkich pól źródłowego strumienia danych. Tutaj na uwagę powinno zasługiwać to, że polecenie tworzy proces w systemie, który po pojawieniu się warunku logicznego którego wartość powinna być prawdą – pobiera dane z przeszłości oraz zakłada ich napływ i rejestrację w przyszłości. Nic nie stoi na przeszkodzie aby jednak zebrać dane tylko z przeszłości lub tylko z przyszłości. Jeśli wartości kroki_* przyjmą wartości ujemne to odnosimy się do przeszłości (tzn. do danych historycznych w stosunku do momentu wystąpienia zdarzenia opisanego warunkiem logicznym)

Klauzula DO SYSTEM umożliwia wywołanie zdarzenia systemowego po zajściu w warunku logicznego opartego na zarejestrowanych danych. W ten sposób dowolne polecenie systemowe może zostać wywołane.

Przykłady deklaracji reguł w języku RQL:

```
RULE testrule1
ON str1
WHEN str1[0] > 11
DO DUMP -5 TO 5 RETENTION 100

RULE testrule2
ON str1
WHEN str1[0] = 13 OR str1[0] = 11
DO SYSTEM 'echo "systemcall"'
```

Zakładamy, że zdefiniowano uprzednio strumień str1 którego dane w postaci liczb o typie całkowitym pojawiają się co sekundę. W takim przypadku pierwsza reguła podpinając się do tego strumienia oczekuje aż dane, których wartość przekracza wartość 11. Jeśli takie zdarzenie zajdzie dokona się zrzut danych obejmujących obszar 5 sekund wstecz i 5 sekund po zajściu zdarzenia opisanego w warunku logicznym.

Druga reguła z trochę innym warunkiem logicznym wyświetli na ekranie w którym został uruchomiony proces systemu RetractorDB tekst o treści „systemcall”.

18.1 Składnia polecenia RULE

Pełna składnia polecenia RULE ma postać:

```
RULE <nazwa>
ON <strumień>
WHEN <warunek>
DO <akcja>
```

Gdzie `<akcja>` może przyjąć jedną z dwóch form:

```
SYSTEM '<polecenie_systemowe>'
DUMP [-]<krok_wstecz> TO [-]<krok_wprzód> [RETENTION <n>]
```

Ograniczenie

Reguła może być podpięta wyłącznie pod strumień zadeklarowany poleceniem SELECT (artefakt lub substrat). Podpięcie pod strumień wejściowy DECLARE jest błędem kompilacji:

```
# NIEPRAWIDŁOWE - core0 jest deklaracją, nie można podpiąć reguły
RULE r1 ON core0 WHEN core0[0] > 10 DO SYSTEM 'echo alarm'
```

Warunek WHEN

Warunek to wyrażenie logiczne ewaluowane do wartości prawda/fałsz po każdej nowej próbce strumienia.

Operatory porównania: =, !=, <, >, <=, >=. Operatory logiczne: OR, AND, NOT. Przykłady:

```
WHEN str1[0] > 100
WHEN str1[0] = 0 OR str1[0] = 255
WHEN str1[0] >= 10 AND str1[0] <= 90
WHEN NOT str1[0] = 0
```

18.2 Akcja DO SYSTEM

Akcja DO SYSTEM wykonuje podane polecenie powłoki (przez wywołanie system(3)) w momencie spełnienia warunku. RetractorDB loguje kod wyjścia polecenia — niezerowy kod jest raportowany jako błąd w logu.

```
RULE alert1
ON wyniki
WHEN wyniki[0] > 1000
DO SYSTEM 'curl -s http://monitoring/alert'
```

W poleceniu można użyć dowolnego programu dostępnego w PATH: skryptów powłoki, programów Pythona, wywołań REST, wysyłki powiadomień, etc.

18.3 Akcja DO DUMP

Akcja DO DUMP zapisuje okno próbek strumienia do pliku binarnego w momencie spełnienia warunku. Pozwala zachować kontekst zdarzenia: dane przed jego wystąpieniem i dane po nim.

```
RULE zdarzenie
ON wyniki
WHEN wyniki[0] > 500
DO DUMP -10 TO 5
```

Parametry zakresu:

Parametr	Znaczenie
ujemny `step_back` (np. -10)	dołącz 10 próbek **historycznych** sprzed zdarzenia
0 jako `step_back`	zacznij zrzut od chwili zdarzenia
dodatni `step_back` (np. 2)	opóźnij start zrzutu o 2 próbki po zdarzeniu
`step_forward` (np. 5)	zbierz łącznie `step_forward - step_back` próbek

Całkowita liczba zrzucanych rekordów: `abs(step_forward - step_back)`. Przykład: `DUMP -5 TO 5` → 10 rekordów (5 historycznych + 5 kolejnych). `DUMP 0 TO 1` → 1 rekord (bieżąca próbka).

Zakres `step_back` musi być mniejszy lub równy `step_forward`. Wartość `step_back` może być ujemna (historia) lub nieujemna (opóźnienie). Obie wartości ujemne nie są obsługiwane.

Pliki zrzutu

Pliki są tworzone w katalogu konfigurowanym przez dyrektywę `STORAGE`. Konwencja nazewnictwa:

```
<strumień>_<nazwa_reguły>_dump.tmp          # bez RETENTION
<strumień>_<nazwa_reguły>_dump_<n>.tmp      # z RETENTION (n = 0..N-1)
```

Format pliku to surowe dane binarne zgodne z deskryptorem strumienia (bez nagłówka). Do odczytu pliku można użyć narzędzia `xtrdb`.

Opcja RETENTION

Parametr `RETENTION <n>` ogranicza liczbę przechowywanych zrzutów — stary plik jest nadpisywany przez nowy (bufor cykliczny). Bez `RETENTION` każde wyzwolenie nadpisuje jeden plik `_dump.tmp`.

```
RULE zdarzenie
ON wyniki
WHEN wyniki[0] > 500
DO DUMP -10 TO 5 RETENTION 20
```

Powyższy przykład przechowuje 20 ostatnich zrzutów w plikach `wyniki_zdarzenie_dump_0.tmp` … `wyniki_zdarzenie_dump_19.tmp`.

18.4 Wiele reguł dla jednego strumienia

Do jednego strumienia można przypiąć dowolną liczbę reguł różnych typów:

```
RULE alert_wysoki   ON pomiary WHEN pomiary[0] > 900 DO SYSTEM 'notify-send "Przekroczono prog"
RULE alert_niski    ON pomiary WHEN pomiary[0] < 10  DO SYSTEM 'notify-send "Zbyt niska wartosc
RULE zapis_anomalii ON pomiary WHEN pomiary[0] > 900 DO DUMP -20 TO 10 RETENTION 5
```

Wszystkie reguły danego strumienia są ewaluowane przy każdej nowej próbce.

Rozdział 19

Konstrukcja mechanizmu

Przez alarmowanie rozumiemy proces przetwarzania danych bieżących i bieżącego reagowania systemu w razie rozpoznania zaistniałego zjawiska przez system. Aby alarmowanie mogło funkcjonować, musza w systemie istnieć mechanizmy wspierające ten proces. W systemie RetractorDB opracowałem model alarmowania oparty na deklaracji reguł związanych z obserwacją strumieni danych. Reguły te zawierają operacje matematyczne umożliwiające analizę warunków logicznych i uruchomienie zewnętrznych procesów lub realizację zrzutu danych w wybranym oknie czasowym.

> **Uwaga**
>
> Opisana funkcjonalność ma pokrycie w testach: `issue42_rule` opisanych w załączniku pt. Testy Integracyjne.

Prezentacji składni polecenia RULE na stronie 24 wspomina o tej funkcjonalności. W tym rozdziale chciałbym przybliżyć zasady działania tego rozwiązania.

Budując przykład przedstawiający zasadę działania alarmowania stwórzmy następujący plik zapytania – query.rql:

```
DECLARE a UINT STREAM core0, 1 FILE 'datafile1.txt'
SELECT str4[0] STREAM str4 FROM core0>1

RULE regulation1
ON str4
WHEN str4[0] = 20 or str4[0] = 23
DO SYSTEM 'echo "test"'
```

W pliku datafile1.txt znajdują się liczby w postaci tekstowej od 20 do 28.

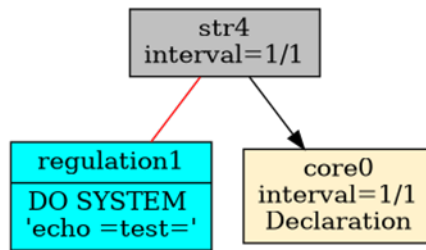
```
$ seq 20 28 > datafile1.txt
```

Powyższe 3 polecenia deklarują efemeryczne źródło danych, jedno polecenie przetwarzania danych poprzez przesunięcie w czasie o jedną próbkę w czasie. Oraz regułę alarmowania. Wykonanie następującego polecenia:

```
$ xretractor -c query.rql -d -u -p -i > out.dot &&
```

```
dot -Tpng out.dot -o out.png
```

Wyświetlając plik out.png zobaczymy na ekranie coś zbliżonego (Rys. 6):



Rys. 6 Zależność obiektów w przypadku użycia alarmowania

Obraz zaprezentuje jaka zachodzi zależność pomiędzy procesami odpowiedzialnymi za artefakty, alarmowanie oraz efemerydy. Równie dobrze powinno się udać podłączyć proces odpowiedzialny za alarmowanie do substratu.

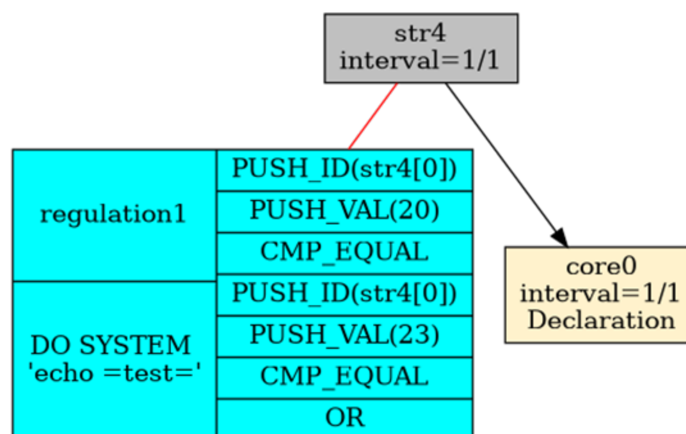
Obiekty alarmowania przedstawiane są w kolorze błękitnym i połączone z obiektami, które monitorują za pomocą czerwonych, nieskierowanych linii.

Obiektów odpowiedzialnych za alarmowanie można podłączyć więcej niż jeden. Można podać więcej poleceń RULE skojarzonych z danym poleceniem tworzącym strumień danych.

Jeśli przyjrzymy się bliżej zobaczymy, że z procesem odpowiedzialnym za alarmowanie jest uruchamiany warunkiem. Następującym poleceniem możemy podejrzeć co tam właściwie się dzieje:

```
$ xretractor -c query.rql -d -u -p > out.dot &&
dot -Tpng out.dot -o out.png
```

Plik wyjściowy prezentuje się w następujący sposób (Rys. 7):



Rys. 7 Kod odpowiedzialny za warunek uruchomienia alarmowania.

Ten warunek musi zostać w ostatecznej formie wyliczony do wyrażenie reprezentującego prawdę lub fałsz.

Rozdział 20

Warunek logiczny w RULE

Klauzula `WHEN` polecenia `RULE` przyjmuje wyrażenie logiczne, które jest ewaluowane na każdym nowym rekordzie wskazanego strumienia. Jeśli wyrażenie zwraca prawdę — uruchamiany jest proces zdefiniowany w klauzuli `DO`.

20.1 Operatory porównania

Operator	Znaczenie
`=`	równy
`!=`	różny
`>`	większy
`<`	mniejszy
`>=`	większy lub równy
`<=`	mniejszy lub równy

20.2 Spójniki logiczne

Operator	Znaczenie
`AND`	koniunkcja — oba warunki muszą być spełnione
`OR`	alternatywa — wystarczy jeden warunek
`NOT`	negacja — warunek musi być niespełniony

20.3 Struktura wyrażenia

Warunek buduje się z pól schematu strumienia wskazanego w klauzuli `ON`. Pola identyfikowane są tak samo jak w `SELECT` — przez nazwę strumienia z indeksem:

```
WHEN strumień[indeks] operator wartość
```

Złożone warunki łączymy spójnikami:

```
WHEN strumień[0] > 10 AND strumień[1] != 0
WHEN strumień[0] = 5 OR strumień[0] = 7
WHEN NOT strumień[0] < 0
```

20.4 Przykłady

```
RULE alarm_wysoki
ON pomiary
WHEN pomiary[0] > 100 OR pomiary[0] < -100
DO DUMP -10 TO 10 RETENTION 50

RULE sygnalizacja
ON status
WHEN status[0] = 1 AND status[1] != 0
DO SYSTEM 'systemctl restart sensor-reader'

RULE jednorazowy
ON dane
WHEN NOT dane[0] = 0
DO DUMP -5 TO 0
```

20.5 Dostęp do pól

Warunek odwołuje się do pól strumienia wskazanego w `ON`. Indeks pola odpowiada pozycji w schemacie tego strumienia — tak samo jak w klauzuli `SELECT`. Aliasowanie działa identycznie jak opisano w rozdziale Aliasowanie.

Rozdział 21

Przykład alarmowania

W oknie terminala uruchamiamy proces xretractor uruchamiając przedstawiony z początku rozdziału plik query.rql

```
$ xretractor query.rql
test
test
test
…
```

W drugim oknie terminala proponuję uruchomić polecenie:

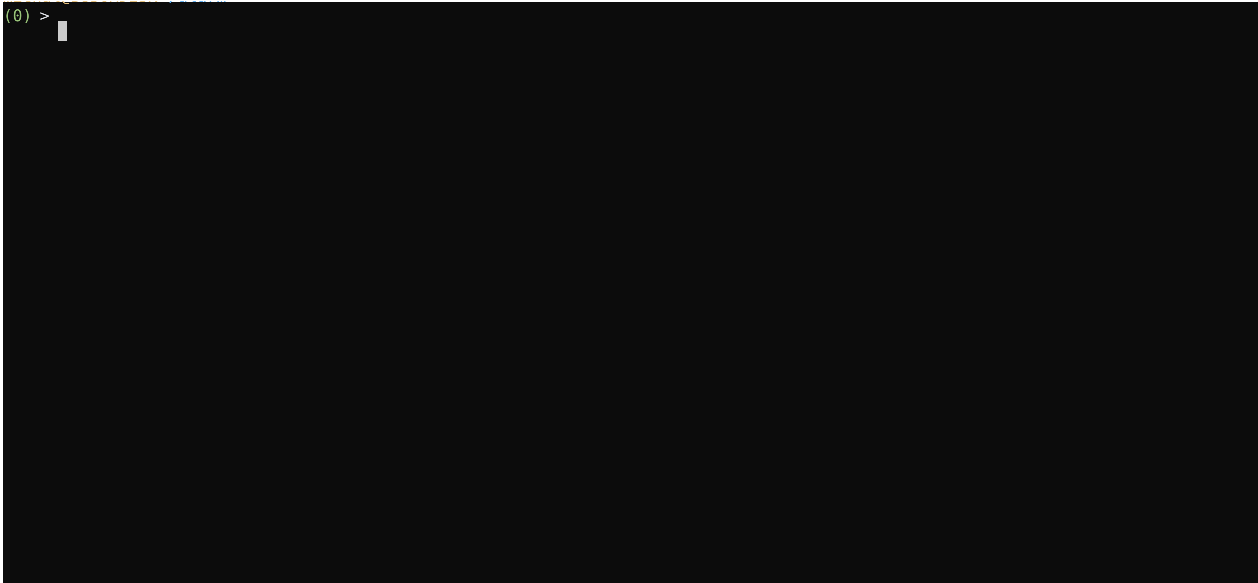
```
$ xqry -s str4
27
28
20
21
22
23
24
25
26
27
```

Oba okna proponuję ustawić obok siebie. Zobaczymy, że pojawianie się wartości 20 i 23 powoduje uruchomienie akcji po stronie serwera wyświetlającej napis test. Należy pamiętać, że w systemie może pojawić się dowolne polecenie systemowe lub wywołanie dowolnego programu w zależności o tego co umieścimy w deklaracji DO SYSTEM.

Zapis sesji (Rys. 8):

21.1 Przykład 2: zapis kontekstu zdarzenia (DO DUMP)

Akcja `DO DUMP` pozwala utrwalić okno próbek z otoczenia zdarzenia — dane sprzed i po jego wystąpieniu. Jest to przydatne gdy chcemy zachować kontekst anomalii do późniejszej analizy.



Rys. 8 Zapis sesji przykładu alarmowania

Tworzymy plik `query.rql`:

```
STORAGE 'temp'

DECLARE a INTEGER STREAM core0, 1 FILE 'datafile1.txt'
SELECT str1[0] STREAM str1 FROM core0

RULE zapis_anomalii
ON str1
WHEN str1[0] > 24
DO DUMP -3 TO 3
```

Dane wejściowe — liczby od 20 do 28:

```
$ seq 20 28 > datafile1.txt
```

Uruchamiamy xretractor:

```
$ xretractor query.rql
```

Gdy wartość strumienia `str1` przekroczy 24, reguła wyzwoli zapis 6 rekordów (3 historyczne + 3 kolejne) do pliku binarnego `temp/str1_zapis_anomalii_dump.tmp`.

Odczyt pliku zrzutu

Plik zrzutu nie zawiera nagłówka `.desc` — przy otwieraniu w `xtrdb` należy podać schemat ręcznie:

```
$ xtrdb
> storage temp
> open str1_zapis_anomalii_dump { INTEGER a }
> size
```

```
> list 6
> quit
```

21.2 Przykład 3: rotacja zrzutów (DO DUMP z RETENTION)

Bez `RETENTION` każde kolejne wyzwolenie reguły nadpisuje ten sam plik. Gdy zdarzenia powtarzają się, użyj `RETENTION N` aby zachować ostatnie N zrzutów w osobnych plikach.

```
STORAGE 'temp'

DECLARE a INTEGER STREAM core0, 1 FILE 'datafile1.txt'
SELECT str1[0] STREAM str1 FROM core0

RULE zapis_anomalii
ON str1
WHEN str1[0] > 24
DO DUMP -3 TO 3 RETENTION 5
```

Każde wyzwolenie tworzy kolejny plik (rotacja cykliczna):

```
temp/str1_zapis_anomalii_dump_0.tmp
temp/str1_zapis_anomalii_dump_1.tmp
temp/str1_zapis_anomalii_dump_2.tmp
temp/str1_zapis_anomalii_dump_3.tmp
temp/str1_zapis_anomalii_dump_4.tmp
```

Po przekroczeniu pojemności (`RETENTION 5`) najstarszy plik jest nadpisywany przez nowy.

21.3 Przykład 4: wiele reguł na jednym strumieniu

Do jednego strumienia można przypiąć dowolną liczbę reguł. Poniższy przykład łączy obie akcje — powiadomienie systemowe i zapis kontekstu:

```
STORAGE 'temp'

DECLARE a INTEGER STREAM core0, 1 FILE 'datafile1.txt'
SELECT str1[0] STREAM str1 FROM core0

RULE prog_dolny
ON str1
WHEN str1[0] < 21
DO SYSTEM 'echo "ALARM: wartosc ponizej progu dolnego" >> alarm.log'

RULE prog_gorny
ON str1
WHEN str1[0] > 26
```

```
DO SYSTEM 'echo "ALARM: wartosc powyzej progu gornego" >> alarm.log'

RULE zapis_kontekstu
ON str1
WHEN str1[0] > 26
DO DUMP -5 TO 5 RETENTION 10
```

Reguły `prog_gorny` i `zapis_kontekstu` reagują na ten sam warunek niezależnie — przekroczenie progu górnego jednocześnie zapisuje log i utrwala okno danych. Reguła `prog_dolny` obsługuje osobno próg dolny.

Wszystkie trzy reguły są ewaluowane przy każdej nowej próbce strumienia `str1`.

Rozdział 22

Dyrektywy konfiguracyjne

Na chwilę obecną opracowałem trzy dyrektywy konfiguracyjne.

- STORAGE
- SUBSTRAT
- ROTATION

Po każdej z tych dyrektyw występuje ciąg tekstowy ujęty w cudzysłowy lub apostrofy. Przykład zastosowania obu dyrektywy konfiguracyjnych przedstawia się następująco:

```
STORAGE 'temp_folder'
SUBSTRAT 'memory'
ROTATION 'rotation_counter.txt'
```

Storage służy do wskazania w którym katalogu systemowym powinny powstawać wszystkie pliki wynikowe. Bez tej dyrektywy, domyślnie pliki tworzone przez system umieszczane są w bieżącym katalogu w którym został uruchomiony główny proces systemu RetractorDB.

Substraty to zapytania oraz ich efekty, które powstają w wyniku rozkładu poleceń systemu przez kompilator na podstawie wyrażeń algebry szeregów czasowych. Są to zapytania, które widać w planie realizacji zapytań ale nie są one specyfikowane bezpośrednio w pliku .rql. Wynikają one z implementacji procesu konstrukcji planu realizacji zapytań.

Domyślnie takie zapytania materializują dane na dysku w postaci nieskończonych plików. Tego typu zachowanie może być pożądane w przypadku prowadzenia procesu rozwoju oprogramowania, w przypadku umieszczenia systemu w środowisku produkcyjnym lepiej substraty przechowywać w tymczasowych obszarach pamięci.

Możliwe opcje w poleceniu SUBSTRAT to: memory, default, direct, posix, posixshd, generic, device, textsource. Pełny opis każdego typu — klasa C++, obsługa retencji i shadow — znajdziesz w rozdziale Typy STORAGE.

Ostatnia dyrektywa - Rotation to dyrektywa wskazująca na odmienny tryb kończenia pracy przez system. Domyślnie po kompilacji wszystkie pliki wytworzone przez system pozostają w stanie w jakim system zarejestrował dane. Po kolejnym wywołaniu

polecenia systemowego – wszystkie pliki artefaktów i substratów są usuwane. Użycie dyrektywy Rotation w pliku rql z deklaracją zapytań sprawi że system utworzy plik wymieniony w parametrze dyrektywy i umieści tam licznik zwiększany z każdym uruchomieniem systemu. Plikom z artefaktami i substratami po każdym zakończeniu pracy systemu zostanie zmieniona nazwa – dostaną rozszerzenie .old oraz numer wynikający ze wzrastającego licznika. Ten proces nazywamy rotacją artefaktów.

Rozdział 23

Architektura systemu

Konstrukcja systemu przetwarzania danych to rozdział stricte techniczny. Przedstawię tutaj jak system został zaprojektowany, zbudowany gdzie i jak obecnie rozmieszczone są jego funkcjonalności.

System RetractorDB został zaimplementowany w języku C++ pod kontrolą systemu Linux. Kod źródłowy podlega procesowi ciągłej integracji i testowania na platformie GitHub wspieranej przez CircleCI. Kod uruchamiany i rozwijany jest lokalnie na platformie Linux WSL2. Porzuciłem rozwój i implementację systemu pod kontrolą systemu Windows. W początkowej fazie utrzymywałem taką opcję i być może w przyszłości do niej powrócę. Jednak utrzymanie zbyt wielu platform rozwojowych znacząco opóźnia proces szybkiego prototypowania i rozwoju systemu. Nadal zachowuję i utrzymuję funkcjonalność systemu na platformie Linux ARM. Kod kompiluję i testuje się pod kontrolą maszyn opartych na architekturze ARM i x86-64 pracujących w zasobach CircleCI. Raspberry PI to jedna z docelowych platform produkcyjnych systemu RetractorDB przewidziana dla potrzeb Edge IoT.

Kompilacja kodu systemu odbywa się ze wsparciem managera pakietów Conan [8]. Jeśli chcemy poznać jak zbudowany jest toolchain budujący kod systemu możemy zajrzeć do pliku /.circleci/config.yml zawierający procedurę budowy i uruchamiania systemu w środowisku kontenerów lub maszyn firmy CircleCI. W plikach `/docker/ci/Dockerfile` oraz `/docker/ci/DockerConan.txt` znajdują się instrukcje w jaki sposób obraz kontenera budującego system z prekonfigurowanymi dependencjami. Analiza tych plików wskaże co jest potrzebne i jak należy zainstalować w swoim systemie aby źródła systemu skompilować lokalnie u siebie.

23.1 Przegląd poruszonych w rozdziale tematów

Rozdział zbudowany jest warstwowo — od widoku ogólnego do szczegółów implementacyjnych.

Perspektywa ogólna

System jako trójka współpracujących programów: `xretractor` jako singleton realizujący plan zapytań, `xqry` jako wieloinstancyjny klient danych bieżących, `xtrdb` jako narzędzie inspekcji plików binarnych. Komunikacja między procesami `xretractor` i `xqry` realizowana jest przez pamięć współdzieloną (Boost IPC). Na schemacie Rys. 9 widać granicę odpowiedzialności każdego z komponentów.

Przepływ danych i sterowania

Które ścieżki danych są zawsze aktywne (napływ danych → xretractor → artefakty), a które opcjonalne lub diagnostyczne. Opisano też mechanizm graceful shutdown — xretractor reaguje na sygnały `SIGINT`, `SIGTERM` i `SIGHUP` kończąc bieżący cykl bez ryzyka uszkodzenia plików.

Artefakty, substraty i efemerydy

Kluczowy podział taksonomiczny systemu. Każdy typ strumienia ma inne przeznaczenie i inną strategię składowania: artefakty materializowane na dysku jako trwały wynik, substraty to strumienie pośrednie niezbędne podczas obliczeń, efemerydy — ulotne źródła danych, których nie można ani nie warto przechowywać.

Format zapisu danych

Czteroplikowa struktura artefaktu: plik binarny z danymi (stałej długości rekordy, brak nagłówka), deskryptor `.desc` opisujący schemat rekordu w gramatyce ANTLR4, plik metadanych `.meta` z indeksem wartości null i przerw w transmisji (kodowanie RLE), opcjonalny plik cienia `.shadow` do niedestruktywnej modyfikacji historycznych rekordów. Deskryptor określa strategię składowania przez pole `TYPE`.

Kompilacja i budowa planu

Proces przekształcania pliku `.rql` w gotowy plan realizacji zapytania. Flaga `-c` uruchamia tryb kompilacji bez wykonania; połączona z `-d -f -s` generuje wyjście DOT, które `graphviz` zamienia w graf przepływu danych. Graf pokazuje dwie domeny: stos wyrażeń arytmetycznych (PUSH, ADD, itp.) i algebrę strumieniową. Opisano pełny zestaw flag trybu kompilacji i wykonania.

Przetwarzanie i dystrybucja danych

Kompletny walkthrough: od przygotowania pliku danych przez uruchomienie `xretractor`, przez podgląd statystyk strumieniowania (`xqry -d`), po wizualizację na żywo w gnuplot (`xqry -s str1 -p 50,50 | gnuplot`) i transmisję przez sieć za pomocą `nc`. Przykład łączy dwa źródła — plik tekstowy i `/dev/urandom` — ilustrując jak operator `+` w klauzuli FROM realizuje algebraiczne łączenie strumieni.

Analiza artefaktów

Narzędzie `xtrdb` — interaktywny inspektor plików binarnych wzorowany na stylu dbase. Polecenia `.open`, `.desc`, `.list`, `.rlist` i `.meta` pozwalają przeglądać zawartość artefaktów bez znajomości formatu binarnego. Narzędzie służy też do weryfikacji deterministyczności: te same dane wejściowe powinny zawsze dawać identyczne wyniki.

Trzy polecenia wystarczające do uruchomienia kompletnego przepływu:

```
xretractor -c query.rql          # weryfikacja poprawności pliku zapytań
xretractor query.rql             # uruchomienie przetwarzania
xqry -s <strumień>               # odczyt danych bieżących
```

Czwarty element — `xtrdb` — pojawia się przy diagnostyce i testowaniu, nie w typowym przepływie produkcyjnym.

Rozdział 24

Perspektywa ogólna

System zbudowany jest w oparciu o 3 programy dostępne jako polecenia systemowe. Pierwszym jest kompilator oraz system realizujący plany zapytań. Drugim jest klient dostępu do danych bieżących, trzecim jest program umożliwiający dostęp zrzutów binarnych. Ich nazwy to kolejno:

- xretractor
- xqry
- xtrdb

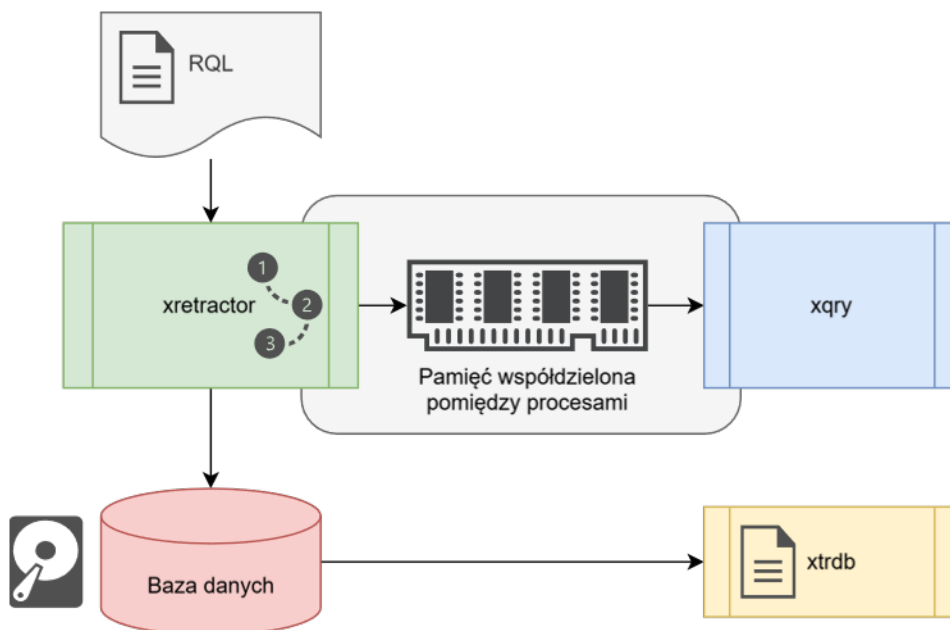
Program xretractor tworzy główny proces systemu RetractorDB. Program xqry tworzy procesy komunikujące się z systemem RetractorDB. Komunikacja zachodzi za pomocą wspólnego obszaru w pamięci. Program xtrdb służy do analizy danych i metadanych zapisywanych w plikach bazy danych.

Poniżej przedstawiona jest na Rys. 9 schematycznie architektura systemu RetractorDB. Uwzględniono wszystkie istniejące aktualnie komponenty. Obszary ujęte w prostokątach z nagłówkami wypełnionymi poleceniami systemowymi odpowiadają istniejącym komponentom. Obszar zapisu artefaktów to symboliczna reprezentacja systemu plików.

Na Rys. 9 widzimy procesy realizowane przez programy xretractor, xtrdb oraz xqry. Na rysunku schematycznie przedstawiono sposób komunikacji pomiędzy procesami w systemie RetractorDB. Rysunek pokazuje części wspólne opracowanych narzędzi.

Proces xretractor komunikuje się z procesami xqry poprzez obszar pamięci współdzielonej. W tej pamięci dla każdego procesu xqry tworzona jest przez xretractor kolejka danych. Dane są odbierane na bieżąco przez procesy xqry. Zadaniem procesów xqry jest wysyłka danych dalej do innych systemów lub procesów. Jeśli proces xqry ginie lub jest kończony xretractor zarządzający obszarem wspólnym zwalnia obszar dedykowany we wspólnym obszarze.

Oprócz kierowania danych do wysyłki poprzez pamięć współdzieloną, system RetractorDB zapisuje dane do tzw. Obszaru zapisu artefaktów. Aktualnie jest to katalog do którego zapisywane są na bieżąco efekty procesu przetwarzania strumieni danych w oparciu o plany realizacji zapytań realizowane w systemie RetractorDB.



Rys. 9 Schemat przepływu danych pomiędzy procesami RetractorDB

Ostrzeżenie

Przedstawiona na rysunku Baza danych to nie jest Relacyjna baza danych. Przez bazę danych na przedstawionym rysunku rozumiemy zbiór plików binarnych lub tekstowych, którymi zarządza RetractorDB. Dane pobierane są z urządzeń i zapisywane w rotujących lub nie plikach binarnych lub tekstowych. Dostęp do tych danych realizowany jest za pomocą narzędzia xtrdb lub w trakcie działania systemu przez proces xqry.

Plik z zapytaniami i dyrektywami RQL podaje się jako wymagany, pierwszy argument polecenia uruchamiającego system. Takie zachowanie prawdopodobnie ulegnie w przyszłości zmianie – system docelowo powinien uruchomić się jako usługa i oczekiwać od operatora pliku z dyrektywami. Na chwilę obecną system jednak uruchamiamy z wkładem inicjującym. Jak chcemy coś dołożyć w trakcie pracy, odsyłam do rozdziału pt. Zapytania Ad hoc.

Rozdział 25

Przepływ danych i sterowania

Dane i sterowanie w systemie RetractorDB tworzą kilka potencjalnych sposobów użycia komponentów systemu. Na Rys. 10 przedstawiono schematycznie przepływ danych pomiędzy procesami systemu RetractorDB, procesami systemu Linux oraz danymi źródłowymi i rezultatami pracy poszczególnych procesów.

Najgrubsze linie przedstawiają przepływ, który występuje zawsze w procesie przetwarzania regularnych serii czasowych. Proces xretractor aby wystartować na chwilę obecną potrzebuje pliku .rql ze sekwencją zapytań. Po przeprowadzeniu kompilacji, proces xretractor buduje drzewo planu zapytania i rozpoczyna proces przetwarzania napływających danych i tworzenia plików binarnych zawierających artefakty.

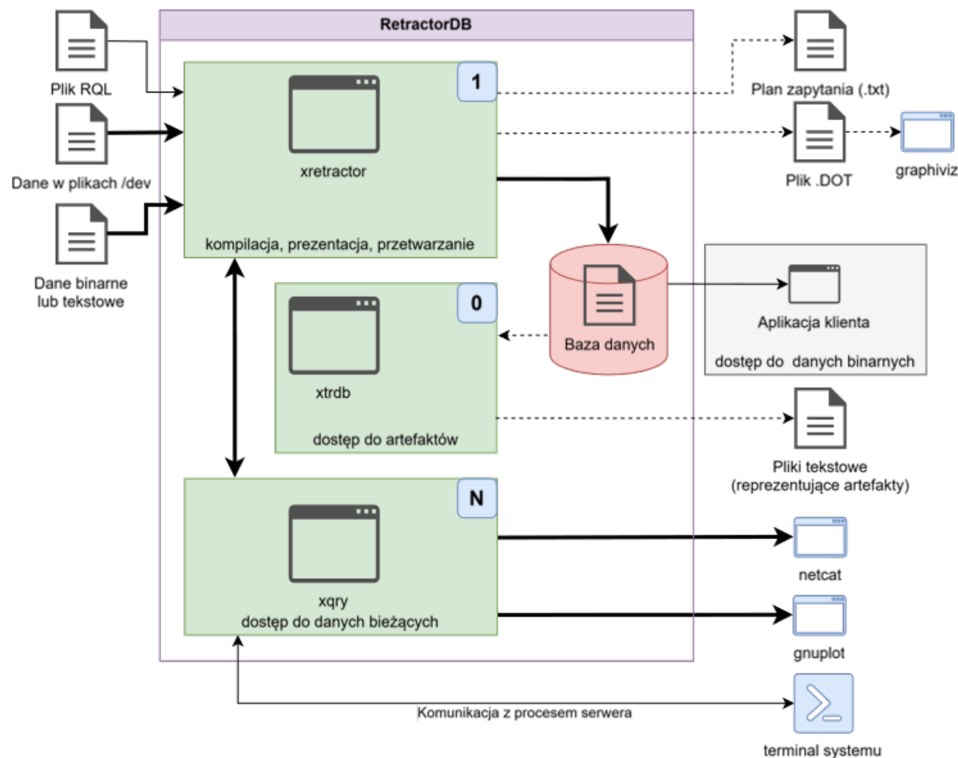
Uwaga

Opisana funkcjonalność ma pokrycie w teście: `consistency` opisanym w załączniku pt. Testy Integracyjne.

Aby móc sterować procesem xretractor po wystartowaniu używamy procesu xqry. Za jego pomocą możemy zatrzymać proces xretractor, pobrać statystyki lub zażądać dostępu do danych bieżących.

Reszta strzałek prezentuje przepływy danych zależne od prowadzonego z użyciem RetractorDB procesu. Strzałki przerywane są typowo przeznaczone do celów diagnostycznych.

Każdy z procesów na schemacie został oznaczony dodatkowo liczbą utrzymywanych ciągłych procesów w systemie. Oznaczenie „1” przy procesie xretractor oznacza że ten program będzie pilnował aby tylko jedna instancja tego procesu funkcjonowała w systemie. Próba uruchomienia kolejnej zakończy się błędem i komunikatem przy uruchomieniu. Program xtrdb nie utrzymuje żadnych ciągłych i nieskończonych procesów. Czyta dane, przetwarza, zwraca wyniki i kończy pracę. Oferuje też opcję pracy w trybie interaktywnym. Proces xqry oznaczony został jako „N”. W ten sposób chciałem wyrazić że procesów xqry można wywoływać więcej niż jeden. Jest to typowy scenariusz pracy z systemem RetractorDB. Klientów komunikujących się z procesorem planów realizacji zapytań z definicji występuje kilka.



Rys. 10. Przepływ danych i sterowania

25.1 Zatrzymanie xretractor

Proces xretractor obsługuje sygnały systemowe i kończy pracę w kontrolowany sposób po otrzymaniu:

Sygnał	Polecenie	Znaczenie
`SIGINT`	Ctrl+C w terminalu	przerwanie interaktywne
`SIGTERM`	`kill <pid>`	standardowe zakończenie procesu
`SIGHUP`	`kill -HUP <pid>`	zakończenie przy zamknięciu terminala

Wszystkie trzy sygnały powodują ten sam efekt: graceful shutdown — pętla przetwarzania kończy bieżący cykl i zatrzymuje się. Pozwala to bezpiecznie zamknąć xretractor działającego jako usługa bez ryzyka uszkodzenia plików artefaktów.

Zatrzymanie przez xqry

Obok sygnałów systemowych xretractor można zatrzymać programowo — za pomocą polecenia:

```
xqry --kill
```

Jak przebiega zamknięcie krok po kroku

1. xqry wysyła żądanie „kill”

Proces xqry buduje komunikat IPC i umieszcza go w kolejce komunikatów `RetractorQueryQueue` — wspólnym kanale łączącym wszystkich klientów z xretractor. Wiadomość zawiera identyfikator procesu xqry (PID) i polecenie `kill`.

2. xretractor odbiera polecenie i ustawia flagę zatrzymania

Wątek komunikacyjny xretractor (`commandProcessorLoop`) stale nasłuchuje na `RetractorQueryQueue`. Po odebraniu komunikatu `kill` ustawia atomową zmienną `iTimeLimitCnt` na wartość `stop_now`. Ten sam mechanizm jest używany przez obsługę sygnałów systemowych — niezależnie od źródła (sygnał `SIGINT`/`SIGTERM`/`SIGHUP` lub polecenie xqry `--kill`) efekt jest identyczny.

3. Główna pętla przetwarzania wykrywa flagę i kończy bieżący cykl

Pętla główna sprawdza `iTimeLimitCnt` przy każdej iteracji. Gdy wykryje wartość `stop_now`, kończy bieżący cykl i wychodzi z pętli — bez przerywania w połowie obliczeń. Zapewnia to integralność zapisywanych artefaktów.

4. xretractor powiadamia wszystkich podłączonych klientów (broadcast OOB)

Po wyjściu z pętli xretractor wywołuje `boradcastOutOfBussiness()`. Funkcja ta przegląda wewnętrzną mapę `id2StreamName_Relation`, która zawiera wpis dla każdego procesu xqry subskrybującego strumień danych (każde wywołanie `xqry --select` rejestruje się w tej mapie przez polecenie `show`). Dla każdego zarejestrowanego klienta xretractor wysyła do jego dedykowanej kolejki komunikat specjalny o wartości `OUT_OF_BUSSINESS`.

5. Każdy klient xqry odbiera sygnał zakończenia i kończy działanie

Każdy działający proces xqry ma własną, indywidualną kolejkę komunikatów o nazwie `brcdbr<PID>`. Po odebraniu komunikatu `OUT_OF_BUSSINESS` xqry ustawia wewnętrzną flagę `done` i kończy działanie w kontrolowany sposób — niezależnie od tego, ile danych zdążył odebrać.

6. Sprzątanie zasobów IPC

Na zakończenie xretractor usuwa wszystkie współdzielone zasoby IPC: segment pamięci współdzielonej `RetractorShmemMap`, kolejkę poleceń `RetractorQueryQueue`, mutex `RetractorMapMutex` oraz indywidualne kolejki wszystkich klientów.

Co się dzieje przy wielu procesach xqry

RetractorDB jest zaprojektowany do pracy z wieloma równoległymi klientami. Gdy w systemie działają jednocześnie — powiedzmy — trzy procesy xqry subskrybujące różne strumienie, a jeden z nich wywoła `xqry --kill`:

- xretractor przetworzy żądanie kill **jednorazowo**, niezależnie od tego, który klient je wysłał,
- mechanizm `boradcastOutOfBussiness()` roześle komunikat `OUT_OF_BUSSINESS` do **wszystkich** zarejestrowanych klientów jednocześnie,
- każdy z trzech procesów xqry otrzyma sygnał zakończenia i zakończy działanie samodzielnie,

- klienci, którzy nie subskrybowali żadnego strumienia (np. xqry wywołany tylko z `--dir` lub `--hello`), nie są wpisani do mapy i nie muszą być powiadamiani — te polecenia kończą działanie natychmiast po udzieleniu odpowiedzi.

Warto zwrócić uwagę, że xqry wykrywa również nieaktywność serwera: jeżeli przez 10 sekund nie napłyną żadne dane, klient sam się wyłącza z ostrzeżeniem w logu. Jest to zabezpieczenie na wypadek nagłej awarii xretractor bez możliwości rozesłania komunikatu OOB.

Rozdział 26

Artefakty, Substraty, Efemerydy

Z racji faktu, że system przeznaczony jest do pracy ciągłej i teoretycznie otrzymywane wyniki bez prowadzenia procesu retencji danych zapełniłby każdy nośnik danych wprowadzamy dodatkowe definicje związane z charakterem przetwarzanych danych.

Przedstawiając opis Rys. 10 wspominano o artefaktach. Jest to jedna z definicji wymagających wyjaśnienia.

Uwaga

Definicja (Artefakt): Przez artefakty rozumiemy dane przetwarzane w systemie w postaci strumieni, które docelowo zostają zmaterializowane jako utrwalony wynik i efekt przetwarzania innych danych.

Ciągłe serie czasowe możemy czytać z urządzeń, następnie je przetwarzać – redukować lub dopasowywać rozmiar danych w czasie i wymiarze. Ale z reguły pewne dane powinny zostać zapisane. Czy te dane potem będą podlegać retencji – jest sprawą drugorzędną. Takie dane, które stanowią efekt i oczekiwaną odpowiedź systemu będziemy nazywać artefaktami. Czymś co oczekujemy i materializujemy dla potrzeb użytkownika końcowego.

Uwaga

Definicja (Substrat): Substraty to obiekty pośrednie. W wyniku przetwarzania serii czasowych mogą powstać strumienie danych, które są ulotne. Potrzebne jedynie do i w trakcie przetwarzania.

Ich rozmiar może być znaczący biorąc pod uwagę jak daleko cofamy się wstecz w przypadku np. konieczności zrzutów danych monitorowania z przeszłości. Jednak ich istnienie jest bez znaczenia w aspekcie pożądanych wyników działania systemu. Takie strumienie danych nazywamy substratami. Pojawiają się w wyniku działania systemu, nie występują z reguły jawnie w zapytaniach – ale wynikają z procesu przetwarzania serii czasowych, jednak ich wyniki są niezbędne do realizacji zadania.

Uwaga

Definicja (Efemeryd): Efemerydy to obiekty, w oparciu o które tworzymy źródłowe strumienie danych, danych których nie można zmagazynować. Są to z reguły dane ulotne, efemeryczne.

System czyta np. liczby przypadkowe z odpowiednią częstotliwością i to właśnie źródło danych dostarcza danych ulotnych. Nie można ich zwrócić, przechowywanie ich z reguły mija się z celem – należy je przekazać do dalszego przetwarzania w celu wytworzenia artefaktów lub substratów a następnie zniszczyć i pobrać, nowe aktualne.

Rozdział 27

Format zapisu danych

W systemie przetwarzane są serie czasowe w trzech postaciach: **artefaktów**, **efemerydów** i **substratów**. Każdy typ ma inne przeznaczenie i inną strategię przechowywania.

Substraty i Artefakty - formalnie niczym nie różnią się w systemie. Jedyna różnica to fakt, że substraty zostały wygenerowane w oparciu o równiania algebry strumieni danych i nie zostały zapisane bezpośrednio w ciągu poleceń dla kompilatora. Jeśli zadeklarujemy strumień Artefaktu, który pokryje postać substratu - substrat zostanie zredukowany. Efemerydy to strumienie, które powstały za pomocą polecenia Declare - zawierają wartości które istnieją tylko przez chwilkę.

Typy akcesorów składowania

Uwaga

Opisana funkcjonalność ma pokrycie w teście: `txtsrc` opisanym w załączniku pt. Testy Integracyjne.

Pole `TYPE` w deskryptorze (lub dyrektywa `STORAGE` w RQL) wybiera implementację `FileInterface`:

Typ (`TYPE_PROFILE`)	Klasa implementacji	Zastosowanie
`DEFAULT`	`groupFile<posixBinaryFileWithShadow>`	Artefakty domyślne — plik danych + plik cienia, z retencją
`DIRECT`	`groupFile<posixBinaryFile>`	Zapis bezpośredni bez cienia, z retencją
`POSIX`	`posixBinaryFile`	Surowy zapis POSIX bez cienia
`POSIXSHD`	`posixBinaryFileWithShadow`	POSIX z plikiem cienia
`MEMORY`	`memoryFile`	Składowanie wyłącznie w RAM (efemerydy)

Typ (`TYPE_PROFILE`)	Klasa implementacji	Zastosowanie
`GENERIC`	`genericBinaryFile`	Ogólny akcesor binarny
`DEVICE`	`binaryDeviceRO`	Zewnętrzne urządzenie binarnych danych wejściowych (tylko odczyt)
`TEXTSOURCE`	`textSourceRO`	Tekstowe źródło danych wejściowych (tylko odczyt)

27.1 Zestaw plików artefaktu i substratu

Artefakty i substraty zapisywane na dysk mogą być skojarzone z maksymalnie czterema plikami:

Plik	Rozszerzenie	Cel
Plik danych binarnych	*(nazwa strumienia)*	Główny strumień rekordów — append-only
Plik deskryptora	`.desc`	Schemat rekordu (pola, typy, rozmiary, typ składowania)
Plik metadanych	`.meta`	Indeks wartości null i przerw w transmisji (RLE)
Plik cienia	`.shadow`	Modyfikacje rekordów bez nadpisywania danych oryginalnych

Rys. 11. Zestaw plików artefaktu i ich powiązania

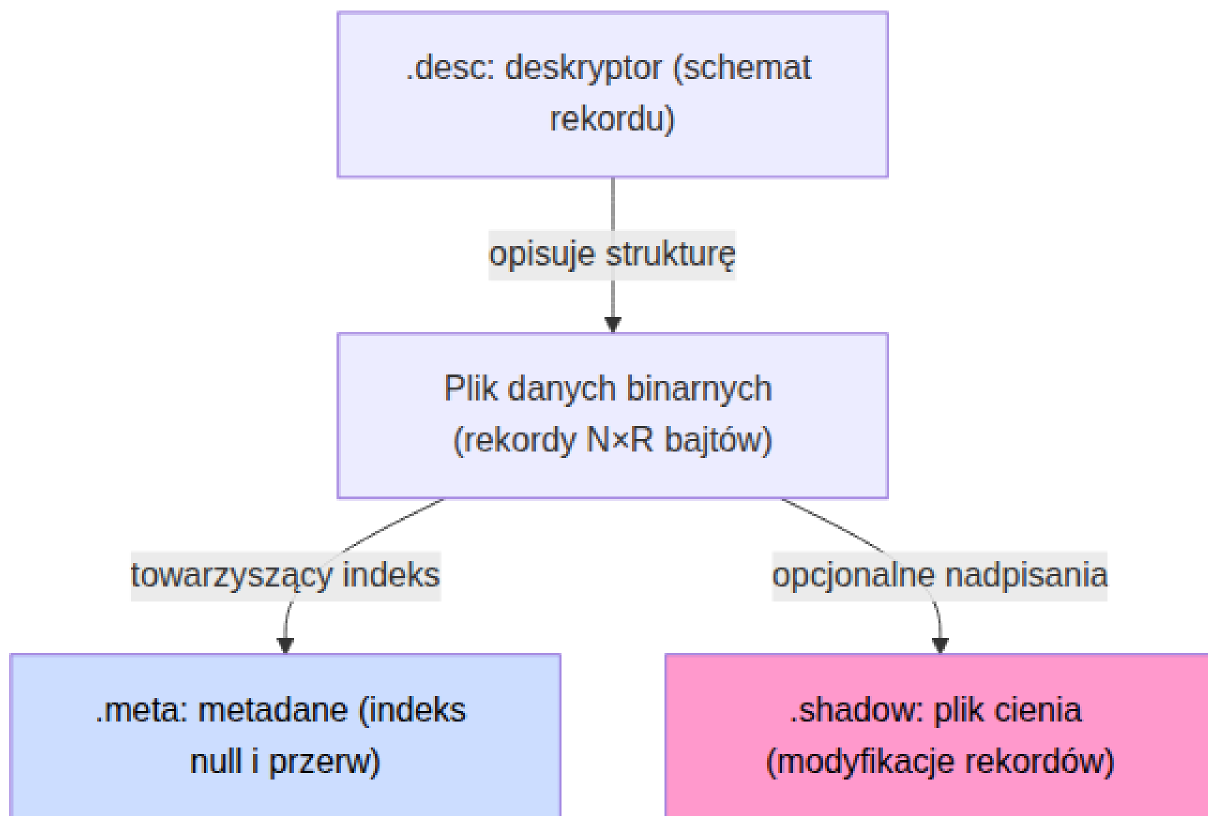
Diagram przedstawia statyczną relację między plikami artefaktu: `.desc` definiuje strukturę rekordu, `.meta` indeksuje null i przerwy, a `.shadow` przechowuje opcjonalne nadpisania rekordów.

Plik cienia i plik metadanych są opcjonalne. Przy ciągłym napływie danych bez przerw i bez modyfikacji wystarczy sam plik danych binarnych i deskryptor.

Efemerydy **nie posiadają żadnych plików na dysku** — istnieją wyłącznie w pamięci operacyjnej procesu i znikają po jego zakończeniu.

27.2 Rozdziały

- Pliki artefaktu — deskryptor, dane binarne, metadane, plik cienia i relacje między nimi
- Mechanizm rotacji plików — dyrektywa `ROTATION`, cykl życia plików, przykłady sesji



Diagram

- Narzędzie inspekcji `xtrdb -s` — mapa składowania, sekcje raportu, przykłady
- Podsumowanie — uzasadnienie przyjętej struktury, porównanie podejść

Rozdział 28

Pliki

Rozdział opisuje pięć plików tworzących kompletny zestaw artefaktu lub substratu: deskryptor schematu (`.desc`), główny plik danych binarnych, indeks metadanych (`.meta`), plik cienia danych (`.shadow`) i plik cienia indeksu (`.meta.shadow`). Dla każdego pliku przedstawiono format binarny, semantykę pól oraz reguły zapisu i odczytu. Rozdział obejmuje też klasę `metaDataStream` — mechanizm kompresji RLE, obsługę przerw w transmisji, interfejs aktualizacji i persystencję po restarcie. Sekcja końcowa pokazuje relacje między wszystkimi plikami na poziomie operacji `append`, `update` i `read`.

Zakres rozdziału **nie obejmuje** mechanizmu rotacji plików między sesjami (→ Rotacja) ani narzędzia inspekcji `xtrdb -s` (→ Narzędzie inspekcji).

28.1 Plik deskryptora (.desc)

Plik `.desc` opisuje strukturę rekordu. Jest parsowany przez gramatykę ANTLR4 (`DESC.g4`) i może zawierać pola danych, metainformację o typie składowania oraz politykę retencji.

Składnia

```
{ <polecenie>* }
```

Każde polecenie to jedno z poniższych:

```
BYTE     nazwa [N]          # tablica N bajtów (domyślnie N=1)
INTEGER  nazwa [N]          # 32-bitowe liczby całkowite ze znakiem
UINT     nazwa [N]          # 32-bitowe bez znaku
FLOAT    nazwa [N]          # 32-bitowe zmiennoprzecinkowe (IEEE 754)
DOUBLE   nazwa [N]          # 64-bitowe zmiennoprzecinkowe
RATIONAL nazwa [N]          # para int64: licznik i mianownik
STRING   nazwa [rozmiar]    # ciąg znaków o stałej długości
REF      "ścieżka/plik"     # referencja do zewnętrznego pliku deskryptora
TYPE     identyfikator      # typ składowania (DEFAULT, MEMORY, POSIXSHD, …)
```

```
RETENTION pojemność segment # retencja cykliczna na dysku
RETMEMORY pojemność         # retencja cykliczna w pamięci
```

Przykłady plików `.desc`

Artefakt domyślny — dwa pola numeryczne, składowanie `DEFAULT` (plik danych + plik cienia):

```
{
  INTEGER  ts
  FLOAT    value
  TYPE     DEFAULT
}
```

Efemeryd — strumień ulotny wyłącznie w RAM:

```
{
  DOUBLE   x
  DOUBLE   y
  TYPE     MEMORY
}
```

Substrat z retencją — cykliczny bufor ostatnich 1000 rekordów na dysku (10 segmentów po 100):

```
{
  INTEGER  ts
  FLOAT    a
  FLOAT    b
  TYPE     DEFAULT
  RETENTION 1000 100
}
```

Deklaracja źródła binarnego (`DECLARE` w RQL generuje ten schemat):

```
{
  INTEGER  a
  FLOAT    b
  TYPE     DEVICE
  REF      "sensor/data.bin"
}
```

Rozmiary typów pól

Typ	Rozmiar pojedynczej wartości
`BYTE`	1 B
`INTEGER`	4 B
`UINT`	4 B
`FLOAT`	4 B
`DOUBLE`	8 B

Typ	Rozmiar pojedynczej wartości
`RATIONAL`	16 B (dwa int64)
`STRING`	N B (deklarowany rozmiar)

Dla pól tablicowych `nazwa[N]` całkowity rozmiar = rozmiar_typu × N. Pola `TYPE`, `REF`, `RETENTION` i `RETMEMORY` nie zajmują miejsca w rekordzie — są metadanymi deskryptora.

Rozmiar rekordu `R` = suma rozmiarów wszystkich pól danych.

Pole TYPE a strategia składowania

Pole `TYPE` w deskryptorze bezpośrednio wyznacza, który akcesor (`FileInterface`) zostanie użyty przez `storage::initializeAccessor()`. Brak pola `TYPE` jest równoznaczny z `DEFAULT`. Wartość jest nieczuła na wielkość liter (`MEMORY` = `memory`).

28.2 Plik danych binarnych

Plik danych to sekwencja rekordów o stałej długości, zapisywanych jeden po drugim bez żadnego nagłówka. Rozmiar pojedynczego rekordu `R` wyznaczany jest przez deskryptor jako suma bajtów wszystkich pól.

Offset w pliku	Zawartość	Rozmiar
0	Rekord 0	R bajtów
R	Rekord 1	R bajtów
2R	Rekord 2	R bajtów
…	…	…
(N-1) × R	Rekord N-1	R bajtów

Każdy rekord zawiera upakowane wartości pól w kolejności zdefiniowanej przez deskryptor:

Offset w rekordzie	Pole	Rozmiar
0	pole_0	len_0 bajtów
len_0	pole_1	len_1 bajtów
len_0 + len_1	…	…
len_0 + len_1 + … + len_n	pole_n	len_n bajtów

Operacja **append** (dodanie nowego rekordu) dopisuje dane na koniec pliku. Operacja **update** (modyfikacja istniejącego rekordu) — jeśli istnieje plik cienia — trafia do pliku cienia, a nie do pliku głównego.

Przykład

```
DECLARE a INTEGER, b FLOAT STREAM str1, 0.1 FILE 'data.dat'
```

Rozmiar rekordu: INTEGER (4 B) + FLOAT (4 B) = **8 bajtów**. Po 5 sekundach napływu danych (10 Hz) plik `data.dat` ma rozmiar $5 \times 10 \times 8$ = **400 bajtów**.

28.3 Plik metadanych (.meta)

Plik `.meta` to indeks wartości null i przerw w transmisji. Przechowuje informację o tym, które pola rekordów mają wartość null i gdzie wystąpiły przerwy — bez duplikowania samych danych.

Format pliku

Pozycja	Zawartość	Rozmiar
Nagłówek	`creationTimeNs` (int64)	8 bajtów
Wpis RLE 0	`gapFlag | count | bitsetSize | bitset`	zmienny
Wpis RLE 1	`gapFlag | count | bitsetSize | bitset`	zmienny
…	…	…
Wpis RLE k	wpis bieżący (w pamięci)	zmienny

Format wpisu RLE

Każdy wpis opisuje ciąg kolejnych rekordów z identycznym wzorcem null:

Pole	Rozmiar	Opis
`gapFlag`	1 B	0 = normalny rekord, 1 = przerwa
`recordCount`	8 B (size_t)	liczba rekordów w ciągu
`bitsetSize`	8 B (size_t)	liczba pól (N)
`bitset`	$\lceil N/8 \rceil$ B	bit i = pole i ma wartość null

Kompresja RLE

Kolejne rekordy z tym samym wzorcem null są scalane w jeden wpis przez zwiększenie `recordCount`. Nowy wpis tworzony jest dopiero gdy wzorzec się zmienia.

10 rekordów, 2 pola, bez null:

Wpis	isGap	count	bitset
wpis 0	F	10	[F,F]

Null w polu 1 od rekordu 5:

Wpis	isGap	count	bitset
wpis 0	F	5	[F,F]
wpis 1	F	5	[F,T]

Przerwa w transmisji po rekordzie 3:

Wpis	isGap	count	bitset
wpis 0	F	3	[F,F]
wpis 1	T	7	[T,T]
wpis 2	F	...	[F,F]

Marker przerwy w transmisji (gap)

Przerwa w transmisji (np. wyłączenie systemu, zanik sygnału) rejestrowana jest jako wpis z `isGap=true` i wszystkimi bitami null ustawionymi na `true`. Parametr `count` przechowuje długość przerwy w jednostkach interwału strumienia. Sam plik danych binarnych nie zawiera żadnych dodatkowych rekordów dla przerwy — informacja żyje wyłącznie w pliku `.meta`.

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `issue113_meta_internal`, `issue113_meta_autocreate` opisanych w załączniku pt. Testy Integracyjne.

Klasa metaDataStream

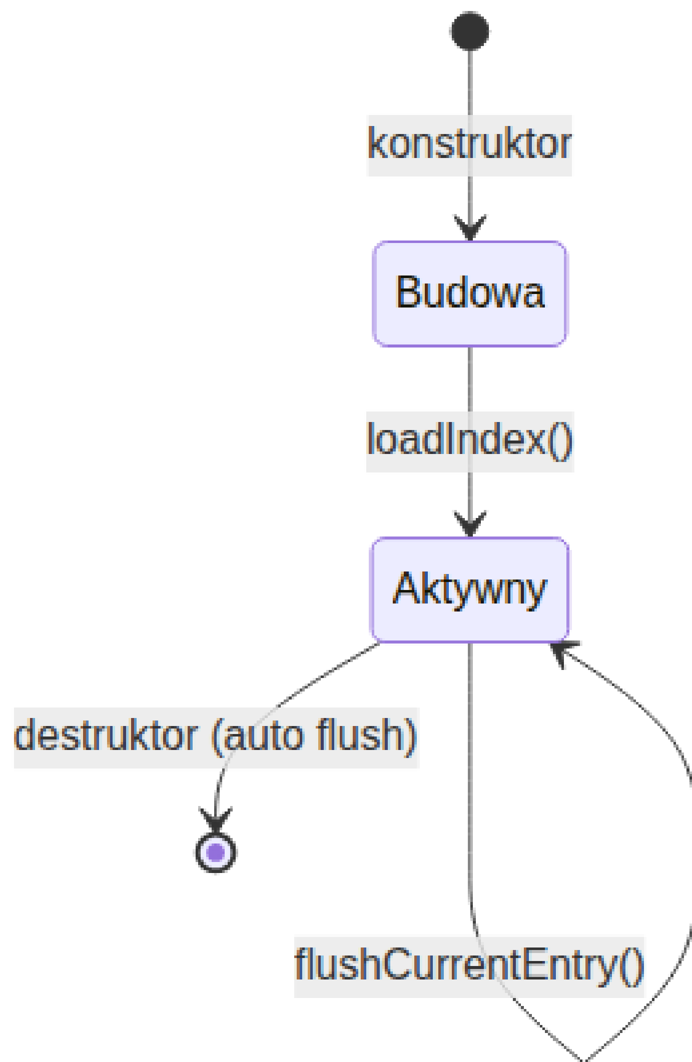
Plikiem `.meta` zarządza klasa `rdb::metaDataStream`. Hermetyzuje ona trzy obszary odpowiedzialności:

1. **Agregację RLE w pamięci** — buforuje bieżący segment (ostatnią serię rekordów z identycznym wzorcem null) w polu `currentEntry_`, nie zapisując go do pliku przy każdym rekordzie.
2. **Trwałość danych** — wyłącznie zakończone segmenty (gdy wzorzec się zmienia lub gdy nastąpi jawne wywołanie `flushCurrentEntry()`) trafiają do pliku jako wpisy zatwierdzone (*committed*).
3. **Indeks zapytań** — udostępnia interfejs do odpytywania wzorca null dla dowolnego rekordu oraz wykrywania przerw w transmisji.

Klasa przechowuje dwa stany:

Stan	Lokalizacja	Opis
Zatwierdzone segmenty	plik `.meta` na dysku	wszystkie zakończone przebiegi RLE
Segment bieżący (`currentEntry_`)	pamięć operacyjna	aktualnie akumulowany przebieg (jeszcze niezapisany lub do nadpisania)

Cykl życia obiektu



Diagram

Konstruktor (`metaDataStream(descriptor, path)`): - Inicjalizuje pusty `currentEntry_` na podstawie liczby pól deskryptora. - Wywołuje `loadIndex()` — jeżeli plik istnieje, wczytuje wszystkie zatwierdzone segmenty, wyznacza `committedRecordCount_`, a ostatni niegapowy segment przenosi z powrotem do `currentEntry_` (umożliwia kontynuację serii RLE po restarcie). - Jeżeli plik nie istnieje, tworzy go i zapisuje nagłówek (znacznik czasu utworzenia strumienia).

Destruktor automatycznie wywołuje `flushCurrentEntry()`, gwarantując, że bieżący bufor trafi na dysk nawet gdy program zakończy pracę w normalnym trybie.

Interfejs aktualizacji

Klasa wyróżnia trzy scenariusze zmiany stanu metadanych:

`onRecordAppended(nullBitset)`

Wywoływany przez `storage` po każdym dołączeniu nowego rekordu do pliku danych.

```
wzorzec identyczny z currentEntry_?
  TAK → zwiększ currentEntry_.recordCount (akumulacja RLE, brak I/O)
  NIE → flushCurrentEntry() (poprzedni segment na dysk)
          ustaw currentEntry_ = {nullBitset, count=1}
```

Operacja I/O następuje **wyłącznie przy zmianie wzorca** — dla serii identycznych rekordów koszt to jedna inkrementacja licznika w pamięci.

`onRecordModified(index, nullBitset)`

Wywoływany przez `storage` przy aktualizacji istniejącego rekordu. Zachowanie zależy od trybu pracy:

Tryb normalny (brak pliku cienia danych): lokalizuje rekord w segmentach RLE i rozbija segment na maksymalnie trzy części: przed modyfikowanym rekordem, sam rekord, za nim.

```
rekord w currentEntry_ (pamięć)?
  TAK → splitSegment() w pamięci, nowe fragmenty dołączone do pliku
  NIE → wczytaj plik, splitSegment(), przepisz plik (rewriteFile)
```

Przykład rozbicia segmentu `[allNull × 5]` przy modyfikacji rekordu 2:

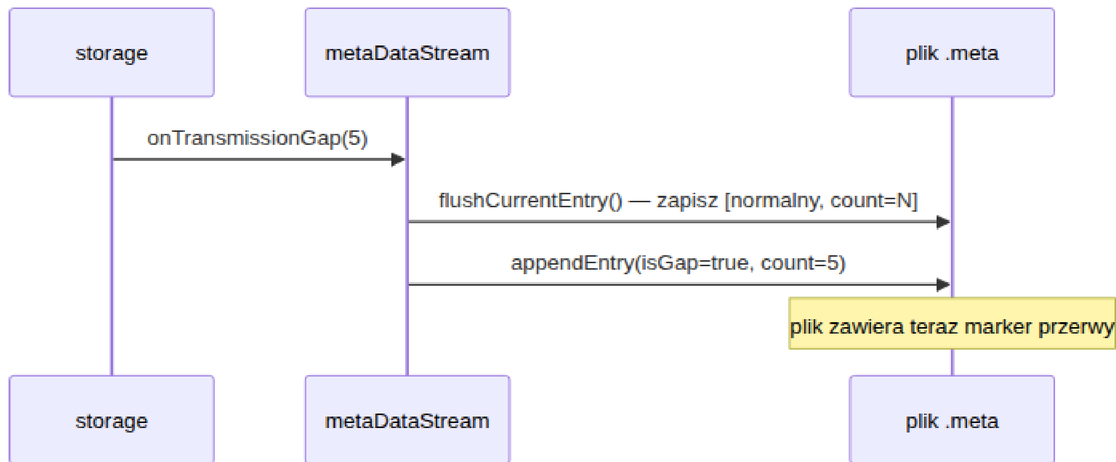
```
Przed:  [allNull × 5]
Po:     [allNull × 2] [allPresent × 1] [allNull × 2]
```

Tryb cienia (`shadowMode_ = true`, aktywowany przez `setShadowMode(true)`): zamiast modyfikować główny indeks, dopisuje jedno nadpisanie wzorca null do pliku `.meta.shadow`. Główny indeks `.meta` pozostaje nienaruszone i spójne z głównym plikiem danych.

```
shadowMode_?
  TAK → appendShadowOverride(index, nullBitset) → wpis w .meta.shadow
  NIE → applyModificationToMainIndex(index, nullBitset) → splitSegment()
```

`onTransmissionGap(duration)`

Rejestruje przerwę w transmisji o podanej długości (w jednostkach interwału strumienia). Najpierw zatwierdza bieżący segment (`flushCurrentEntry()`), następnie dołącza do pliku wpis z `isGap=true`.



Diagram

Mechanizm bezpieczeństwa: `flushCurrentEntry()` i nadpisywanie (tailDirty_)

Klasa `storage` wywołuje `flushCurrentEntry()` po **każdym** wywołaniu `write()`, aby zagwarantować przeżycie awarii procesu. Naiwna implementacja dopisywałaby nowy wpis do pliku przy każdym flushu — powodując wzrost pliku proporcjonalny do liczby rekordów, nawet bez zmian wzorca null.

Rozwiązanie: mechanizm **lazy overwrite** oznaczany flagą `tailDirty_`.

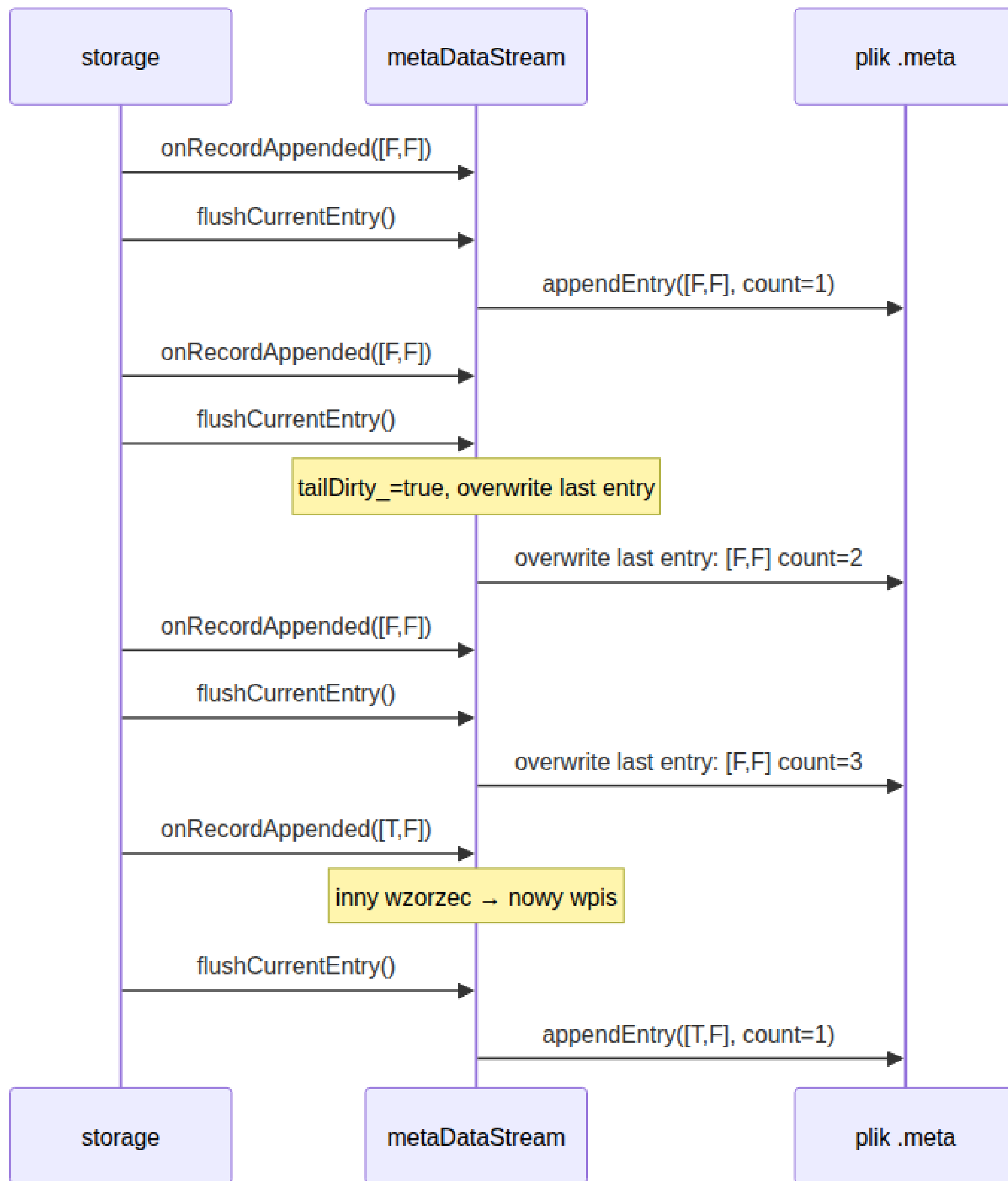
```
flushCurrentEntry() → zapis [wzorzec, count=2] na dysk
onRecordAppended(ten sam wzorzec):
    currentEntry_.count = 2 (przywrócony z dysku)
    tailDirty_ = true        ← następny flush nadpisze, nie doda
    currentEntry_.count++    → count = 3
flushCurrentEntry() → seek na ostatni wpis, overwrite [wzorzec, count=3]
    (rozmiar pliku bez zmian)
```

Diagram sekwencji dla typowego wzorca `storage` (append + flush po każdym rekordzie):

Dzięki temu plik `.meta` rośnie wyłącznie przy **zmianie wzorca null** — nie przy każdym rekordzie. Przy ciągłym napływie jednorodnych danych plik ma stały rozmiar niezależnie od liczby rekordów.

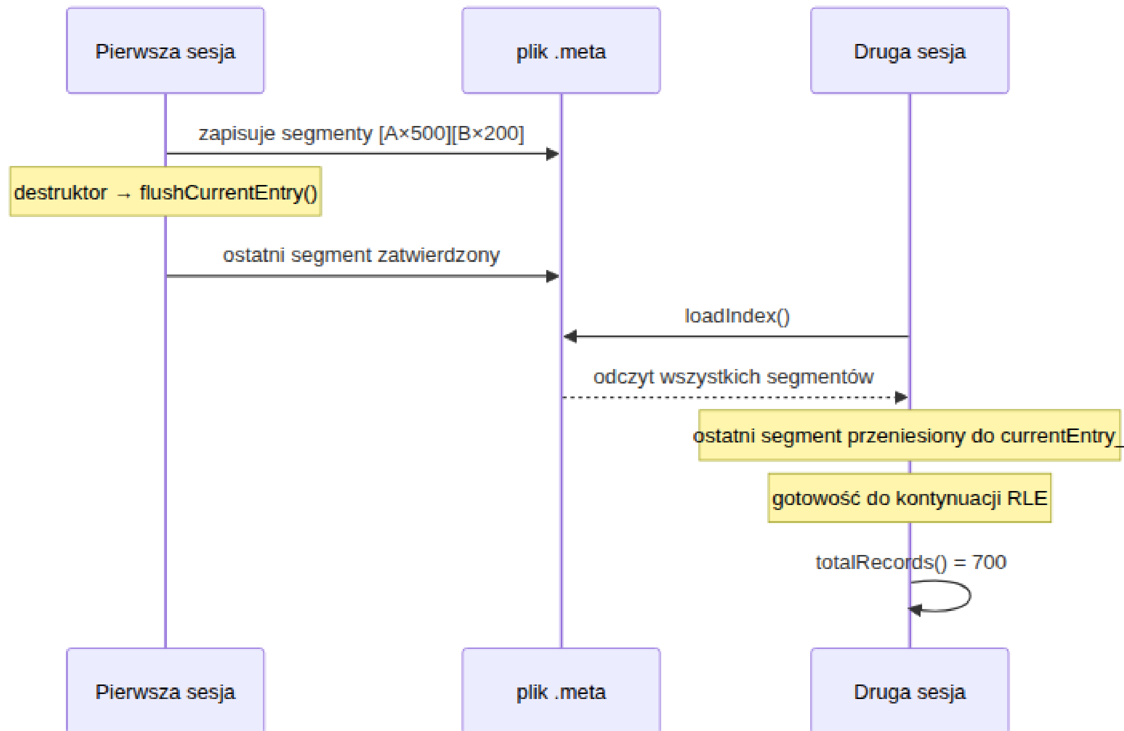
Persystencja i odtwarzanie stanu

Po restarcie procesu nowy obiekt `metaDataStream` wczytuje plik przez `loadIndex()`:



Diagram

1. Odczytuje nagłówek — znacznik czasu (`creationTimeNs`), przechowywany jako `int64` nanosekund od epoki.
2. Wczytuje wszystkie zatwierdzone wpisy z pliku.
3. Jeżeli ostatni wpis **nie jest gap-em** — przenosi go z powrotem do `currentEntry_` i usuwa z pliku (umożliwia kontynuację RLE po restarcie bez duplikacji).
4. Wyznacza `committedRecordCount_` jako sumę `recordCount` wszystkich niegalowych wpisów pozostałych w pliku.



Diagram

Interfejs zapytań

Metoda	Opis
`getNullBitset(i)`	Zwraca wzorzec null dla rekordu `i`. W trybie cienia najpierw sprawdza nadpisania w `shadowOverrides_` (od końca — ostatnie wygrywa), a dopiero przy braku wpisu sięga do głównego indeksu.
`isGapBefore(i)`	Zwraca `true`, jeżeli bezpośrednio przed rekordem `i` w indeksie RLE znajduje się wpis `isGap=true`. Rekord 0 nigdy nie ma przerwy przed sobą.

Metoda	Opis
segments()	Zwraca wszystkie segmenty RLE: zatwierdzone (z dysku) oraz bieżący (z pamięci), jeżeli jest niepusty. Nie obejmuje nadpisań z .meta.shadow. Służy do inspekcji i testów.
totalRecords()	Suma rekordów we wszystkich segmentach (committed + pending).
isEmpty()	Skrót: totalRecords() == 0.
rotate(percounter)	Rotuje plik indeksu: przemianowuje bieżący plik .meta na .meta.old<N>, tworzy nowy pusty plik. Wywoływana przez storage::detectStartupState() po wykryciu rotacji pliku danych (plik danych pusty, indeks niepusty). Gdy percounter < 0, plik nie jest przemianowywany — wykonywany jest tylko reset indeksu.
reset()	Czyści indeks w miejscu: zeruje liczniki, przepisuje plik z samym nagłówkiem bez zmiany jego nazwy. Wywołuje też discardShadow(). Wywoływany przez storage przy czyszczeniu bez zachowania historii (np. po purge()).

Interfejs cienia indeksu

Zestaw metod zarządzających plikiem .meta.shadow. Wywoływane przez storage::attachStorage() i powiązane operacje na pliku cienia danych.

Metoda	Opis
setShadowMode(enabled)	Włącza lub wyłącza tryb cienia. Przy enabled=true wywołuje loadShadow() — wczytuje istniejące nadpisania z pliku .meta.shadow.

Metoda	Opis
`mergeShadow()`	Scala nadpisania z cienia do głównego indeksu (wywołuje `applyModificationToMainIndex()` dla każdego nadpisania w kolejności zapisu — ostatnie wygrywa), a następnie usuwa plik `.meta.shadow`. Odpowiednik `merge()` dla pliku cienia danych.
`discardShadow()`	Czyści listę nadpisań w pamięci i usuwa plik `.meta.shadow`. Wywoływany przy odrzuceniu cienia danych (purge, reset, rotacja).

Przykład użycia — typowy scenariusz produkcyjny

```
storage.write(rec0)           → onRecordAppended([F,F,F]) + flushCurrentEntry()
storage.write(rec1)           → onRecordAppended([F,F,F]) + flushCurrentEntry()
storage.write(rec2_val_null)  → onRecordAppended([T,F,F]) + flushCurrentEntry()
storage.write(rec3)           → onRecordAppended([F,F,F]) + flushCurrentEntry()

Plik .meta po powyższych operacjach (4 flushe, 2 segmenty):
  [isGap=F, count=2, bitset=[F,F,F]]   ← wpis 0
  [isGap=F, count=1, bitset=[T,F,F]]   ← wpis 1  (rec2)
  [isGap=F, count=1, bitset=[F,F,F]]   ← wpis 2  (rec3, bieżący w pamięci)

getNullBitset(2) → [T,F,F]   (pole 0 rekordu 2 jest null)
isGapBefore(2)  → false
totalRecords()  → 4
```

28.4 Plik cienia (.shadow)

Plik cienia umożliwia modyfikację zarejestrowanych rekordów bez niszczenia danych oryginalnych. Usunięcie pliku `.shadow` przywraca oryginalny stan danych.

Format wpisu

Pole	Rozmiar	Opis
`position`	8 B (size_t)	indeks rekordu w pliku głównym
`data`	R bajtów	nowe wartości rekordu

Każda modyfikacja dopisuje nowy wpis na koniec pliku cienia. Przy wielu modyfikacjach tego samego rekordu plik może zawierać wiele wpisów dla tej samej pozycji — aktualny jest ostatni.

Priorytety odczytu

Priorytety odczytu to reguła rozstrzygania, z którego źródła system ma zwrócić wartość rekordu, gdy ten sam indeks może występować jednocześnie w pliku głównym i w pliku cienia. W RetractorDB priorytet definiowany jest deterministycznie: najpierw sprawdzany jest `.shadow` (od końca, aby wybrać najnowszą modyfikację), a dopiero przy braku wpisu wykonywany jest odczyt z pliku głównego. Pojęcie to dotyczy aspektu **spójności i wersjonowania odczytu** danych po modyfikacjach, a nie samego fizycznego formatu zapisu rekordu w pliku binarnym.

Rys. 12. Priorytety odczytu rekordu z pliku cienia

Rys. 12 przedstawia logikę odczytu rekordu: system najpierw sprawdza wpis w `.shadow`, a dopiero przy jego braku odczytuje rekord z pliku głównego.

Scalanie (merge)

Operacja `merge()` scala zmiany z pliku cienia do pliku głównego i zeruje plik cienia. Po scaleniu dane oryginalne są bezpowrotnie nadpisane.

Rys. 13. Scalanie pliku cienia z plikiem głównym

Rys. 13 przedstawia przebieg `merge()`: kolejne wpisy `(position, data)` z `.shadow` są zapisywane do pliku głównego, a po zakończeniu plik cienia jest czyszczony.

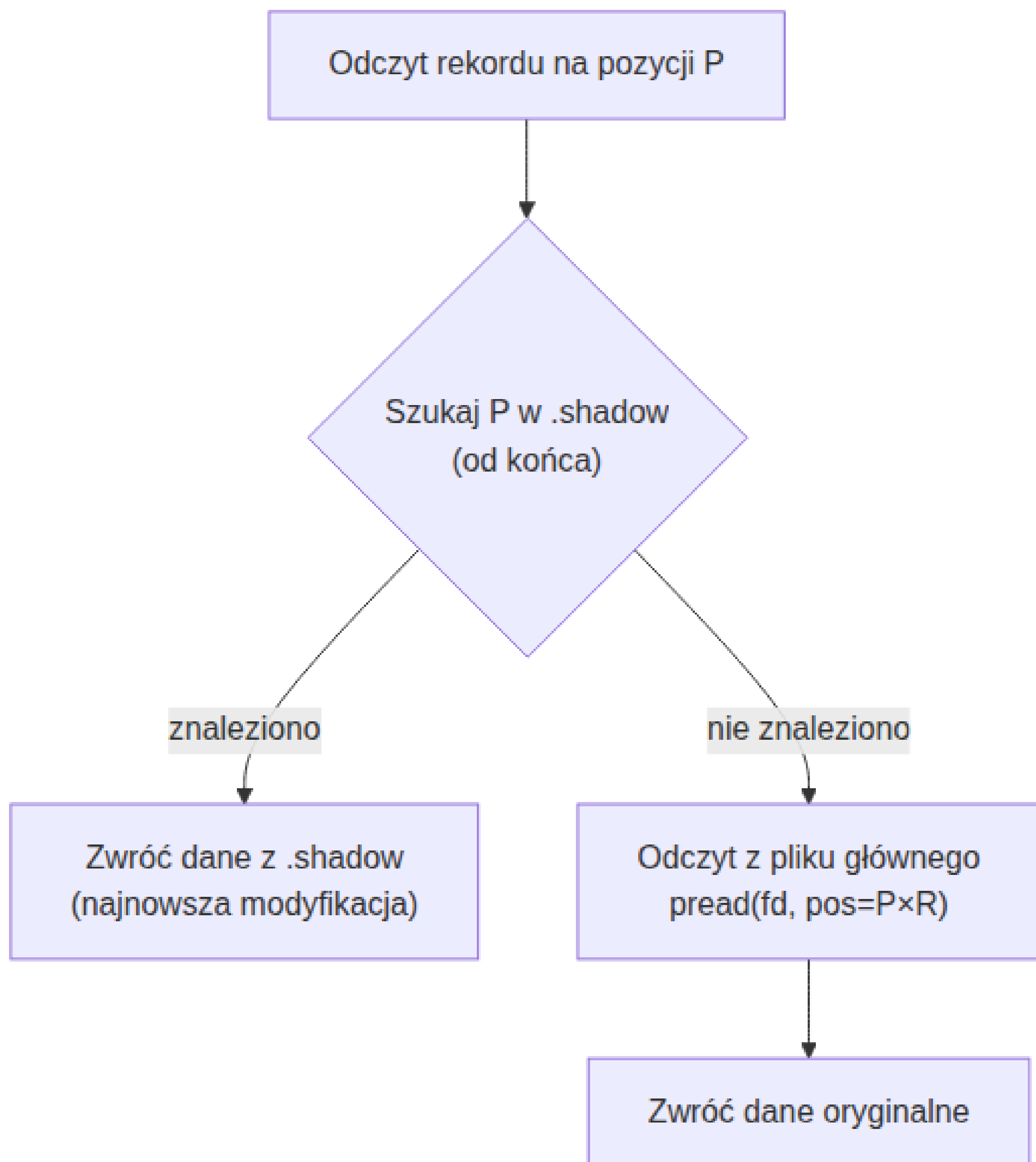
Przykład: modyfikacja rekordu

```
# Strumień str1: 2 pola INTEGER (4B każde), recordSize = 8B
# Rekord 2 (oryginał): [100, 200]
# Modyfikacja: pole 0 → 999

# Plik .shadow po modyfikacji:
# offset 0: [position=2 (8B)][999, 200 (8B)]
```

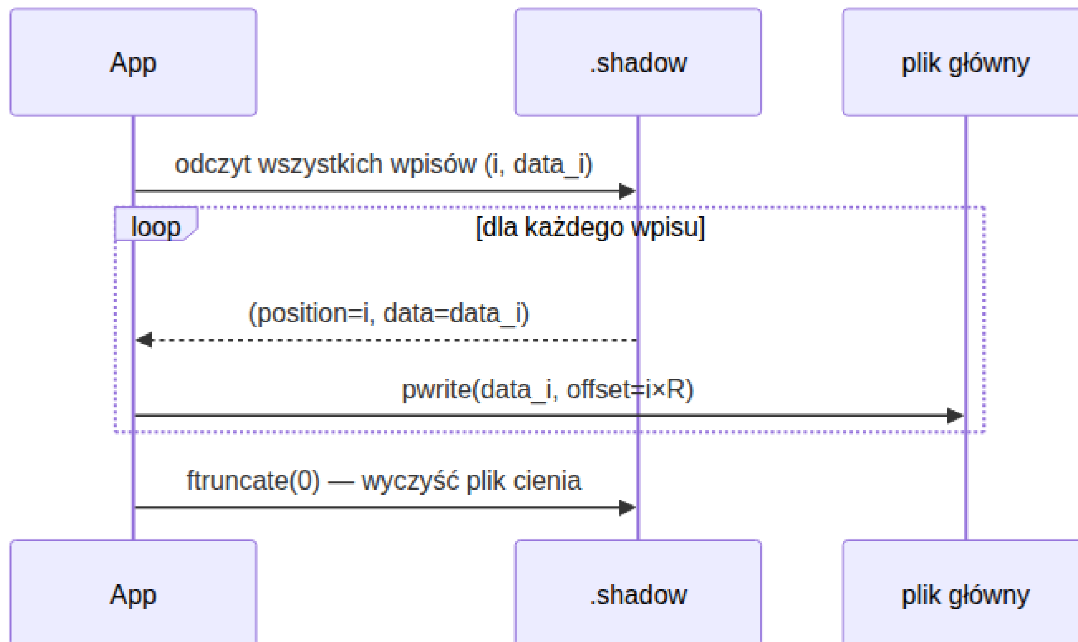
Odczyt rekordu 2 zwróci `[999, 200]`. Odczyt rekordu 0 i 1 zwróci dane z pliku głównego (nie ma ich w shadow).

28.5 Plik cienia indeksu (.meta.shadow)

Plik `.meta.shadow` jest odpowiednikiem `.shadow` na poziomie indeksu null. Rejestruje nadpisania wzorców null dla poszczególnych rekordów bez modyfikowania głównego pliku `.meta`, zachowując spójność pary: `plik główny  .meta` oraz `plik cienia  .meta.shadow`.



Diagram



Diagram

Kiedy powstaje

Plik `.meta.shadow` jest tworzony automatycznie przez `metaDataStream`, gdy spełnione są dwa warunki:

1. Magazyn jest typu `DEFAULT` lub `POSIXSHD` — czyli taki, który trzyma modyfikacje rekordów w pliku `.shadow` (nie w pliku głównym).
2. W danej sesji wykonana zostanie przynajmniej jedna modyfikacja istniejącego rekordu (`storage::write()` na indeks inny niż maksymalny).

Warunek 1 sprawdzany jest podczas `storage::attachStorage()` — jeżeli jest spełniony, wywoływane jest `metaDataStream::setShadowMode(true)`.

Format pliku

Plik `.meta.shadow` nie ma nagłówka. Jest sekwencją wpisów w tym samym formacie binarnym co wpisy w pliku `.meta`, z tą różnicą, że pole `recordCount` przechowuje **bezwzględny indeks rekordu** (nie liczbę rekordów w serii RLE):

Pole	Rozmiar	Znaczenie w `.meta.shadow`
`gapFlag`	1 B	zawsze 0 (nadpisania nie są przerwami)
`recordCount`	8 B (size_t)	bezwzględny indeks nadpisywanego rekordu
`bitsetSize`	8 B (size_t)	liczba pól deskryptora (N)
`bitset`	⌈N/8⌉ B	nowy wzorzec null dla tego rekordu

Każde wywołanie `onRecordModified()` w trybie cienia dopisuje jeden wpis na koniec

pliku. Wiele wpisów dla tej samej pozycji jest dozwolone — obowiązuje **ostatni** wpis (semantyka „last-write-wins", zgodna z plikiem `.shadow`).

Priorytety odczytu

W trybie cienia `getNullBitset(i)` skanuje listę nadpisań od końca. Jeżeli znajdzie wpis dla indeksu `i`, zwraca jego wzorzec null bez sięgania do głównego indeksu:

Rys. 15. Priorytety odczytu wzorca null — główny indeks vs. cień indeksu

Cykl życia

Plik `.meta.shadow` jest zarządzany równolegle z plikiem cienia danych:

Zdarzenie na pliku `.shadow`	Akcja na `.meta.shadow`
Pierwsza modyfikacja rekordu	Tworzenie pliku; dołączenie pierwszego wpisu
Kolejne modyfikacje	Dołączanie kolejnych wpisów
`merge()` — scalenie cienia z plikiem głównym	`mergeShadow()` — nadpisania aplikowane do `.meta`; plik usuwany
`purge()` / `reset()` — odrzucenie cienia	`discardShadow()` — plik usuwany bez scalania
Restart procesu	`setShadowMode(true)` → `loadShadow()` — plik odczytywany; nadpisania przywrócone w pamięci
Usunięcie tymczasowego magazynu (destruktor)	Plik `.meta.shadow` usuwany razem z `.meta`

Persystencja po restarcie

Po restarcie procesu nowy obiekt `metaDataStream` przywraca stan cienia przez `loadShadow()`:

1. Odczytuje wszystkie wpisy z `.meta.shadow` (brak nagłówka — format bezpośredni).
2. Ładuje je do `shadowOverrides_` w kolejności zapisu.
3. `getNullBitset()` i kolejne `onRecordModified()` działają tak samo jak przed restartem.

Przykład użycia — korekta rekordu z zachowaniem spójności

```
# 5 rekordów w strumieniu str1, 3 pola FLOAT
# Rekord 2 ma wartość null w polu 0: nullBitset=[T,F,F]
# Operator koryguje pole 0 rekordu 2 → zmiana wzorca na [F,F,F]

# Operacje:
storage.write(rec2_corrected, pos=2)
  → .shadow: dołącz (position=2, data_corrected)
  → metaDataStream.onRecordModified(2, [F,F,F])
```

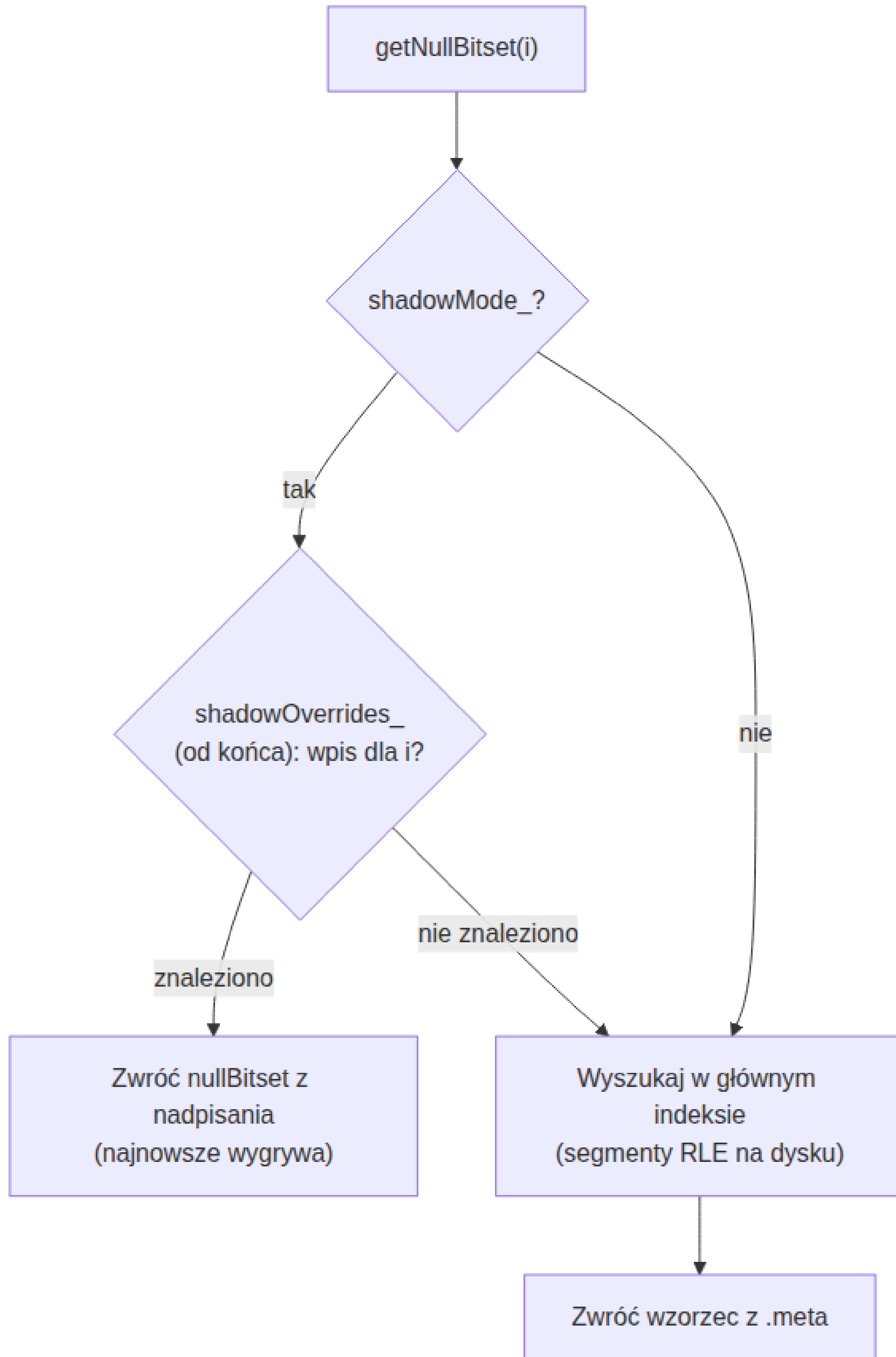
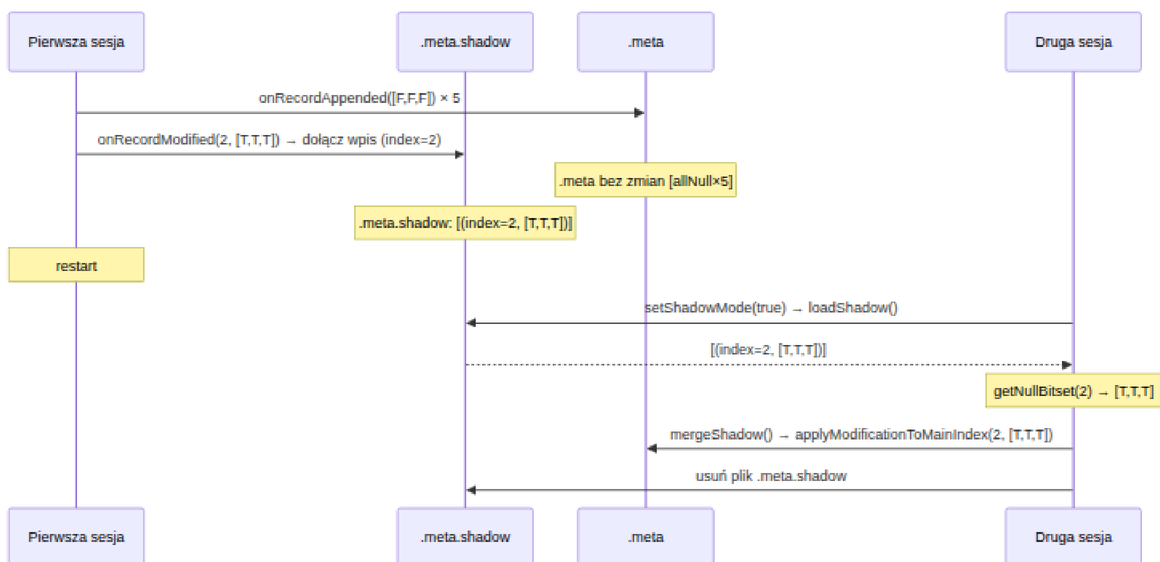


Diagram
104



Diagram

```
    → tryb cienia: .meta.shadow: dołącz (index=2, [F,F,F])

# Stan plików:
# .meta        - bez zmian: [isGap=F, count=2, [F,F,F]], [isGap=F, count=1, [T,F,F]], [isGap=F,
# .meta.shadow - nowy wpis: [gapFlag=0, recordCount=2, bitset=[F,F,F]]

# Odczyt:
getNullBitset(2) → [F,F,F]  (z .meta.shadow)
getNullBitset(1) → [F,F,F]  (z .meta)

# Po scaleniu:
storage.merge() → .shadow wchłonięty do pliku głównego
metaDataStream.mergeShadow() → .meta przebudowany, .meta.shadow usunięty
# .meta po merge: [isGap=F, count=5, [F,F,F]]  (wszystkie rekordy pełne)
```

Uwaga

Mechanizm `.meta.shadow` jest testowany w teście jednostkowym `scenariusz_cien_indeksu` (`test_metaDataStream_usage.cpp`).

28.6 Relacja pomiędzy plikami

W tej części relacje między plikami są pokazane na dwóch poziomach. Poziom strukturalny opisuje, że plik danych jest nośnikiem rekordów, deskryptor `.desc` definiuje ich format, plik `.meta` przechowuje informację o wartościach null i przerwach transmisji, `.shadow` gromadzi modyfikacje danych bez niszczenia oryginału, a `.meta.shadow` gro-

madzi analogicznie nadpisania wzorców null. Poziom operacyjny (Rys. 16) pokazuje przebieg odczytu i zapisu: odczyt najpierw sprawdza `.shadow` i `.meta.shadow`, `merge()` przenosi poprawki do pliku głównego i głównego indeksu, a operacje `append`, `update` i `read` utrzymują spójność danych i metadanych w całym cyklu życia artefaktu.

Rys. 16. Relacja pomiędzy operacjami zapisu, modyfikacji i odczytu artefaktu

Rys. 16 przedstawia przepływ operacji `append`, `update` i `read` przez warstwę `storage` oraz ich bezpośredni wpływ na plik danych, `.meta`, `.shadow` i `.meta.shadow`.

28.7 Punkt wyjścia — plik binarny bez metadanych

Najprostszy możliwy zapis serii czasowej to sekwencja surowych wartości w pliku binarnym: stały rozmiar rekordu, brak nagłówka, brak opisu struktury. Takie podejście ma jedną zaletę — minimalny narzut — i szereg istotnych ograniczeń:

- Interpretacja danych wymaga wiedzy zewnętrznej wobec pliku (nazwy pól, typy, kolejność).
- Brak informacji o przerwach w transmisji — ciągłość danych jest pozorna.
- Każda modyfikacja historycznego rekordu niszczy dane oryginalne nieodwracalnie.
- Zmiana struktury rekordu unieważnia cały plik.

RetractorDB rejestruje dane z czujników działających w czasie rzeczywistym, gdzie przerwy zasilania, zaniki sygnału i konieczność retrospektywnej korekty danych są normalnym zjawiskiem eksploatacyjnym, nie wyjątkiem. Struktura czterech plików odpowiada bezpośrednio na każde z tych ograniczeń.

28.8 Co wnosi każdy plik

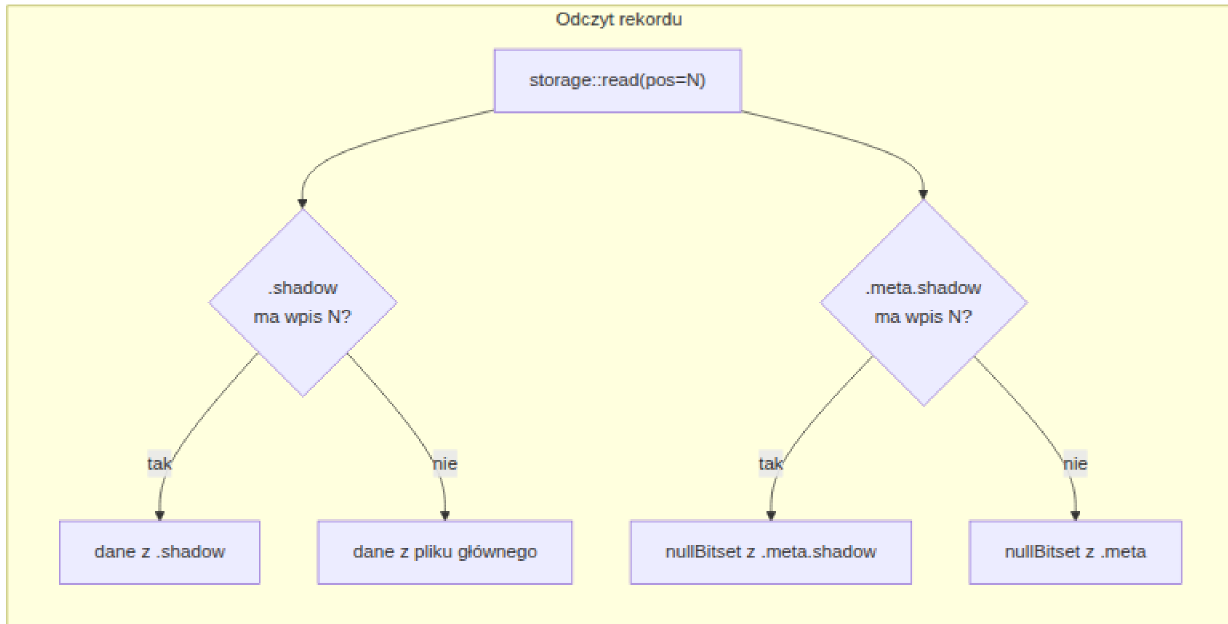
Deskryptor (`.desc`) — samoopisywalność i niezależność od kodu

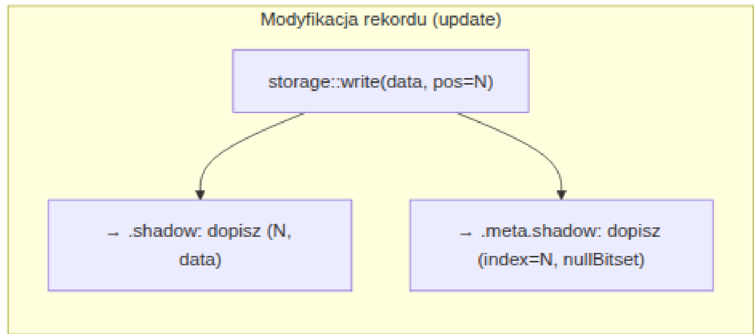
Plik danych binarnych jest bezużyteczny bez znajomości struktury rekordu. Deskryptor przechowuje tę wiedzę obok danych, co oznacza:

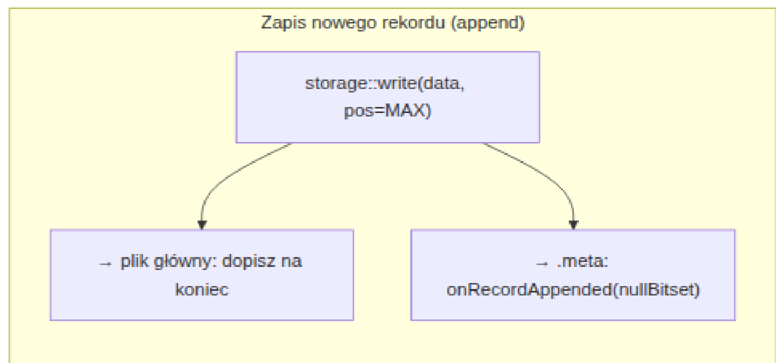
- Dane można odczytać i zinterpretować bez dostępu do kodu źródłowego ani konfiguracji — wystarczy plik `.desc`.
- Narzędzie `xtrdb` może analizować dowolny artefakt bez dodatkowych parametrów.
- Zmiana struktury strumienia (dodanie pola, zmiana typu) jest jawna i wersjonowalna.
- Pole `TYPE` w deskryptorze decyduje o strategii składowania, co pozwala temu samemu silnikowi obsługiwać trwałe artefakty, ulotne efemerydy i zewnętrzne źródła danych bez zmiany logiki zapytań.

Plik metadanych (`.meta`) — wiarygodność serii czasowej

Seria czasowa z dziurami, traktowana jako ciągła, prowadzi do błędnych obliczeń okien czasowych, błędnych agregacji i fałszywych korelacji. Plik `.meta` zapewnia:







Diagram

- Odróżnienie rekordu z wartością zero od rekordu nieobecnego (null) — semantycznie zupełnie różnych stanów.
- Rejestrację przerw w transmisji bez wstawiania fikcyjnych rekordów do pliku danych — plik binarny pozostaje gęsty i adresowalny pozycyjnie.
- Kompresję RLE — typowe serie czasowe mają długie okresy bez null, więc koszt metadanych jest bliski zeru dla danych dobrej jakości.
- Możliwość odtworzenia dokładnego harmonogramu rejestracji, w tym długości przerw, co jest niezbędne przy obliczaniu interwałów w algebrze strumieni.

Plik cienia (`.shadow`) — niedestruktywna korekta danych

W systemach pomiarowych korekta błędnych próbek po fakcie jest standardową procedurą. Nadpisanie pliku binarnego jest nieodwracalne i usuwa dowód oryginalnego pomiaru. Plik cienia:

- Pozwala skorygować dowolny historyczny rekord bez modyfikacji pliku głównego.
- Zachowuje oryginalny pomiar jako domyślny — usunięcie pliku `.shadow` w pełni przywraca stan wyjściowy.
- Umożliwia scalenie (`merge`) korekt do pliku głównego wtedy, gdy jest to świadoma decyzja operatora, nie skutek uboczny zapisu.
- Separuje dane certyfikowane (plik główny) od danych roboczych (plik cienia), co ma znaczenie w zastosowaniach wymagających audytowalności.

Plik cienia indeksu (`.meta.shadow`) — spójność metadanych przy korekcie

Korekta rekordu w pliku cienia danych musi znaleźć odzwierciedlenie w indeksie null — inaczej `getNullBitset()` zwróciłoby przestarzały wzorzec z głównego `.meta`. Plik `.meta.shadow`:

- Utrzymuje spójność między parami: `plik główny` ↔ `.meta` oraz `.shadow` ↔ `.meta.shadow`.
- Pozwala `getNullBitset()` zwrócić aktualny wzorzec null dla skorygowanego rekordu bez modyfikowania głównego indeksu.
- Śledzi cykl życia pliku cienia danych — scalany i usuwany dokładnie razem z `.shadow`.
- Umożliwia pełne odtworzenie stanu po restarcie: nadpisania załadowane z `.meta.shadow` są natychmiast dostępne bez ponownego skanowania pliku cienia danych.

Rozdział 29

Mechanizm rotacji plików

Przez rotację plików rozumiemy kontrolowane zamykanie bieżącego zestawu plików danych i metadanych oraz przeniesienie ich do wersji historycznych (`.old<N>`), tak aby nowa sesja mogła rozpocząć zapis od czystego stanu bez utraty wcześniejszych pomiarów. Stosuje się to po to, aby oddzielić kolejne sesje akwizycji, zachować pełną ścieżkę audytu i ułatwić diagnostykę problemów w czasie. Celem rotacji jest jednocześnie utrzymanie porządku operacyjnego (aktualny zestaw roboczy + archiwum sesji) oraz zapewnienie możliwości odtworzenia i porównania danych historycznych.

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `rotation_test`, `retention` opisanych w załączniku pt. Testy Integracyjne.

29.1 Domyślne zachowanie (bez dyrektywy `ROTATION`)

Bez dyrektywy `ROTATION` w skrypcie RQL, `xretractor` przy każdym starcie **usuwa** pliki artefaktów (dane binarne, `.desc`, `.meta`) i zaczyna rejestrację od nowa. Pliki deklaracji (`DECLARE`) oraz efemerydy nie są usuwane — nie mają plików na dysku.

29.2 Dyrektywa `ROTATION` i licznik sesji

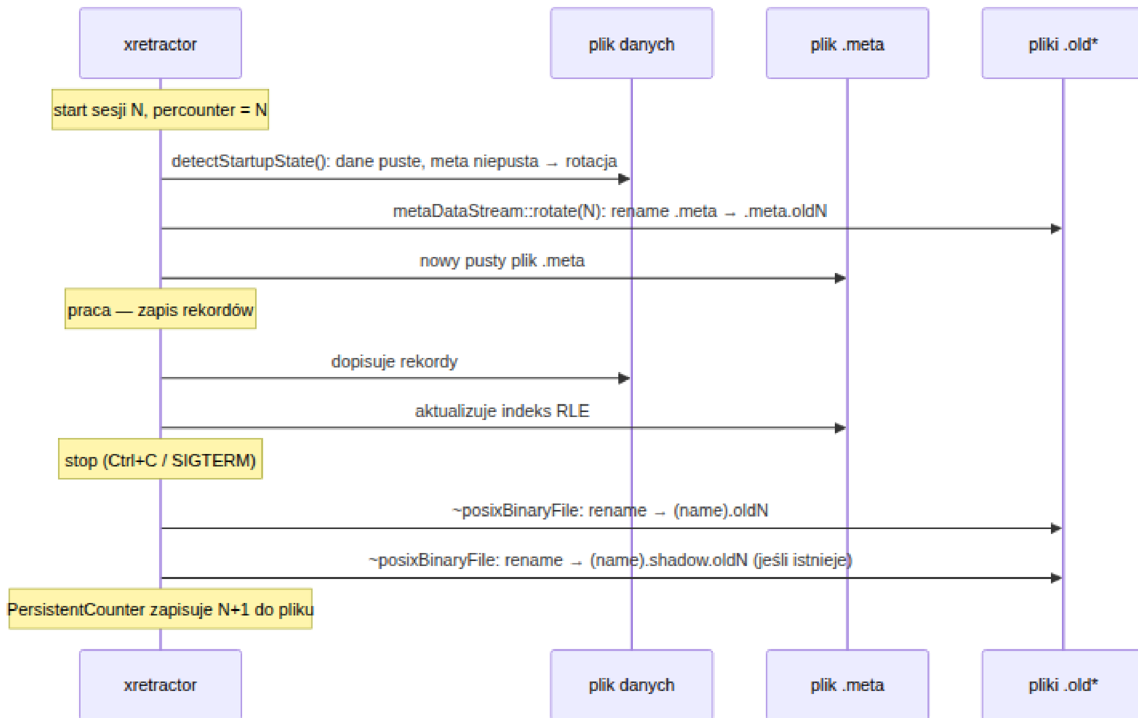
Dyrektywa `ROTATION` włącza tryb zachowania historii. Przyjmuje ścieżkę do pliku przechowującego trwały licznik sesji:

```
ROTATION rdb_counter
```

Obiekt `PersistentCounter` wczytuje wartość `N` z pliku przy starcie (`getCount() = N`) i zapisuje `N+1` przy zamknięciu. Licznik rośnie monotonicznie z każdą sesją `xretractor`.

29.3 Przepływ sterowania w procesie rotacji

W tym punkcie chcemy pokazać pełną sekwencję życia plików podczas jednej sesji i przejścia do kolejnej. Diagram ma wyjaśnić kolejność zdarzeń: wykrycie rotacji przy starcie, utworzenie nowego indeksu `.meta`, normalny zapis danych w trakcie pracy oraz archiwizację plików przy zamknięciu procesu. Kluczowy przekaz jest taki, że rotacja nie jest pojedynczą operacją, lecz procesem rozłożonym w czasie, który łączy moment startu i stopu sesji.



Diagram

Rotacja pliku `.meta` następuje **przy starcie** sesji N — `detectStartupState()` wykrywa niezgodność (plik danych pusty, indeks niepusty ze starej sesji) i wywołuje `metaDataStream::rotate(N)`. Plik danych binarnych jest przemianowywany dopiero przy **zamknięciu** sesji przez destruktor `posixBinaryFile`.

29.4 Co trafia do plików `.old<N>`

Plik	Kiedy powstaje
`<name>.oldN`	Zamknięcie sesji N — destruktor `posixBinaryFile` przemianowuje plik danych
`<name>.shadow.oldN`	Zamknięcie sesji N — destruktor `posixBinaryFileWithShadow` przemianowuje plik cienia
`<name>.meta.oldN`	Start sesji N — `detectStartupState()` wykrywa rotację i przemianowuje `.meta` pozostawiony przez sesję N−1

Wskutek tej kolejności: plik `.meta.oldN` zawiera metadane null dla danych z sesji `N-1`, podczas gdy plik `.oldN` zawiera dane sesji `N`. W sekcji `ROTATED FILES` narzędzia `xtrdb -s` pliki są grupowane według numeru suffiksu — pary `.oldN` i `.meta.oldN` różnią się więc o 1 w stosunku do sesji, której fizycznie odpowiadają.

29.5 Przykład sekwencji trzech sesji

Po trzech zakończonych sesjach (0, 1, 2) i w trakcie czwartej (3):

```
pomiar.old0        ← dane z sesji 0 (zapis sesji 0, przemianowanie w destruktorze sesji 0)
pomiar.meta.old1   ← metadane z sesji 0 (przemianowanie przy starcie sesji 1)
pomiar.old1        ← dane z sesji 1
pomiar.meta.old2   ← metadane z sesji 1 (przemianowanie przy starcie sesji 2)
pomiar.old2        ← dane z sesji 2
pomiar.meta.old3   ← metadane z sesji 2 (przemianowanie przy starcie sesji 3)
pomiar             ← dane bieżące (sesja 3)
pomiar.meta        ← metadane bieżące (sesja 3)
```

Widok `xtrdb -s` w trakcie sesji 3:

```
$ xtrdb -s pomiar
...

  ROTATED FILES
  [3] pomiar.meta.old3                                26 B
  [2] pomiar.old2                                    800 B
      pomiar.meta.old2                                26 B
  [1] pomiar.old1                                    800 B
      pomiar.meta.old1                                26 B
  [0] pomiar.old0                                    400 B
```

Plik `pomiar.meta.old3` jest w grupie `[3]` sam — odpowiadający mu plik `pomiar.old3` powstanie dopiero przy zamknięciu bieżącej sesji.

29.6 Otwieranie pliku rotowanego w `xtrdb`

Pliki rotowane można analizować poleceniem `open` w trybie interaktywnym `xtrdb`. Polecenie `open` automatycznie wyciąga nazwę bazową (usuwa `.old<N>`) i szuka deskryptora `<nazwa_bazowa>.desc`:

```
$ xtrdb
. open pomiar.old1
ok
. print
...
```

Rozdział 30

Narzędzie inspekcji: `xtrdb -s`

Polecenie `xtrdb -s <ścieżka>` wyświetla kompletny obraz stanu składowania artefaktu — bez otwierania procesu `xretractor`, bez wchodzenia w tryb interaktywny. Wystarczy wskazać ścieżkę bazową (bez rozszerzenia), a narzędzie samo znajdzie powiązane pliki: `.desc`, dane binarne, `.meta`, `.shadow`, segmenty cykliczne i pliki rotowane.

Uwaga

Opisana funkcjonalność ma pokrycie w teście: `issue153_storagemap_meta_cases` opisanym w załączniku pt. Testy Integracyjne.

30.1 Cel i zastosowanie

Sytuacja	Co daje `xtrdb -s`
Diagnoza po awarii	Widać od razu, czy plik danych jest spójny z metadanymi — różne liczby rekordów sygnalizują problem
Weryfikacja retencji	Sekcja DATA TOTAL pokazuje podział na segmenty i aktualny stopień wypełnienia bufora cyklicznego
Kontrola modyfikacji	Sekcja SHADOW ujawnia liczbę niezatwierdzonych zmian — `Updates: N` oznacza, że `merge()` nie był wykonany
Analiza jakości danych	Pasek META z symbolami `=`, `-`, `~`, `X` pokazuje wzorzec null i przerwy bez parsowania pliku binarnego
Audyt historii rotacji	Sekcja ROTATED FILES wymienia stare wersje pliku po kolejnych rotacjach

Polecenie jest **tylko do odczytu** — nie modyfikuje żadnego pliku. Można je uruchamiać również gdy `xretractor` nie działa.

30.2 Co pokazuje mapa

Górna część raportu to trzyelementowa mapa poglądowa:

```
[shadow]    [binary data]  [meta index]
```

Każdy wiersz mapy odpowiada jednemu segmentowi RLE lub segmentowi danych:

Kolumna	Zawartość
`[shadow]`	Dla artefaktu bez retencji: liczba niezapisanych modyfikacji (`N updates`). Dla retencji segmentowej: etykieta segmentu `sN` z liczbą modyfikacji.
`[binary data]`	Zakres indeksów rekordów w pliku binarnym (`begin-end`) lub etykieta segmentu `sN begin-end`. Wiersze z przerwą w transmisji (gap) mają puste pole.
`[meta index]`	Opis segmentu RLE z pliku `.meta`: liczba rekordów i wzorzec null w formie `[====]`.

Poniżej mapy następują kolejne sekcje:

Sekcja	Opis
`DESCRIPTOR`	Ścieżka i rozmiar pliku `.desc`, lista pól z typami i rozmiarami, rozmiar rekordu w bajtach.
`DATA`	Liczba rekordów, ścieżka do pliku danych. Przy retencji (`RETENTION`): podział na segmenty, polityka (liczba segmentów i pojemność), maksymalny dopuszczalny rozmiar bufora, lista plików `_segment_*`.
`META`	Liczba segmentów RLE i rekordów w indeksie, graficzny pasek obrazujący wzorzec null w czasie.
`SHADOW`	Ścieżka i rozmiar pliku cienia oraz liczba niezatwierdzonych modyfikacji.
`ROTATED FILES`	Pliki z poprzednich rotacji (`.old1`, `.old2`, ...) wraz z rozmiarami.

Legenda paska META

```
[====] - dane bez wartości null
[----] - częściowe null (przynajmniej jedno pole ma wartość null)
[~~~~] - wszystkie pola mają wartość null (nullfill)
[XXXX] - przerwa w transmisji (gap)
```

30.3 Przykład 1 — artefakt prosty

Strumień `pomiar` z dwoma polami, 100 rekordów, bez modyfikacji, bez przerw:

```
{
  INTEGER  ts
  FLOAT    value
  TYPE     DEFAULT
}
```

```
$ xtrdb -s pomiar
```

```
Storage map: pomiar

[shadow]   | [binary data] | [meta index]
           | 0-100         | [====] 100 records, no nulls

DESCRIPTOR  pomiar.desc                              43 B
  INTEGER   ts                                        4 B
  FLOAT     value                                     4 B
  Record size:                                        8 B

DATA        pomiar                                  800 B
  Records: 100

META        pomiar.meta                              26 B
  Segments: 1   Records: 100
  [=======================100========================]
  Legend: [====] data  [----] partial null
          [~~~~] nullfill  [XXXX] gap

SHADOW      pomiar.shadow (missing)                   0 B
```

Interpretacja: jeden segment RLE, brak przerw, brak null, plik cienia nieobecny. Plik binarny ma dokładnie $100 \times 8 = 800$ bajtów.

30.4 Przykład 2 — artefakt z przerwą w transmisji i modyfikacją

Strumień `czujnik` z trzema polami. Po 50 rekordach nastąpiła przerwa (10 jednostek interwału), następnie napłynęło 30 rekordów z częściowymi brakami w polu `pressure`. Dwa rekordy zostały później zmodyfikowane (plik cienia obecny):

```
{
  INTEGER  ts
  FLOAT    temp
  FLOAT    pressure
  TYPE     DEFAULT
```

```
}
```

```
$ xtrdb -s czujnik
```

```
Storage map: czujnik

[shadow]   | [binary data] | [meta index]
           | 0-50          | [====] 50 records, no nulls
           |               | [XXXX] 10 records, gap
2 updates  | 50-80         | [----] 30 records, some nulls

DESCRIPTOR  czujnik.desc                              52 B
  INTEGER   ts                                         4 B
  FLOAT     temp                                       4 B
  FLOAT     pressure                                   4 B
  Record size:                                        12 B

DATA        czujnik                                  960 B
  Records: 80

META        czujnik.meta                              60 B
  Segments: 3   Records: 80
  [========50=========][gap:10][===========30============]
  Legend: [====] data  [----] partial null
          [~~~~] nullfill  [XXXX] gap

SHADOW      czujnik.shadow                            26 B
  Updates: 2
```

Interpretacja: plik binarny zawiera 80 rekordów (gap nie zajmuje miejsca w pliku danych), przerwa jest zakodowana wyłącznie w `.meta`. Kolumna `[binary data]` pokazuje pusty zakres dla segmentu gapowego — dane binarnych nie ma. Pole `pressure` w rekordach 50–79 ma wartości null w niektórych polach (`[----]`). Plik cienia zawiera 2 modyfikacje, które jeszcze nie zostały scalone z plikiem głównym.

30.5 Przykład 3 — artefakt z retencją segmentową

Strumień `bufor` z retencją cykliczną: maksymalnie 10 segmentów po 100 rekordów (łącznie 1000 rekordów). Aktualnie zapisano 280 rekordów w trzech segmentach:

```
{
  DOUBLE   value
  TYPE     DEFAULT
  RETENTION 1000 100
}
```

```
$ xtrdb -s bufor
```

```
Storage map: bufor
```

```
[shadow]    | [binary data] | [meta index]
s0          | s0 0-100      | [====] 100 records, no nulls
s1          | s1 100-200    | [====] 100 records, no nulls
s2          | s2 200-280    | [====] 80 records, no nulls

DESCRIPTOR  bufor.desc                               48 B
  DOUBLE    value                                     8 B
  Record size:                                        8 B

DATA TOTAL  rec=280 src=0 seg=280                  2240 B
  Records: 280
  Source: bufor   Segments: bufor_segment_*
  Segmented data (RETENTION): 3
  Policy: segments=10 capacity=100
  Retention cap records: 1000
  Retention cap bytes: 8000
  Total records: 280
    current=0  segments=280
  Total bytes: 2240
    current=0  segments=2240
    [0] bufor_segment_0 rec:100 range:0-100
    [1] bufor_segment_1 rec:100 range:100-200
    [2] bufor_segment_2 rec:80 range:200-280

META        bufor.meta                               26 B
  Segments: 1   Records: 280
  [=======================280========================]
  Legend: [====] data  [----] partial null
          [~~~~] nullfill  [XXXX] gap

SHADOW      bufor.shadow (missing)                    0 B
```

Interpretacja: kolumna `[binary data]` pokazuje każdy segment z etykietą `sN` i zakresem indeksów globalnych. Sekcja DATA TOTAL zawiera pełne zestawienie: `src=0` (brak rekordów poza segmentami), `seg=280` (wszystkie rekordy w segmentach). Przy wypełnieniu bufora (10 segmentów × 100 = 1000 rekordów) najstarszy segment zostanie usunięty, a nowy dopisany.

Rozdział 31

Podsumowanie: uzasadnienie przyjętej struktury

Rozdział zbiera wnioski z wszystkich części dokumentacji formatu zapisu danych i wyjaśnia, dlaczego przyjęta struktura czterech plików jest minimalna i wystarczająca dla systemu rejestracji serii czasowych działającego w czasie rzeczywistym.

31.1 Zestaw plików i typy akcesorów

Każdy artefakt lub substrat składa się z maksymalnie czterech plików — plik danych binarnych, deskryptor `.desc`, indeks `.meta` i plik cienia `.shadow`. Pole `TYPE` w deskryptorze wybiera implementację `FileInterface`: `DEFAULT` (dane + cień + retencja), `MEMORY` (wyłącznie RAM, efemerydy), `DEVICE` / `TEXTSOURCE` (zewnętrzne źródła tylko do odczytu) i warianty pośrednie. Wybór akcesora następuje raz przy inicjalizacji `storage` — logika zapytań RQL nie zna szczegółów składowania.

31.2 Pliki artefaktu

Deskryptor (`.desc`) definiuje schemat rekordu w gramatyce ANTLR4: nazwy pól, typy (`BYTE`, `INTEGER`, `FLOAT`, `DOUBLE`, `RATIONAL`, `STRING`), rozmiary tablic, politykę retencji (`RETENTION`, `RETMEMORY`) i typ akcesora (`TYPE`). Rozmiar rekordu `R` to suma bajtów wszystkich pól danych — pola metadeskryptora nie zajmują miejsca w rekordzie. Deskryptor przy danych oznacza samoopisywalność: narzędzie `xtrdb` lub dowolny kod może zinterpretować artefakt bez dostępu do kodu źródłowego.

Plik danych binarnych to płaska sekwencja rekordów stałej długości `R` bez nagłówka. Rekord `i` leży zawsze na offsecie `i × R`. Operacja `append` dopisuje na koniec; operacja `update` — przy obecnym `.shadow` — trafia do pliku cienia, nie nadpisuje pliku głównego.

Plik metadanych (`.meta`) przechowuje kompresowany RLE indeks wartości null i przerw w transmisji. Każdy wpis RLE opisuje ciąg kolejnych rekordów z identycznym wzorcem null: flagę `isGap`, liczbę rekordów `recordCount`, rozmiar bitset i sam bitset. Przerwa w transmisji (`gap`) istnieje wyłącznie w `.meta` — plik binarny jej nie rejestruje

i pozostaje gęsty. Klasą zarządzającą jest `rdb::metaDataStream`: buforuje bieżący segment w `currentEntry_`, zapisuje segment na dysk tylko przy zmianie wzorca, a mechanizm `tailDirty_` zapewnia, że rozmiar pliku nie rośnie przy ciągłym napływie jednorodnych danych. Po restarcie `loadIndex()` odtwarza stan i przenosi ostatni niegapowy segment z powrotem do pamięci, umożliwiając kontynuację RLE.

Plik cienia (`.shadow`) gromadzi modyfikacje rekordów jako sekwencję wpisów `(position, data)`. Odczyt rekordu sprawdza `.shadow` od końca (najnowsza modyfikacja wygrywa), przy braku wpisu czyta z pliku głównego. Usunięcie `.shadow` w pełni przywraca stan wyjściowy. Operacja `merge()` przepisuje poprawki do pliku głównego i zeruje plik cienia.

31.3 Mechanizm rotacji

Dyrektywa `ROTATION rdb_counter` włącza tryb zachowania historii sesji. `PersistentCounter` przechowuje monotonicznie rosnący numer sesji `N`. Rotacja jest procesem rozłożonym w czasie: przy **starcie** sesji N funkcja `detectStartupState()` wykrywa niezgodność (plik danych pusty, `.meta` niepusty) i przemianowuje `.meta` na `.meta.oldN`; przy **zamknięciu** sesji destruktor `posixBinaryFile` przemianowuje plik danych na `.oldN` i plik cienia na `.shadow.oldN`. Konsekwencją tej kolejności jest przesunięcie o 1: `.meta.oldN` zawiera metadane sesji `N-1`, a `.oldN` — dane sesji `N`. Bez dyrektywy `ROTATION` pliki artefaktów są usuwane przy każdym starcie.

31.4 Narzędzie inspekcji `xtrdb -s`

Polecenie `xtrdb -s <ścieżka>` jest jedynym narzędziem do inspekcji stanu składowania bez uruchamiania `xretractor`. Raport składa się z mapy poglądowej (kolumny: shadow, binary data, meta index) i sekcji szczegółowych: `DESCRIPTOR`, `DATA` (lub `DATA TOTAL` przy retencji segmentowej), `META` z paskiem RLE, `SHADOW` z liczbą niezatwierdzonych modyfikacji oraz `ROTATED FILES` z historią rotacji. Pasek META używa czterech symboli: `=` (dane bez null), `-` (częściowe null), `~` (nullfill), `X` (gap). Narzędzie jest tylko do odczytu i działa gdy proces `xretractor` nie działa.

31.5 Porównanie podejść

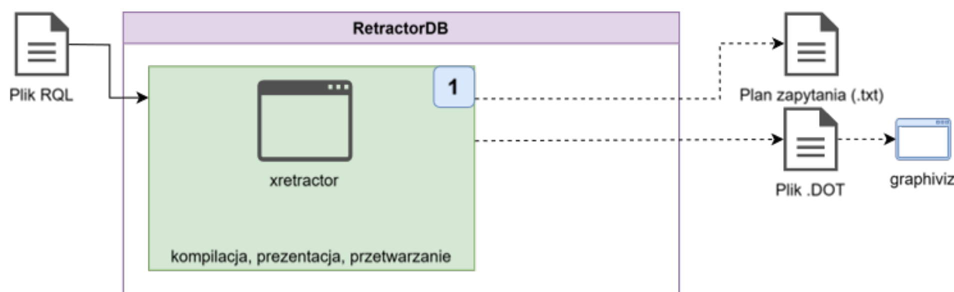
Właściwość	Surowy plik binarny	Struktura RetractorDB
Samoopisywalność	brak — wymaga zewnętrznej dokumentacji	tak — deskryptor `.desc` przy danych
Obsługa przerw w transmisji	brak — przerwy niewidoczne lub fikcyjne rekordy	tak — `.meta` rejestruje przerwy bez rozszerzania pliku danych
Wartości null per pole	brak — zero = null nierozróżnialne	tak — bitset null w `.meta`

Właściwość	Surowy plik binarny	Struktura RetractorDB
Korekta danych historycznych	destruktywna	niedestruktywna — `.shadow`
Przywrócenie oryginału po korekcie	niemożliwe	tak — usunięcie `.shadow`
Wielokrotność strategii składowania	brak	tak — pole `TYPE` w deskryptorze
Koszt przy danych bez przerw i null	—	minimalny: `.meta` ≈ 17 B nagłówek + 1 wpis RLE

Rozdział 32

Kompilacja i budowa planu

Proces kompilacji odbywa się przed każdym uruchomieniem procesu xretractor. Argument w postaci pliku z sekwencją poleceń i zapytań jest wymagany. W oparciu o przepływ przedstawiony na Rys. 10 przygotowałem opis procesu Rys. 15 realizujący proces kompilacji w trybie rozwojowym. Proces kompilacji można wywołać nawet jak już jakiś inny proces xretractor funkcjonuje. Blokowanie jednej instancji procesu przetwarzania danych odnosi się tylko do procesu realizacji planu zapytania. Wywołanie kompilacji w tym przypadku, nawet jeśli funkcjonuje już ten proces w systemie nie zgłosi błędu. Próba uruchomienia kolejnego przetwarzania – tak.



Rys. 15. Proces kompilacji

Jako przykładowy plik przeznaczony do kompilacji przyjmiemy plik query.rql o następującej zawartości:

```
DECLARE a INTEGER
STREAM core0, 0.1
FILE 'datafile1.dat'

SELECT str1[0]+1
STREAM str1
FROM core0>2
```

Jest to bardzo prosty przykład pliku zawierającego dwie dyrektywy. Pierwsza deklaruje istnieje efemerydu w postaci źródła danych binarnych zawierającego 4-bajtowe liczby typu INTEGER. Dane z tego pliku będą czytane z szybkością 10 razy na sekundę. A nazwa tego obiektu to core0.

Drugie polecenie tworzy artefakt o nazwie str1 pobierający przesunięte w czasie od dwa odczyty czyli 0.2 sekundy dane efemeryczne. W trakcie tworzenia kolejnych elementów strumienia wynikowego dochodzi do przetwarzania danych odczytanych z core0 i do każdej odczytanej wartości dodawana jest wartość 1.

Aby przeprowadzić kompilację tego pliku należy wywołać następujące polecenie:

```
$ xretractor -c query.rql
```

Na ekranie wyświetli się następująca odpowiedź systemu:

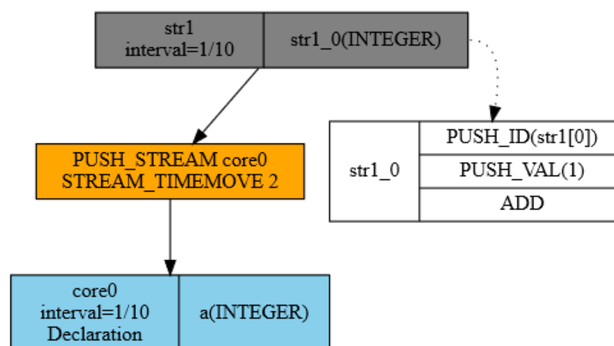
```
str1(1/10)
      :- PUSH_STREAM(core0)
      :- STREAM_TIMEMOVE(2)
      str1_0: INTEGER
            PUSH_ID(str1[0])
            PUSH_VAL(1)
            ADD
core0(1/10) datafile1.dat
      a: INTEGER
```

Pominięcie parametru -c spowoduje podjęcie próby kompilacji i natychmiastowego wysłania skompilowanego planu realizacji zapytania do wykonania. Taka akcja spowoduje wystąpienie błędu. Bowiem pliku z danymi datafile1.dat zapewne jeszcze nie przygotowaliśmy.

Oprócz przeglądu tekstowego możemy obejrzeć również pliki kompilacji w postaci graficznej. Do tego celu należy wywołać następujący ciąg poleceń:

```
$ xretractor -c -d -f -s query.rql > out.dot && dot -Tpng out.dot -o out.png
```

Zakładając że w środowisku uruchomieniowym masz zainstalowany program dot z pakietu graphivz wygenerujesz tym poleceniem plik graficzny przedstawiający odpowiedź systemu w postaci grafu.



Rys. 16. Graficzna reprezentacja planu zapytania

System RetractorDB potrafi wygenerować rysunek jako odpowiedź na jeden ze zleconych ciągów przetwarzania danych. Prezentacja graficzna jest najbardziej odpowiednia w przypadku tworzenia i przedstawiania grafów przetwarzania danych. Niestety czytelność ucierpi w przypadku bardzo skomplikowanych schematów.

Na Rys. 16 widać trywialny plan realizacji zapytania jaki powstał w wyniku kompilacji dwulinijkowego pliku query.rql. U samej góry widać obiekt str1 tworzący artefakty z częstotliwością 10 rekordów na sekundę. Informacja o szybkości tworzenia artefaktów nie występuje w zapytaniu, jest wyznaczana w oparciu wyrażenie algebraiczne z klauzuli FROM w zapytaniu SELECT. Widać też w jaki sposób wytwarzane są kolejne rekordy strumienia str1. Tutaj mamy do czynienia z typowym algorytmem przetwarzania danych na stosie. Najpierw na stos odkładana jest wartość efemeryczna powstałego z wyrażenia algebraicznego a następnie umieszczana jest na stosie wartość 1. Polecenie ADD zdejmuje obie wartości ze stosu pozostawiając na stosie wynik dodawania. To co zostało na stosie – czyli wynik dodawania umieszczane jest w polu tworzonego rekordu.

Z drugiej strony widać operacje na strumieniach. Operacje na strumieniach realizowane są w innej domenie. Tam występuje przetwarzanie obiektów dwu lub jednowartościowych. Operacjom poddawane są albo dwa strumienie albo tylko jeden z argumentem. Klasyczny stos w przypadku Algebraicznych operacji strumieniowych nie ma zastosowania. Dla uproszczenia zapis przypomina trochę operacje na stosie. Widzimy w załączonym przykładzie że operacje na danych bieżących realizujemy poprzez przesunięcie danych w czasie o 2. Celowo nie mówię że to 2 sekundy – tutaj 2 oznacza wartość relatywną względem szybkości napływu. W przypadku szybkości napływu 10 próbek na sekundę – wartość 2 oznacza przesunięcie w czasie o 0.2 sekundy.

Skomplikowane wyrażenia algebraiczne w których biorą udział co najmniej dwa operatory strumieniowe powodują powstanie wspominanych w poprzednich rozdziałach substratów. Każde zapytanie, które zawiera wyrażenia algebraiczne w klauzuli from z więcej niż jednym operatorem są rozbijane na operacje dwuargumentowe, zależne od siebie. Lista argumentów substratu to domyślnie pełne rozwinięcie schematu.

32.1 Dostępne flagi xretractor

W trybie kompilacji (`-c`) i w trybie wykonania dostępne są różne zestawy flag. Poniżej flagi trybu kompilacji używane przy generowaniu grafów:

Flaga	Pełna nazwa	Znaczenie
`-c`	`--onlycompile`	tylko kompilacja — nie uruchamia przetwarzania
`-d`	`--dot`	generuj wyjście w formacie DOT (graphviz)
`-f`	`--fields`	pokaż pola strumieni w grafie DOT
`-s`	`--streamprogs`	pokaż programy strumieni w grafie DOT
`-u`	`--rules`	pokaż reguły RULE w grafie DOT
`-p`	`--transparent`	przezroczyste tło grafu DOT
`-i`	`--hideruleprog`	ukryj program warunku reguły (z `-u`)
`-m`	`--csv`	wyjście w formacie CSV

Flagi trybu wykonania (bez `-c`):

Flaga	Pełna nazwa	Znaczenie
`-m N`	`--tlimitqry N`	uruchom N cykli przetwarzania, potem zakończ
`-k`	`--noanykey`	nie czekaj na klawisz — tryb daemon/skrypt
`-t`	`--realtime`	tryb czasu rzeczywistego (SCHED_FIFO, mlockall)
`-x`	`--xqrywait`	czekaj na pierwsze połączenie xqry przed startem
`-s`	`--status`	sprawdź czy instancja xretractor już działa
`-v`	`--verbose`	wyświetl parametry strumieni przy starcie

Info

Parametr `-m N` liczy iteracje pętli głównej, nie sekundy. Dla strumieni z interwałem 0.1 s (10 Hz), `-m 10` oznacza ~1 sekundę przetwarzania.

Ostrzeżenie

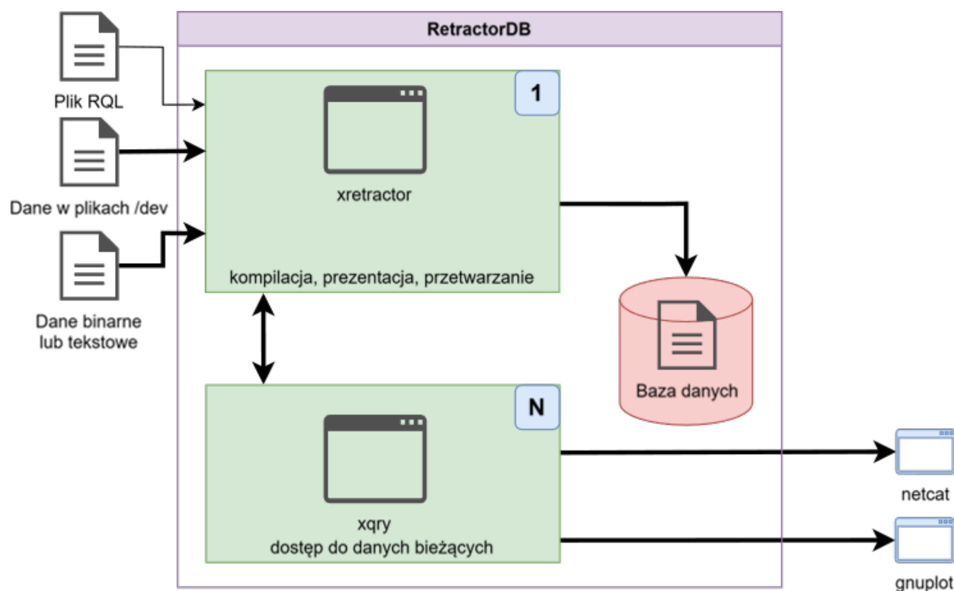
Przy użyciu `-m N` w skryptach i testach zawsze dodawaj `-x` (`--xqrywait`). Bez tej flagi serwer może przetworzyć wszystkie N cykli zanim klient (`xqry`) zdąży się podłączyć — klient nie otrzyma żadnych danych i będzie czekał do przekroczenia limitu czasowego. Flaga `-x` wstrzymuje przetwarzanie do nadejścia pierwszej komendy od `xqry`.

Pełna lista wszystkich opcji z opisem każdej z nich — w tym opcja `--realtime` wymagająca uprawnień systemowych — znajduje się w Załączniku A.

Rozdział 33

Przetwarzanie i dystrybucja danych

W przypadku rozpoczęcia procesu przetwarzania danych analizując przedstawiony na Rys. 10 można wydzielić następujący schemat przepływu - Rys. 17:



Rys. 17. Schemat przepływu sterowania w procesie przetwarzania

Do przeprowadzania procesu przetwarzania potrzebne będzie przygotowanie danych i zbudowanie ciągu przetwarzającego dane. W ramach tego ciągu na wejściu użyjemy przygotowanego pliku z planem realizacji zapytania, przygotujemy plik binarny z danymi. Zbudujemy proces przetwarzający dane i prezentujący wyniki.

Źródłowy plik danych query.rql zmienimy na następujący:

```
DECLARE a INTEGER
STREAM core0, 0.1
FILE 'datafile1.txt'
```

```
DECLARE a BYTE
STREAM core1, 0.2
FILE '/dev/urandom'

SELECT str1[0], str1[0] + str1[1]/20
STREAM str1
FROM core0 + core1
```

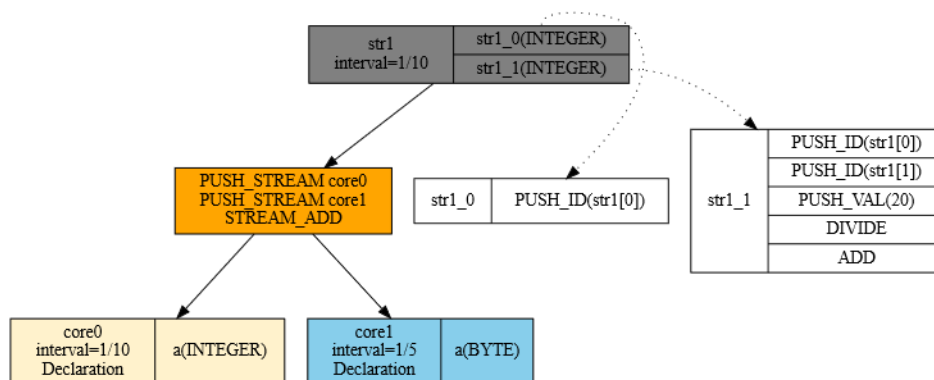
W tym przykładzie deklarujemy istnienie pliku tekstowego zawierającego dane tekstowe. Proponuję wypełnić plik datafile1.txt następującą zawartością:

```
$ seq 20 28 > datafile1.txt
```

```
20
21
22
23
24
25
26
27
28
```

Plik będzie zawierać kolejne liczby od 20 do 28.

Rzut okna na plan realizacji zapytania przedstawi obraz na Rys. 18:



Rys. 18. Graficzna reprezentacja planu realizacji zapytania 2

Jeśli przygotowaliśmy plik z danymi możemy uruchomić proces kompilacji i przetwarzania danych. Realizujemy to wydając następujące polecenie:

```
$ xretractor query.rql
```

I tu pojawia się istotna właściwość opracowanego systemu. System powinien rozpocząć natychmiast realizację procesu. Dowolny klawisz naciśnięty w terminalu przerwie ten proces.

Proponuję uruchomić drugie okno terminala i tam kontynuować sesję. W drugim oknie terminala możemy wydać następujące polecenie:

```
$ xqry -d
| str1|1/10|6912|864|             |0|
|core0|1/10|  -1| 49|datafile1.txt|1|
|core1| 1/5|  -1| 25| /dev/urandom|1|
```

Powinno się pojawić coś podobnego. Oczywiście liczniki danych przy str1 powinny się różnić. Logicznym jest że za każdym odczytem otrzymamy większe wartości dotyczące rozmiaru zgromadzonego strumienia str1.

Jeśli chcemy na ekranie zobaczyć co tam się właśnie dzieje wewnątrz procesu przetwarzania danych proponuję wydać poniższe polecenie i po kilku wierszach na ekranie nacisnąć dowolny klawisz aby przerwać ten proces:

```
$ xqry -s str1
20 26
21 33
22 34
23 27
24 28
25 35
26 36
27 28
```

Pierwsza kolumna zawiera sekwencję liczb – taką jaką wpisaliśmy do pliku datafile1.txt. Druga kolumna zawiera efekt przetwarzania. Dodawana jest wartość pobrana z generatora liczb pseudolosowych podzielona przez 20 – druga kolumna opływa dane poniżej kolumny pierwszej.

Jak można to zobaczyć w formie graficznej? Proponuję wydać następujące polecenie:

```
$ xqry -s str1 -p 50,50 | gnuplot
```

Na ekranie pojawi się następujące okno z płynącymi na bieżąco danymi:

Rys. 19. Zrzut zawartości okna gnuplot przedstawiający dane napływające

Na Rys. 19 widzimy to co dane przedstawiały w postaci numerycznej. Kształt piły to pierwsza kolumna, nieregularny kształt opływający kształt piły to druga kolumna. Rysunek przedstawia dane statyczne – w oknie jednak dane te napływają i rysunek jest aktualizowany na bieżąco.

Typowym pomysłem na wysłanie danych poza system na którym funkcjonuje xretractor i xqry jest użycie polecenia:

```
$ xqry -s str1 | nc -l 8888
```

na drugim komputerze trzeba napisać:

```
$ nc nazwa_serwera_lub_jego_ip 8888
```

Info

Flaga `-p` w netcat (składnia BSD) nie jest obsługiwana przez GNU netcat dostępny na współczesnych systemach Ubuntu/Debian. Poprawna składnia to `nc -l 8888` (bez `-p`).

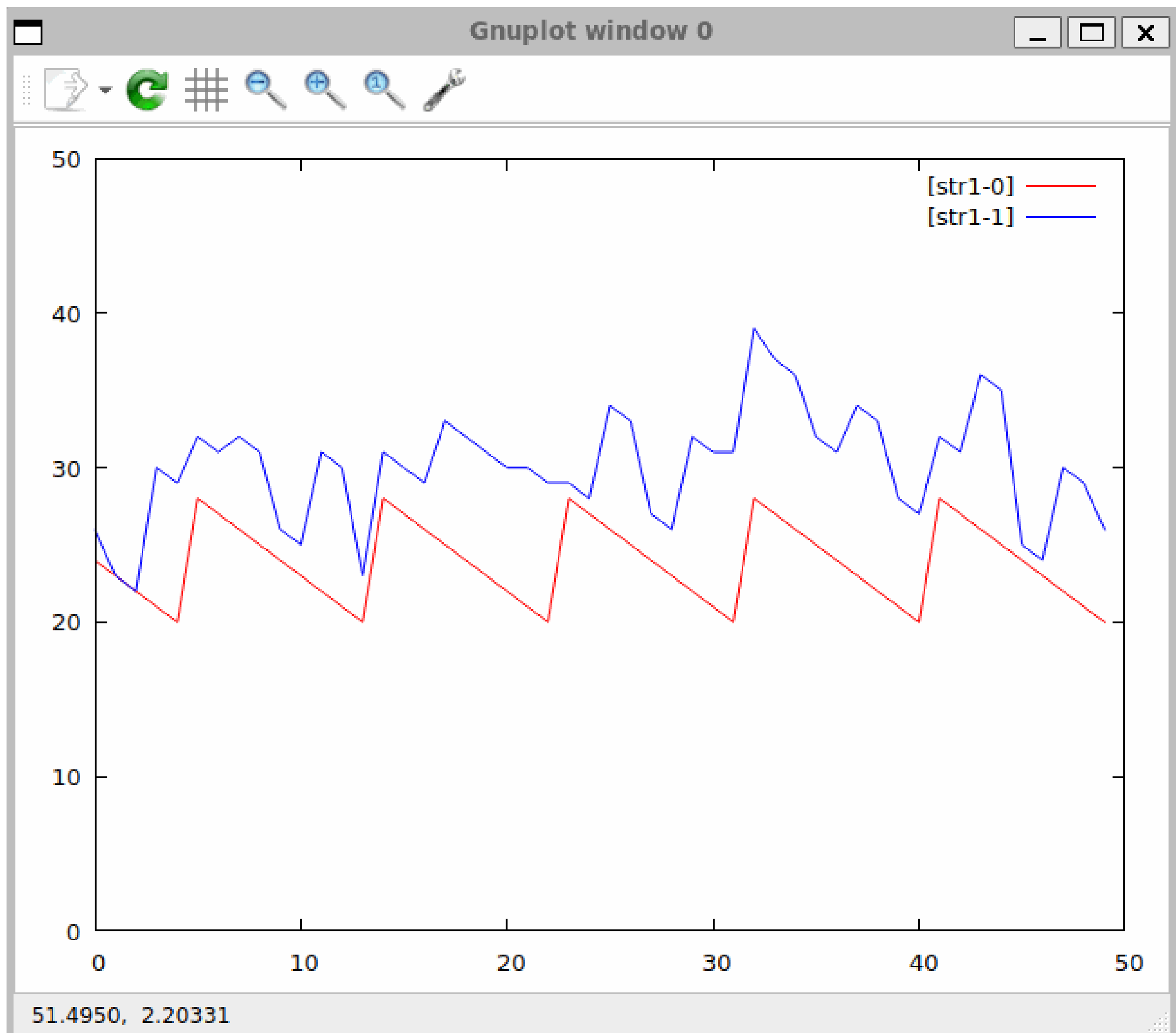
Transmisja danych odbędzie się przez sieć.

Jeśli chcemy zakończyć proces xretractor za pomocą polecenia xqry możemy wydać następujące polecenie:

```
$ xqry -k
kill sent to server
ok.
```

Po wydaniu tego polecenia proces xretractor zakończy swoje działanie i przerwie przetwarzane planów realizacji zapytań.

Zapis procesu prezentowany na ekranie przedstawia się następująco:

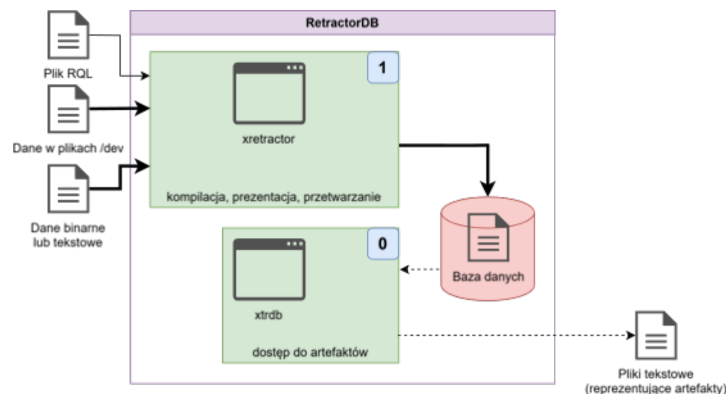

Gnuplot window 0
[str1-0]
[str1-1]
50
40
30
20
10
0
0
10
20
30
40
50
51.4950, 2.20331

Rozdział 34

Analiza artefaktów

Analizując szerzej potencjalne ścieżki danych na Rys. 10 ostatnią nieopisaną ścieżką jest ścieżka w której bierze udział narzędzie xtrdb.

W trakcie tworzenia systemu potrzebowałem narzędzia umożliwiającego dostęp do artefaktów w celu przeprowadzenia testów integracyjnych. W celu weryfikacji poprawności musiałem porównać wyniki przetwarzania na różnych etapach. Na Rys. 20 przedstawiono kompletny przepływ danych uwzględniający rolę narzędzia xtrdb.



Rys. 20. Przepływ danych w analizie artefaktów

W celu przedstawienia procesu analizy artefaktów konieczne jest uwzględnienie całego ciągu przetwarzania. Użyjemy tego samego zapytania co poprzednio. Uruchomimy jednak nasz proces przetwarzania danych w trochę inny sposób.

```
$ xretractor -m 10 query.rql
```

Tak wywołany proces przetwarzania zapytań zakończy swoją pracę po 10 cyklach przetwarzania. Parametr `-m` określa liczbę iteracji pętli głównej, nie liczbę sekund — czas działania zależy od interwału strumieni źródłowych. Dla strumieni z interwałem 0.1 s (10 Hz) oznacza to ~1 sekundę działania. Po zakończeniu działania i przejrzeniu katalogu w którym realizowaliśmy zapytanie powinniśmy zobaczyć następujące pliki:

```
$ ls -al
total 32
```

```
drwxr-xr-x  2 michal michal 4096 Oct  4 18:01 .
drwxr-xr-x 10 michal michal 4096 Oct  4 17:59 ..
-rw-r--r--  1 michal michal   51 Oct  4 18:01 core0.desc
-rw-r--r--  1 michal michal   43 Oct  4 18:01 core1.desc
-rw-r--r--  1 michal michal   27 Oct  4 17:59 datafile1.txt
-rw-r--r--  1 michal michal  180 Oct  4 18:00 query.rql
-rw-r--r--  1 michal michal   72 Oct  4 18:01 str1
-rw-r--r--  1 michal michal   34 Oct  4 18:01 str1.desc
```

Jak widać powstały trzy pliki .desc i jeden plik z artefaktami. Jeśli zajrzymy do pliku str1 to zobaczymy bardzo skromną zawartość:

```
$ hexdump str1
0000000 0014 0000 0015 0000 0015 0000 0016 0000
0000010 0016 0000 0017 0000 0017 0000 0018 0000
0000020 0018 0000 0019 0000 0019 0000 001a 0000
0000030 001a 0000 001b 0000 001b 0000 001c 0000
0000040 001c 0000 001d 0000
0000048
```

Wraz z plikiem artefaktu powstają pliki metadanych. Ich zawartość informuje o strukturze pliku.

```
$ cat str1.desc
{       INTEGER str1_0
        INTEGER str1_1
}
```

O wiele ciekawsze są opisy plików efemerydów. Pliki opisu danych efemerycznych wskazują na pliki w systemie Linux.

```
$ cat core0.desc
{       INTEGER a
        REF "datafile1.txt"
        TYPE TEXTSOURCE
}
$ cat core1.desc
{       BYTE a
        REF "/dev/urandom"
        TYPE DEVICE
}
```

Pliki opisu metadanych są tworzone automatycznie w momencie zarejestrowania w systemie RetractorDB obiektu. Należy pamiętać aby usunąć te deskryptory w przypadku zmodyfikowania pliku query.rql

Po uruchomieniu programu xtrdb w terminalu narzędzie wyświetli znak zachęty w postaci kropki (.). Znak ten to wyłącznie prompt — nie jest częścią polecenia. Można od razu rozpocząć komunikację z tym narzędziem. Przykład sesji:

```
$ xtrdb
.open str1
```

```
ok
.desc
{       INTEGER str1_0
        INTEGER str1_1
}
.list 1
{ str1_0:20 str1_1:21 }
.quit
```

Praca z tym narzędziem przypomina pracę z klasyczną, starą bazą danych dbase. Nie mamy tu jednak maszyny stanów, pętli czy warunków. Tylko odczyt i modyfikacje plików binarnych opisanych metadanymi.

Głównym celem tego narzędzia było wsparcie przy tworzeniu skryptów testowych. RetractorDB jest deterministyczny. W systemie nie występuje zjawisko wyścigu – dane, które trafią na wejście – zawsze powinny dać te same wyniki na wyjściu. Chyba że zmieszamy wyniki z danymi przypadkowymi jak w przedstawionym przykładzie.

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `issue113_meta_xtrdb`, `issue113_meta`, `issue113_null_txtsrc`, `Pattern5` opisanych w załączniku pt. Testy Integracyjne.

Bardzo użyteczną funkcją w tym narzędziu jest funkcja `list` oraz `rlist`. Listująca początkowe elementy pliku lub końcowe elementy pliku — uwzględniając strukturę opisaną w metadanych.

```
.list 4
{ str1_0:20 str1_1:21 }
{ str1_0:21 str1_1:22 }
{ str1_0:22 str1_1:23 }
{ str1_0:23 str1_1:24 }
.rlist 4
{ str1_0:28 str1_1:29 }
{ str1_0:27 str1_1:28 }
{ str1_0:26 str1_1:27 }
{ str1_0:25 str1_1:26 }
```

Zachęcam do eksperymentów i przejrzenia źródeł tego narzędzia. Jest to jeden z mniej skomplikowanych a bardzo użytecznych elementów systemu RetractorDB.

34.1 Inspekcja metadanych null/gap

Każdy artefakt ma skojarzony plik indeksu `.meta` opisany szczegółowo w rozdziale dotyczącym formatu zapisu. Zawartość tego pliku można obejrzeć bezpośrednio w xtrdb poleceniem `meta`:

```
.open str1
ok
```

```
.meta
record 0: count=9 gap=false nullBitset=00
```

Wpis `gap=false` oznacza brak przerwy w danych, `nullBitset` informuje które pola zawierają wartości null (po jednym bicie na pole). Dane bez żadnych braków tworzą jeden wpis `count=N` gdzie N to łączna liczba rekordów.

Rozdział 35

Podsumowanie

W podsumowaniu należy wskazać na przekazy w rozdziale zakres wiedzy. Tutaj chciałem przedstawić jak poszczególne elementy systemu możemy uruchamiać, jak wyglądają ciągi poleceń w oparciu o które budujemy dalsze funkcjonalności z wykorzystaniem systemu RetractorDB.

Starałem się zredukować ilość potencjalnych poleceń do minimum. Na chwile obecną efektywnie zredukowałem zbiór do 3 poleceń. Wydaje mi się tak zaprojektowany system będzie maksymalnie użyteczny i w miarę efektywny. Osobną kwestią jest komplikacja. Samo tłumaczenie procesu przetwarzania, nowej algebry i dlaczego znak plus nie oznacza plus – jest problematyczne. Mam jednak nadzieję że po przeskoczeniu pewnej bariery poznania – reszta będzie oczywista. Podjęte decyzje były efektem przemyśleń, prób i błędów. Chcę podkreślić że zwyczajnie po ludzku nie znalazłem lepszej metody.

Rozdział 36

Kompilacja zapytań

Uważny czytelnik zauważy zapewne, że w przedstawionych w poprzednim rozdziale skompilowanych planach realizacji zapytań pewne wartości nie odpowiadają temu, co zostało napisane w zapytaniu.

Kompilator prowadząc proces budowania planu zapytania prowadzi proces autonomicznie. Wydaje się czasem, że prosząc o jedno - dostaje się coś innego – na pierwszy rzut oka jest to zachowanie zupełnie nieoczywiste. I jako użytkownik nie mam zasadniczo na to wpływu. Co ciekawe efekt zapytania odpowiada temu o co prosiłem w zapytaniu. Być może poprawny tytuł tego rozdziału powinien brzmieć: Dlaczego kompilator robi po swojemu i do tego wie lepiej?

W tym rozdziale chcę wyjaśnić, jak rozwiązałem problemy syntaktyczne, które napotkałem w trakcie tworzenia języka zapytań.

36.1 Wejście i wyjście kompilatora

Plik `.rql`

Wejście kompilatora — tekst w języku RetractorQL zawierający dyrektywy `DECLARE` i `SELECT`. Parser ANTLR4 czyta plik sekwencyjnie; odwołanie do strumienia niezdefiniowanego wcześniej w pliku kończy się błędem kompilacji. {% endstep %}

Parser ANTLR4 → `qTree`

Parser buduje wewnętrzną reprezentację `qTree`: topologicznie posortowany `std::vector<query>`. Każdy element opisuje jeden strumień — jego schemat pól, sekwencję instrukcji stosu, zależności od innych strumieni i interwał czasowy (delta).

10 etapów kompilacji

`qTree` przechodzi przez łańcuch przekształceń: od rozbicia wyrażeń FROM na operacje dwuargumentowe, przez wyznaczenie delt i offsetów bajtowych, aż po weryfikację semantyczną i obliczenie rozmiarów buforów. Każdy etap zakłada sukces poprzedniego.

Plan wykonania → `dataModel`

Na wyjściu kompilacji każde zapytanie w `qTree` ma wyznaczone: schemat pól z typami i offsetami, deltę, rozmiary buforów oraz gotową sekwencję instrukcji. Ten plan przejmuje `dataModel` i realizuje go cyklicznie w czasie rzeczywistym.

Flaga `-c` zatrzymuje `xretractor` po tym kroku i drukuje plan na standardowe wyjście — bez uruchamiania przetwarzania.

36.2 Przegląd poruszonych w rozdziale tematów

Rozdział zbudowany jest zgodnie z kolejnością etapów kompilatora — od opisu struktury danych i łańcucha etapów, przez poszczególne przekształcenia, aż po obsługę błędów.

Przebiegi kompilacji opisuje cały łańcuch etapów funkcji `compiler::compile()`. Kompilacja to nie jeden krok — to sekwencja dziesięciu kolejnych przekształceń wewnętrznej reprezentacji `qTree`, od sprowadzenia wyrażeń FROM do postaci dwuargumentowej, przez wyznaczanie interwałów i offsetów pól, aż po weryfikację semantyczną i alokację buforów. Każdy etap zakłada sukces poprzedniego i zwraca komunikat błędu, gdy warunki nie są spełnione.

Budowa drzewa zależności opisuje strukturę DAG powstającego w trakcie kompilacji — fundament, na którym opierają się wszystkie etapy. Korzeniami są deklaracje efemerydów (źródła zewnętrzne), wewnątrz grafu leżą substraty pośrednie, a liśćmi są artefakty. Flaga `-d` generuje wyjście w formacie DOT, które `graphviz` zamienia w wizualny graf zależności. Kolejność zapytań w pliku `.rql` ma znaczenie — odwołanie do niezdefiniowanego jeszcze strumienia kończy się błędem.

Substraty wyjaśnia etap `extractIntermediateStreams` — pierwszy krok kompilacji. Gdy wyrażenie FROM zawiera więcej niż dwa argumenty (np. `(core0#core1)+core2`, `core0+core1+core2`), kompilator rozbija je na operacje dwuargumentowe i tworzy nazwane substraty. Późniejszy etap `deduplicateSubstrats` wykrywa, gdy substrat jest strukturalnie identyczny z zapytaniem użytkownika, i zastępuje odwołania — unikając powielania obliczeń.

Rozwijanie symbolu * wyjaśnia etap `expandSchemaWildcards`. Symbol `*` w klauzuli SELECT zostaje zastąpiony pełną listą pól wynikających ze schematu strumienia źródłowego — w tym polami pochodzącymi z operacji sumy strumieni. Przykład pokazuje, jak typy pól decydują o tym, które pole trafia na które miejsce w schemacie wynikowym.

Rozwiązywanie interwałów opisuje etap `resolveStreamIntervals`. Kompilator wyznacza deltę każdego strumienia wynikowego z równań algebry strumieniowej: dla operatora `+` delta to minimum wejść, dla `#` — średnia harmoniczna, dla `@(step, window)` — pochodna rozmiaru okna. Algorytm działa iteracyjnie — każda runda rozwiązuje co najmniej jeden strumień, aż wszystkie delty są znane.

Wykrywanie pętli opisuje mechanizm wbudowany w etap `resolveStreamIntervals`. Jeśli liczba nierozwiązanych strumieni przestaje maleć, żaden strumień nie może uzyskać delty — znak, że graf zależności zawiera cykl. Kompilacja kończy się błędem

`"Circular dependency in stream definitions"`. Rozdział zawiera przykład cyklicznego zapytania i sposób jego naprawy.

Aliasowanie opisuje etap `resolveFieldReferences`. Do pola wynikowego można odwoływać się zarówno przez indeks w schemacie sumarycznym (`str1[1]`), jak i przez nazwę strumienia źródłowego z lokalnym indeksem (`core1[0]`). Kompilator tłumaczy obie formy na tę samą pozycję w buforze wynikowym.

Przetwarzanie symbolu _ opisuje etap `expandIndexWildcards` — cukier syntaktyczny do równoległych operacji na parach pól. Symbol `_` w indeksie powoduje powielenie formuły dla wszystkich par pól ze schematów obu argumentów — `core0[_] * core1[_]` przy dwupólowych schematach generuje dwa pola mnożące odpowiadające pary. Zastosowanie: budowa zapytań filtrów sygnałowych.

Równanie typów w górę definiuje reguły promocji typów obowiązujące przez cały łańcuch kompilacji. Wynik działania `BYTE * INTEGER` ma typ `INTEGER` — kompilator wyznacza typ pola wyjściowego statycznie, zanim dane zostaną przetworzone. Opisano też kompletną hierarchię typów obsługiwanych przez RetractorDB.

Debugowanie kompilacji zbiera w jednym miejscu narzędzia diagnostyczne: flaga `-c` do inspekcji planu, pipeline `-c -d -f -s` do wizualizacji grafu przez `graphviz`, tablicę znaczeń instrukcji planu (PUSH_ID, PUSH_STREAM, STREAM_ADD, …) oraz katalog typowych błędów kompilacji z ich przyczynami i sposobem naprawy.

Rozdział 37

Przebiegi kompilacji

Kompilacja zapytań w RetractorDB przebiega w wielu etapach. Każdy etap transformuje wewnętrzną reprezentację zapytań — drzewo `qTree` — i przekazuje wynik do następnego. Kolejność jest ściśle ustalona: każdy etap zakłada, że poprzedni zakończył się sukcesem.

`qTree` to topologicznie posortowany `std::vector<query>` — centralna struktura danych kompilatora i executora. Każdy element wektora odpowiada jednemu zapytaniu (`SELECT` lub `DECLARE`) i przechowuje jego schemat pól, sekwencję instrukcji stosu, interwał czasowy oraz referencje do strumieni źródłowych. Sortowanie topologiczne gwarantuje, że strumień źródłowy zawsze poprzedza strumień wynikowy — etapy mogą przetwarzać `qTree` liniowo, bez nawrotów.

37.1 Przykład śledzący

Przez cały rozdział śledzimy jedno zapytanie — `query.rql` — przez kolejne etapy:

```
DECLARE a BYTE, b INTEGER
STREAM core0, 0.1
FILE 'sensor_a.txt'

DECLARE c INTEGER, d FLOAT
STREAM core1, 0.2
FILE 'sensor_b.txt'

DECLARE e INTEGER
STREAM core2, 0.3
FILE 'sensor_c.txt'

SELECT *
STREAM merged
FROM core0 + core1

SELECT merged[0], merged[2], core0[0], core1[0]
```

```
STREAM result
FROM merged
```

Po przejściu przez wszystkie etapy `xretractor -c query.rql` drukuje:

```
merged(1/10)
        :- PUSH_STREAM(core0)
        :- PUSH_STREAM(core1)
        :- STREAM_ADD
        core0_0: BYTE
                PUSH_ID(merged[0])
        core0_1: INTEGER
                PUSH_ID(merged[1])
        core1_2: INTEGER
                PUSH_ID(merged[2])
        core1_3: FLOAT
                PUSH_ID(merged[3])
result(1/10)
        :- PUSH_STREAM(merged)
        result_0: BYTE
                PUSH_ID(merged[0])
        result_1: INTEGER
                PUSH_ID(merged[2])
        result_2: BYTE
                PUSH_ID(merged[0])
        result_3: INTEGER
                PUSH_ID(merged[2])
core0(1/10)     sensor_a.txt
        a: BYTE
        b: INTEGER
core1(1/5)      sensor_b.txt
        c: INTEGER
        d: FLOAT
core2(3/10)     sensor_c.txt
        e: INTEGER
```

Podrozdziały o substratach i symbolu _ używają rozszerzonych wariantów tego samego zestawu deklaracji. Jak interpretować każdy element tego planu — patrz Debugowanie kompilacji.

37.2 Łańcuch etapów

Łańcuch etapów definiuje funkcja `compiler::compile()`:

{% stepper %} {% step %} #### extractIntermediateStreams

Sprowadza każde wyrażenie FROM do postaci co najwyżej dwuargumentowej. Złożone wyrażenia jak `(core0#core1)+core2` oraz zapisy łańcuchowe bez nawiasów (`core0+core1+core2`, `core0#core1#core2`) wymagają pośrednich strumieni. Etap tworzy automatycznie substraty — patrz Substraty. {% endstep %}

{% step %} #### expandSchemaWildcards

Rozwija symbol * w klauzuli SELECT. Zastępuje go listą pól wynikających z schematu strumienia źródłowego — patrz Rozwijanie symbolu *. {% endstep %}

{% step %} #### resolveStreamIntervals (← tu wykrywane są pętle)

Wyznacza interwał czasowy (delta) każdego strumienia na podstawie operatorów algebraicznych i interwałów strumieni wejściowych. Algorytm iteracyjny — w każdej rundzie rozwiązuje tyle strumieni, ile jest możliwe. Wykrywa cykliczne zależności zatrzymując się, gdy liczba nierozwiązanych strumieni przestaje maleć — patrz Rozwiązywanie interwałów i Wykrywanie pętli. {% endstep %}

{% step %} #### deduplicateSubstrats

Optymalizacja: jeśli dwa zapytania korzystają z tej samej operacji pośredniej (np. `core0#core1`), etap wskazuje drugie zapytanie na substrat utworzony przez pierwsze. Unika powielania obliczeń — patrz przykład w Substraty. {% endstep %}

{% step %} #### resolveFieldReferences

Przekształca odwołania do pól ze schematów źródłowych na indeksy w schemacie wynikowym. Obsługuje aliasowanie — `core0[0]` zamienia na `str1[0]` itp. — patrz Aliasowanie. {% endstep %}

{% step %} #### expandIndexWildcards

Rozwija symbol _ w indeksach pól. Powielenie formuły dla wszystkich pasujących par pól ze schematów argumentów — patrz Przetwarzanie symbolu _. {% endstep %}

{% step %} #### localizeFieldOffsets

Przelicza referencje do pól (`b[x]`, `c[y]`) na indeksy w spłaszczonym schemacie wynikowym (`merged[z]`). Dla ADD indeks wynika z sumy liczności pól poprzedzających strumieni; dla HASH każde pole otrzymuje indeks 0 (schemat jednoargumentowy). Etap uwzględnia nie tylko źródła bezpośrednie, ale także źródła przechodnie ukryte za automatycznymi substratami. {% endstep %}

{% step %} #### computeRequiredCapacities

Oblicza wymagane pojemności buforów dla każdego strumienia na podstawie rozmiarów schematów i wymagań okien czasowych. {% endstep %}

{% step %} #### validateConstraints

Weryfikuje poprawność semantyczną skompilowanego planu: zgodność typów, rozmiary okien, dostępność źródeł danych. {% endstep %}

{% step %} #### applyCapacitiesToStreams

Aplikuje obliczone pojemności do obiektów strumieni. Po tym etapie plan jest gotowy do wykonania przez `dataModel`. {% endstep %} {% endstepper %}

Każdy etap zwraca `"OK"` lub komunikat błędu — wówczas kompilacja się zatrzymuje.

Rozdział 38

Budowa drzewa zależności

Drzewo zależności to plan realizacji zapytań w postaci grafu skierowanego. Jest to struktura danych, która budowana jest w trakcie kompilacji oraz modyfikowana w trakcie dodawania zapytań AdHoc. Korzeniami tego grafu są deklaracje efemerydów. Wszelkiej postaci deklaracje tworzące obiekty zewnętrzne – tzw. Źródła danych. Wewnątrz grafu występują artefakty i substraty. Na końcu łańcucha przetwarzania znajdują się artefakty – jako wyniki końcowe łańcucha.

Taka konstrukcja to graf skierowany. Graf, który posiada wiele korzeni i wiele wierzchołków końcowych. Wewnątrz grafu znajdują się węzły łączące. Każdy węzeł znajduje się na drodze od korzenia do wierzchołka końcowego. Najlepiej to zwizualizuje przykład.

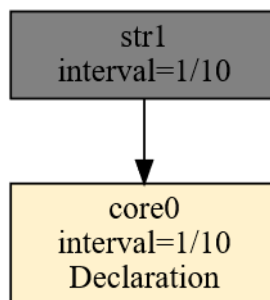
Na początku rozważmy następujące trywialne zapytanie:

```
DECLARE a UINT STREAM core0, 0.1 FILE 'datafile1.txt'
SELECT str1[0] STREAM str1 FROM core0
```

Graf, w którym uwypuklone zostaną dependencje pomiędzy poszczególnymi obiektami uzyskamy w następujący sposób (Rys. 21):

```
$ xretractor -c query5.rql -d > out.dot && dot -Tsvg out.dot -o out.svg
```

Pełny opis flag `-d` `-f` `-s` i interpretacja wyjścia — patrz Debugowanie kompilacji.

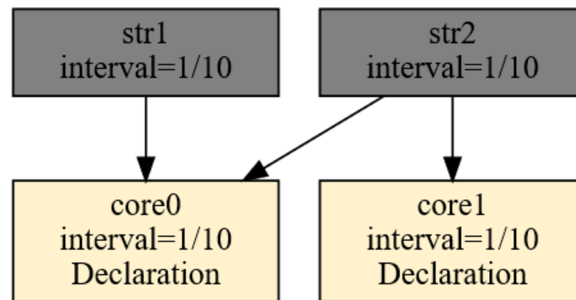


Rys. 21. Dependencja efemeryd-artefakt

Skomplikujmy trochę ten graf dodając dwie deklaracje efemerydów i dodatkowy artefakt.

```
DECLARE a UINT STREAM core0, 0.1 FILE 'datafile1.txt'
DECLARE a UINT STREAM core1, 0.1 FILE 'datafile2.txt'
SELECT str1[0] STREAM str1 FROM core0
SELECT str2[0] STREAM str2 FROM core0 + core1
```

Graf zależności dla powyższego zestawu zapytań prezentuje się następująco (Rys. 22):

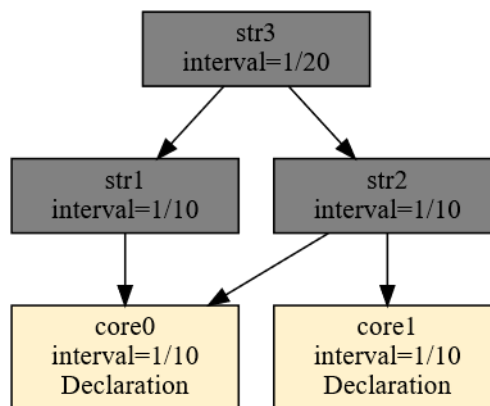


Rys. 22. Dependencja efemerydy-artefakty

Zbudujmy dodatkowy węzeł zależny od artefaktów. Najprościej dodać następujące zapytanie na końcu:

```
SELECT str3[0] STREAM str3 FROM str1#str2
```

Graf zmieni swoją postać:



Rys. 23. Dependencja efemerydy-artefakty-artefakty

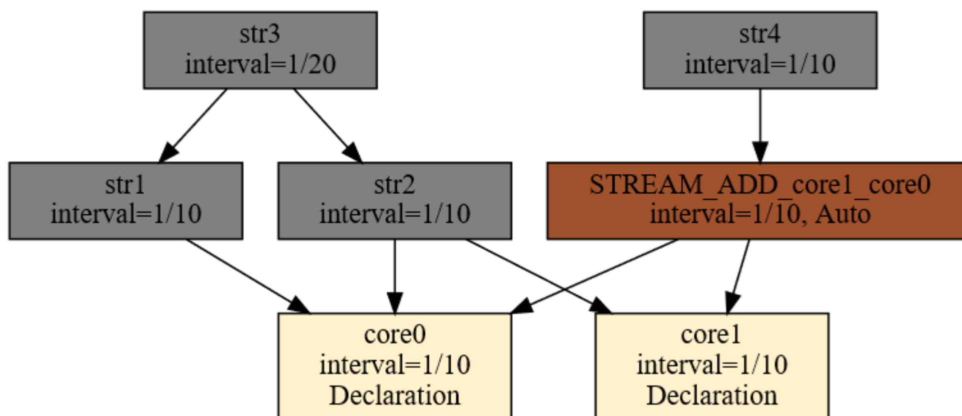
Jak widać na Rys. 23 strumień str3 nie jest zależny bezpośrednio od danych dostarczanych przez strumienie core0 i core1. Zapytania tworzą graf zależności a kolejności ich wywoływania jest uporządkowana. Wartość interwału w strumieniach rośnie w kierunku korzeni. Wzrost w kierunku korzenia wynika z równań wyznaczających interwały opracowanej algebry.

Proszę zwrócić uwagę, że zapytania w pliku rql przetwarzane są sekwencyjnie. Próba odwołania się w zapytaniu do obiektu, który nie jest jeszcze zdefiniowany, skończy się błędem kompilacji.

W przypadku dołączenia do drzewa zależności następującego zapytania wytworzymy dodatkowy substrat.

```
SELECT str4[0] STREAM str4 FROM (core1+core0)>2
```

Tak dołączone zapytanie spowoduje modyfikację drzewa zależności w sposób przedstawiony na Rys. 24.



Rys. 24. Dependencja z substratem

Substrat został oznaczony innym kolorem oraz oznaczeniem Auto znajdującym się obok interwału czasowego.

Graf zależności musi być acyklicznym grafem skierowanym (DAG). Próba zdefiniowania strumienia odwołującego się do własnych wyników tworzy cykl i kończy się błędem kompilacji. Mechanizm wykrywania opisany jest w rozdziale Wykrywanie pętli w kompilacji.

Uwaga

Opisana funkcjonalność ma pokrycie w teście: `subquery` opisanym w załączniku pt. Testy Integracyjne.

Rozdział 39

Substraty

O substratach, efemerydach i artefaktach wspomniałem w rozdziale dotyczącym architektury systemu. W tym przypadku przedstawię przykład.

Na początek chciałbym zwrócić uwagę na pewną własność wprowadzonych wyrażeń algebraicznych. W praktyce możemy zapisać dowolne wyrażenie, skompilować i przedstawić wzór na operacje na poszczególnych elementach serii czasowych umożliwiających uzyskanie pożądanego wyniku.

W praktyce w systemie realizuję wyłącznie operacje jedno lub dwuargumentowe. Przykładem operacji jednoargumentowych to przesunięcie w czasie lub operacja Agse. Tam argumentem jest tylko jeden strumień danych. Reszta operacji to operacje na dwóch strumieniach danych. W trakcie kompilacji wszystkie wyrażenia algebraiczne rozbijane są na takie, które mają dwa argumenty.

Parser akceptuje zarówno formę z nawiasami, jak i łańcuchy bez nawiasów, np. `s1+s2+s3`, `s1#s2#s3` oraz `s1+s2+s3+s4`. Taki zapis jest następnie redukowany do sekwencji operacji dwuargumentowych z automatycznymi substratami pośrednimi.

Przykład używa kanonicznych deklaracji z całego rozdziału — trzy strumienie o różnych typach i interwałach:

```
DECLARE a BYTE, b INTEGER
STREAM core0, 0.1
FILE 'sensor_a.txt'

DECLARE c INTEGER, d FLOAT
STREAM core1, 0.2
FILE 'sensor_b.txt'

DECLARE e INTEGER
STREAM core2, 0.3
FILE 'sensor_c.txt'

SELECT merged[0]
STREAM merged
```

```
FROM (core0 # core1) + core2
```

Kompilacja:

```
$ xretractor -c query.rql
STREAM_HASH_core0_core1(1/15)
        :- PUSH_STREAM(core0)
        :- PUSH_STREAM(core1)
        :- STREAM_HASH
        a: BYTE
                PUSH_ID(STREAM_HASH_core0_core1[0])
        b: INTEGER
                PUSH_ID(STREAM_HASH_core0_core1[1])
        c: INTEGER
                PUSH_ID(STREAM_HASH_core0_core1[2])
        d: FLOAT
                PUSH_ID(STREAM_HASH_core0_core1[3])
merged(1/15)
        :- PUSH_STREAM(STREAM_HASH_core0_core1)
        :- PUSH_STREAM(core2)
        :- STREAM_ADD
        merged_0: BYTE
                PUSH_ID(merged[0])
core0(1/10)     sensor_a.txt
        a: BYTE
        b: INTEGER
core1(1/5)      sensor_b.txt
        c: INTEGER
        d: FLOAT
core2(3/10)     sensor_c.txt
        e: INTEGER
```

Pojawił się niezapowiedziany strumień `STREAM_HASH_core0_core1` — to właśnie substrat. Kompilator rozbił `(core0 # core1) + core2` na dwie operacje dwuargumentowe i wstawił pośredni strumień. Delta substratu: $\Delta = (1/10 \cdot 1/5) / (1/10 + 1/5) = 1/15$.

Co się stanie po dołączeniu zapytania:

```
SELECT merged2[0] STREAM merged2 FROM (core0 # core1) > 2
```

Do planu dołączone zostanie tylko jedno nowe zapytanie:

```
merged2(1/15)
        :- PUSH_STREAM(STREAM_HASH_core0_core1)
        :- STREAM_TIMEMOVE(2)
        merged2_0: BYTE
                PUSH_ID(merged2[0])
```

Zastanawiasz się pewnie dlaczego tylko jedno a nie ponownie dwa? Odpowiedź to optymalizacja. Korzystamy z pośrednich wyników poprzedniego. To jedna z nieoczekiwanych korzyści zastosowania RetractorDB.

Jest jeszcze jedna istotna rzecz o której należy wspomnieć w tym punkcie. Istnieje dyrektywa SUBSTRAT, której argumentem jest ciąg znaków ujęty w apostrofy. Można użyć następujących typów 'memory', 'default', 'direct', 'posix', 'posixshd', 'generic', 'device', 'textsource'. Pełny opis każdego typu znajdziesz w rozdziale Typy STORAGE. Domyślny typ 'default' spowoduje, że substraty będą materializować się w całości na dysku. To nie jest oczekiwana wartość w systemie produkcyjnym, ale oczekiwana w trakcie rozwoju i debugowania. Typ użyteczny to 'memory'. Substraty tego typu lądują tylko w pamięci. Ich dane nigdy nie lądują na dysku – wszystko odbywa się w pamięci, danych jest tylko tyle ile jest wymaganych do realizacji zapytań. Reszta typów na chwilę obecną jest nieprzetestowana i znajduje się w fazie rozwojowej.

Dodanie zapytania o tych samych operacjach, ale innej nazwie może spowodować deduplikację substratów. Jeśli program, delta i schemat są równoważne, kompilator przepnie odwołania `PUSH_STREAM` na istniejący strumień i usunie duplikat.

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `issue96_no_substrat_reduction`, `issue96_substrat_reference` opisanych w załączniku pt. Testy Integracyjne.

39.1 Redukcja substratów

Kompilator realizuje optymalizację zwaną **redukcją substratów** (funkcja `deduplicateSubstrats`). Polega ona na tym, że jeśli użytkownik zdefiniował zapytanie strukturalnie identyczne z wygenerowanym substratem, substrat jest usuwany z planu, a jego odwołania zastępowane są nazwą zapytania użytkownika.

Warunki redukcji

Redukcja substratu do zapytania użytkownika następuje wtedy i tylko wtedy, gdy spełnione są jednocześnie trzy warunki:

1. **Ten sam schemat** — typy i nazwy pól wyjściowych są identyczne.
2. **Ta sama delta** — częstotliwość próbkowania strumieni jest taka sama.
3. **Te same operacje przetwarzania** — sekwencja instrukcji `PUSH_STREAM` / `STREAM_TIMEMOVE` / `STREAM_HASH` itp. jest identyczna.

Przykład redukcji

Rozważmy zapytanie z kanonicznymi deklaracjami:

```
DECLARE a BYTE, b INTEGER   STREAM core0, 0.1 FILE 'sensor_a.txt'
DECLARE c INTEGER, d FLOAT  STREAM core1, 0.2 FILE 'sensor_b.txt'

SELECT merged[0] STREAM merged FROM (core0 > 2) + core1
SELECT shifted[0] STREAM shifted FROM core0 > 2
```

Bez redukcji kompilator wygenerowałby trzy strumienie: substrat `STREAM_TIMEMOVE_core0`, `merged` i `shifted`. Substrat i `shifted` mają identyczną strukturę — ten sam strumień

źródłowy `core0` i tę samą operację `>2`. Po redukcji substrat jest usuwany, a odwołanie `PUSH_STREAM(STREAM_TIMEMOVE_core0)` w `merged` zostaje zastąpione przez `PUSH_STREAM(shifted)`:

```
merged(1/10)
        :- PUSH_STREAM(shifted)
        :- PUSH_STREAM(core1)
        :- STREAM_ADD
        merged_0: BYTE
                PUSH_ID(merged[0])
shifted(1/10)
        :- PUSH_STREAM(core0)
        :- STREAM_TIMEMOVE(2)
        shifted_0: BYTE
                PUSH_ID(shifted[0])
core0(1/10)     sensor_a.txt
        a: BYTE
        b: INTEGER
core1(1/5)      sensor_b.txt
        c: INTEGER
        d: FLOAT
```

Ważne ograniczenie: tylko substraty są redukowane

Redukcja dotyczy wyłącznie substratów wygenerowanych przez kompilator (`isSubstrat = true`). Zapytania zdefiniowane jawnie przez użytkownika **nigdy** nie są redukowane, nawet jeśli dwa z nich mają identyczną strukturę.

Przykład — dwa zapytania użytkownika o tej samej operacji:

```
DECLARE a BYTE, b INTEGER   STREAM core0, 0.1 FILE 'sensor_a.txt'

SELECT shifted1[0] STREAM shifted1 FROM core0 > 2
SELECT shifted2[0] STREAM shifted2 FROM core0 > 2
```

Wynik kompilacji zachowa oba strumienie bez żadnej redukcji:

```
shifted1(1/10)
        :- PUSH_STREAM(core0)
        :- STREAM_TIMEMOVE(2)
        shifted1_0: BYTE
                PUSH_ID(shifted1[0])
shifted2(1/10)
        :- PUSH_STREAM(core0)
        :- STREAM_TIMEMOVE(2)
        shifted2_0: BYTE
                PUSH_ID(shifted2[0])
core0(1/10)     sensor_a.txt
        a: BYTE
        b: INTEGER
```

Semantyczna decyzja jest tu celowa: użytkownik zadeklarował dwa odrębne strumienie wynikowe i oba mają prawo istnieć niezależnie w planie wykonania.

39.2 Eliminacja duplikatów substratów

Gdy kilka zapytań korzysta z tej samej operacji strumieniowej – np. `core0 + core1` – faza ekstrakcji substratów (`extractIntermediateStreams`) tworzy dla każdego z nich osobny substrat. Bez kolejnej fazy naprawczej w grafie powstawałyby równoległe, identyczne węzły pośrednie obliczające dokładnie tę samą wartość.

Kiedy substrat jest tworzony

Substrat generowany jest dla każdego zapytania, którego program zawiera więcej niż jeden operator strumieniowy. Dotyczy to operatorów: `STREAM_ADD`, `STREAM_SUBTRACT`, `STREAM_HASH`, `STREAM_DEHASH_DIV`, `STREAM_DEHASH_MOD`, `STREAM_TIMEMOVE`, `STREAM_AGSE`. Warunek sprawdza funkcja `query::isReductionRequired()`.

Nowo powstałemu substratowi nadawana jest nazwa zbudowana z symbolu operacji i nazw operandów, np. `STREAM_ADD_core1_core0` (funkcja `composeStreamName` w `compiler.cpp`). W programie zapytania macierzystego token operatora zastępowany jest tokenem `PUSH_STREAM` wskazującym na ten substrat.

Algorytm deduplikacji

Po ekstrakcji substratów i wyznaczeniu interwałów czasowych kompilator uruchamia krok `deduplicateSubstrats()`. Algorytm działa iteracyjnie – pętla `while(changed)` powtarza przeszukiwanie aż do momentu, gdy żadna para duplikatów nie zostanie już znaleziona.

W każdym przebiegu dla każdej pary substratów `(it, it2)` sprawdzane są kolejno pięć warunków równoważności:

1. **Interwał czasowy** – `it->rInterval == it2->rInterval`
2. **Długość programu** – liczba tokenów w `lProgram` musi być identyczna
3. **Długość schematu** – liczba pól w `lSchema` musi być identyczna
4. **Zawartość programu** – każdy token porównywany jest według typu polecenia (`getCommandID()`) i wartości parametru (`getVT()`)
5. **Zawartość schematu** – każde pole porównywane jest według typu (`rtype`), rozmiaru w bajtach (`rlen`) i liczności (`rarray`)

Jeśli wszystkie warunki są spełnione, substrat `it` uznawany jest za duplikat substratu `it2`. Kompilator przechodzi przez cały `coreInstance` i we wszystkich tokenach `PUSH_STREAM` odnoszących się do starej nazwy (`it->id`) podstawia nową nazwę (`it2->id`). Następnie duplikat jest usuwany z listy zapytań (`coreInstance.erase(it)`), a pętla startuje od początku.

Miejsce w potoku kompilacji

Deduplikacja jest czwartym krokiem ośmiofazowego potoku (funkcja `compiler::compile()`):

```
1. extractIntermediateStreams   - wyodrębnienie substratów
2. expandSchemaWildcards        - rozwinięcie symboli wieloznacznych w schematach
3. resolveStreamIntervals       - obliczenie interwałów czasowych
4. deduplicateSubstrats         - eliminacja duplikatów  ← ten krok
5. resolveFieldReferences       - rozwiązanie referencji do pól
6. expandIndexWildcards         - rozwinięcie indeksów wieloznacznych
7. localizeFieldOffsets         - wyznaczenie przesunięć pól
8. validateConstraints / applyCapacities
```

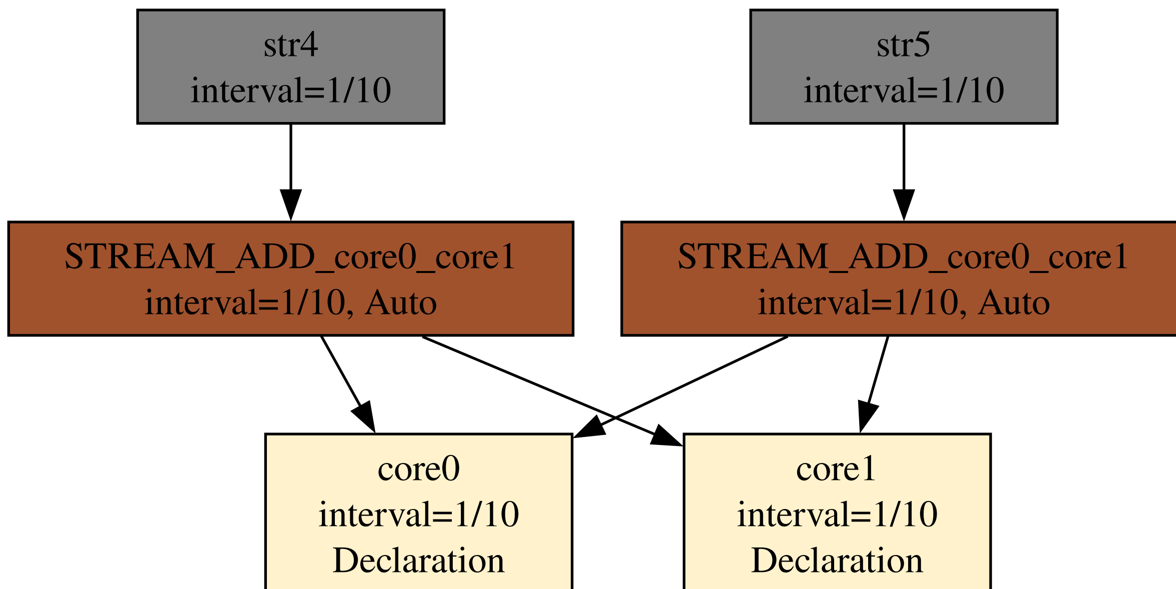
Deduplikacja musi nastąpić po kroku 3, ponieważ porównanie interwałów jest jednym z kryteriów równoważności – substraty o różnych interwałach nie są identyczne nawet jeśli realizują tę samą operację algebraiczną.

Efekt w grafie zależności

Rozważmy zapytania:

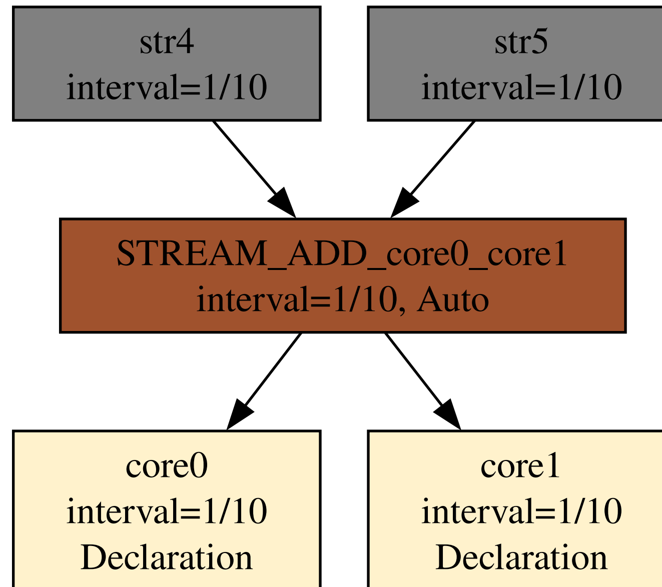
```
DECLARE a UINT STREAM core0, 0.1 FILE 'datafile1.txt'
DECLARE a UINT STREAM core1, 0.1 FILE 'datafile2.txt'
SELECT str4[0] STREAM str4 FROM (core0+core1)>2
SELECT str5[0] STREAM str5 FROM (core0+core1)>3
```

Oba zapytania wymagają uprzedniego obliczenia sumy `core0+core1`. Faza `extractIntermediateStream` tworzy osobny substrat dla każdego zapytania, co daje dwa identyczne węzły pośrednie w grafie (Rys. 25):



Rys. 25. Graf przed deduplikacją — dwa identyczne substraty STREAM_ADD_core0_core1

Po uruchomieniu `deduplicateSubstrats()` jeden z duplikatów jest usuwany, a wszystkie odwołania `PUSH_STREAM` przepinane są do ocalałego węzła. W grafie pozostaje jeden wspólny substrat (Rys. 26):



Rys. 26. Graf po deduplikacji — jeden wspólny substrat, wygenerowany poleceniem: xretractor dedup_after.rql -c -d

Graf po deduplikacji to dokładnie to, co zwraca `xretractor -c -d` — kompilator zawsze prezentuje wynik po wszystkich fazach optymalizacji.

39.3 Wchłonięcie substratu przez jawny strumień

Pętla wewnętrzna w `deduplicateSubstrats()` nie sprawdza flagi `isSubstrat` dla kandydata `it2` — sprawdzenie to istnieje tylko w pętli zewnętrznej. Oznacza to, że substrat automatyczny może zostać wchłonięty nie tylko przez inny substrat, ale przez **dowolny strumień o identycznym programie i schemacie** — w tym przez strumień zdefiniowany jawnie przez użytkownika.

Rozważmy zapytanie zawierające wyłącznie złożone wyrażenie:

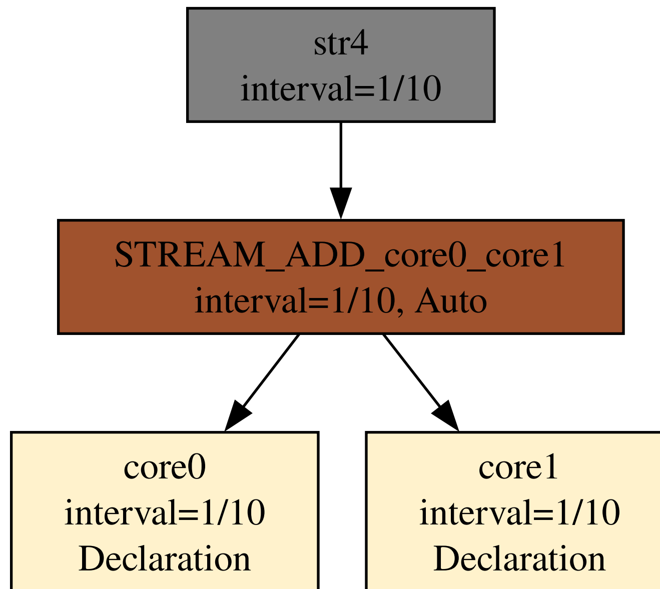
```
DECLARE a UINT STREAM core0, 0.1 FILE 'datafile1.txt'
DECLARE a UINT STREAM core1, 0.1 FILE 'datafile2.txt'
SELECT str4[0] STREAM str4 FROM (core0+core1)>2
```

`extractIntermediateStreams` wyodrębnia tutaj substrat `STREAM_ADD_core0_core1` dla wyrażenia `core0+core1`. Artefakt `str4` zależy od niego (Rys. 27):

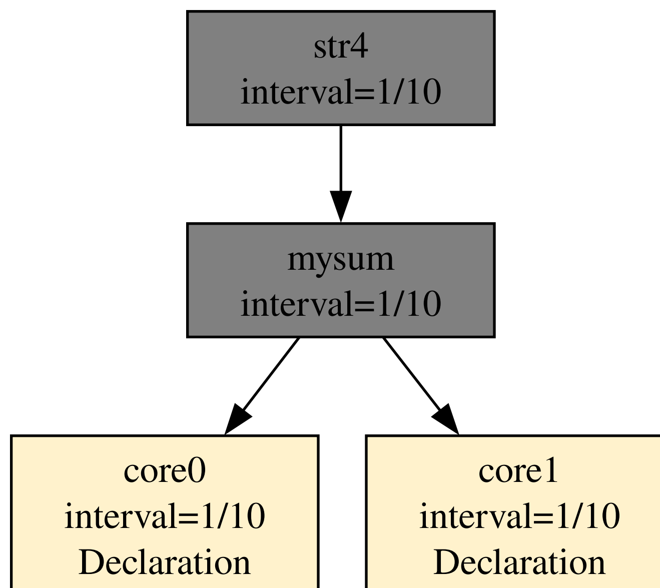
Gdy użytkownik doda jawną deklarację strumienia będącego dokładnie tą samą sumą:

```
SELECT * STREAM mysum FROM core0+core1
```

substrat `STREAM_ADD_core0_core1` spełnia wszystkie warunki równoważności względem `mysum` — identyczny interwał, identyczny program tokenów, identyczny schemat pól. Faza `deduplicateSubstrats()` usuwa substrat i przepina wszystkie odwołania `PUSH_STREAM` na `mysum`. Substrat znika z grafu w zupełności (Rys. 28):



Rys. 27. Graf z automatycznym substratem STREAM_ADD_core0_core1



Rys. 28. Graf po dodaniu SELECT * STREAM mysum FROM core0+core1 — substrat zastąpiony przez jawny strumień

Efekt uboczny: `mysum` staje się węzłem wspólnym — obsługuje zarówno własnych konsumentów, jak i tych, którzy wcześniej korzystali z automatycznego substratu. Użytkownik zyskuje przy tym jawną nazwę dla wyników pośrednich i może odpytywać je przez `xqry`.

39.4 Aktualizacja schematu po wchłonięciu

Samo przepięcie tokenów `PUSH_STREAM` to za mało. Każdy strumień przechowuje w `lSchema` sekwencję instrukcji opisujących, jak zbudować wartość wyjściową każdego pola — w tym tokeny `PUSH_ID(nazwa_strumienia, N)`, które mówią: „weź N-te pole z bufora wejściowego o nazwie `nazwa_strumienia`". Gdy substrat zostaje wchłonięty, te tokeny wciąż odnoszą się do starej, usuniętej nazwy substratu. Krok `localizeFieldOffsets()` buduje mapę offsetów na podstawie tokenów `PUSH_STREAM` w programie — jeśli klucz z `PUSH_ID` nie pasuje do żadnego wpisu w mapie, domyślnie przyjmuje offset 0.

Scenariusz błędu przy niezerowym offsecie

Rozważmy zapytanie:

```
DECLARE a INTEGER STREAM s1, 1 FILE 'data1.dat'
DECLARE b INTEGER STREAM s2, 1 FILE 'data2.dat'
DECLARE c INTEGER STREAM s3, 1 FILE 'data3.dat'

SELECT * STREAM mysum  FROM s1+s2
SELECT * STREAM merged FROM s3+(s1+s2)
```

Kompilator tworzy substrat `STREAM_ADD_s1_s2`. Strumień `merged` ma dwa źródła: `s3` (offset 0) i substrat `STREAM_ADD_s1_s2` (offset 1, bo s3 zajmuje pozycję 0). Funkcja `buildOutputSchema` zapisuje w `merged.lSchema` tokeny:

```
PUSH_ID(STREAM_ADD_s1_s2, 0)   ← pole a ze źródła na offsecie 1
PUSH_ID(STREAM_ADD_s1_s2, 1)   ← pole b ze źródła na offsecie 1
```

Po wchłonięciu `deduplicateSubstrats()` przepina `PUSH_STREAM` z `STREAM_ADD_s1_s2` na `mysum`. Jednak bez aktualizacji `lSchema` tokeny `PUSH_ID` wciąż noszą starą nazwę. Gdy `localizeFieldOffsets()` nie znajdzie `STREAM_ADD_s1_s2` w mapie offsetów, przyjmuje offset 0 — kolizję z polami `s3`. Efekt: pola `a` i `b` z `mysum` były odczytywane z offsetu 0 (pozycja `s3`) zamiast z offsetu 1 (pozycja `mysum`).

Poprawka: aktualizacja lSchema w deduplicateSubstrats

Aby uniknąć tej rozbieżności, `deduplicateSubstrats()` po zaktualizowaniu tokenów `PUSH_STREAM` wykonuje dodatkowy przebieg przez `lSchema` wszystkich zapytań i przepisuje:

- tokeny `PUSH_ID(stara_nazwa, N)` na `PUSH_ID(nowa_nazwa, N)` — to przypadek pól z `buildOutputSchema` dla `STREAM_ADD`,

- tokeny `PUSH_ID2("stara_nazwa[N]")` na `PUSH_ID2("nowa_nazwa[N]")` — to przypadek symbolicznych nazw tworzonych przez `buildOutputSchema` dla `STREAM_TIMEMOVE`, `STREAM_HASH`, `STREAM_SUBTRACT`.

Po poprawce wyjście kompilatora dla powyższego przykładu wygląda poprawnie:

```
merged(1/1)
        :- PUSH_STREAM(mysum)
        :- PUSH_STREAM(s3)
        :- STREAM_ADD
        a: INTEGER
             PUSH_ID(merged[1])
        b: INTEGER
             PUSH_ID(merged[2])
```

Pola `a` i `b` z `mysum` mają offset 1 (`merged[1]`, `merged[2]`), co odpowiada faktycznej pozycji `mysum` w buforze `merged` — po polu `c` ze strumienia `s3`.

Kaskadowe wchłonięcie

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `issue167_dedup_cascaded`, `issue167_dedup_field_names`, `issue167_dedup_nonzero_offset`, `issue167_dedup_positive`, `issue167_triarg` opisanych w załączniku pt. Testy Integracyjne.

`deduplicateSubstrats()` działa iteracyjnie (`while(changed)`), co pozwala na wielokrokowe wchłonięcia. W przykładzie:

```
SELECT * STREAM mysum   FROM s1+s2
SELECT * STREAM shifted FROM (s1+s2)>1
SELECT * STREAM merged  FROM s3+((s1+s2)>1)
```

w pierwszej rundzie `mysum` wchłania `STREAM_ADD_s1_s2` i przepisuje jego nazwy — również w schemacie pośredniego substratu `STREAM_TIMEMOVE_STREAM_ADD_s1_s2`. Dzięki temu w drugiej rundzie `shifted` może wchłonąć ten substrat (warunek programowy jest teraz spełniony, bo oba wskazują na `mysum`). Po dwóch rundach w planie nie pozostaje żaden substrat automatyczny, a `merged` korzysta bezpośrednio z `s3` i `shifted`.

Rozdział 40

Rozwijanie symbolu *

Każdy, który pisał w języku SQL poznał magiczny znak * w tym języku. Wywołanie polecenia SELECT z tym argumentem rozwinie listę argumentów w oparciu o schematy tabel powstałych w wyniku złączeń relacyjnych. Coś podobnego chciałem osiągnąć w języku RQL.

Przykład używa kanonicznych deklaracji z całego rozdziału:

```
DECLARE a BYTE, b INTEGER
STREAM core0, 0.1
FILE 'sensor_a.txt'

DECLARE c INTEGER, d FLOAT
STREAM core1, 0.2
FILE 'sensor_b.txt'

SELECT *
STREAM merged
FROM core0 + core1

SELECT merged[2]
STREAM result
FROM merged
```

Skompilujmy i zobaczmy efekt:

```
$ xretractor -c query.rql
merged(1/10)
        :- PUSH_STREAM(core0)
        :- PUSH_STREAM(core1)
        :- STREAM_ADD
        core0_0: BYTE
                PUSH_ID(merged[0])
        core0_1: INTEGER
                PUSH_ID(merged[1])
        core1_2: INTEGER
```

```
                PUSH_ID(merged[2])
        core1_3: FLOAT
                PUSH_ID(merged[3])
result(1/10)
        :- PUSH_STREAM(merged)
        result_0: INTEGER
                PUSH_ID(result[2])
core0(1/10)     sensor_a.txt
        a: BYTE
        b: INTEGER
core1(1/5)      sensor_b.txt
        c: INTEGER
        d: FLOAT
```

Symbol $*$ zamienił się w cztery pola: `core0_0`, `core0_1`, `core1_2`, `core1_3`. Konwencja nazewnictwa: nazwa strumienia źródłowego + absolutna pozycja w schemacie wynikowym. Typy pól decydują o kolejności — `core0` wnosi BYTE i INTEGER na pozycje 0 i 1, `core1` wnosi INTEGER i FLOAT na pozycje 2 i 3. Odwołując się przez `merged[2]` w zapytaniu `result` dostajemy pole typu INTEGER — trzecie w kolejności, pierwsze z `core1`.

Uwaga

Opisana funkcjonalność ma pokrycie w teście: `Pattern3` opisanym w załączniku pt. Testy Integracyjne.

Rozdział 41

Rozwiązywanie interwałów

Każdy strumień w RetractorDB ma przypisany interwał czasowy — delta (Δ). Interwał określa, jak często produkowane są nowe wartości. Dla strumieni deklarowanych (`DECLARE`) interwał podaje użytkownik. Dla strumieni wynikowych (`SELECT`) interwał wyznacza kompilator z równań algebry strumieni.

Przykłady w tym rozdziale używają kanonicznych deklaracji z całego rozdziału: `core0` (Δ=1/10), `core1` (Δ=1/5), `core2` (Δ=3/10).

41.1 Algorytm

Etap `resolveStreamIntervals` działa iteracyjnie:

```
prevUnresolved = ∞
pętla:
    unresolvedCount = 0
    posortuj qTree topologicznie
    dla każdego zapytania:
        jeśli delta strumieni źródłowych znana:
            wyznacz deltę wynikową z równania operatora
        w przeciwnym razie:
            unresolvedCount++
    jeśli unresolvedCount == 0: koniec (sukces)
    jeśli unresolvedCount >= prevUnresolved: błąd (pętla w grafie)
    prevUnresolved = unresolvedCount
```

Każda runda rozwiązuje co najmniej jeden strumień — bo graf jest acykliczny i sortowanie topologiczne gwarantuje, że źródła są przetwarzane przed wynikami. Jeśli liczba nierozwiązanych strumieni nie maleje, oznacza to cykl — patrz Wykrywanie pętli.

41.2 Równania operatorów

Suma strumieni (+, STREAM_ADD)

```
SELECT ... STREAM c FROM a + b
```

$$\Delta_c = \min(\Delta_a, \Delta_b)$$

Strumień wynikowy produkuje wartości tak często, jak szybszy ze strumieni wejściowych.

Przykład: core0(Δ=1/10) + core1(Δ=1/5) → str1(Δ=1/10)

Synchronizacja strumieni (#, STREAM_HASH)

```
SELECT ... STREAM c FROM a # b
```

$$\Delta_c = \frac{\Delta_a \cdot \Delta_b}{\Delta_a + \Delta_b}$$

Wynik odpowiada średniej harmonicznej interwałów — strumień produkuje wartości tylko wtedy, gdy oba wejścia są dostępne jednocześnie.

Przykład: core0(Δ=1/10) # core1(Δ=1/5) → str1(Δ=1/15)

Przesunięcie w czasie (`>n`, STREAM_TIMEMOVE)

```
SELECT ... STREAM c FROM a > n
```

$$\Delta_c = \Delta_a$$

Przesunięcie nie zmienia częstotliwości strumienia — tylko przesuwa okno odczytu o n próbek.

Agregaty okienkowe (`.max`, `.min`, `.avg`, `.sum`)

$$\Delta_c = \Delta_a$$

Agregaty redukują wartości w oknie, ale interwał strumienia wyjściowego pozostaje taki sam jak źródłowego.

Algorytm AGSE (`@(step, window)`, STREAM_AGSE)

```
SELECT ... STREAM c FROM a @ (step, window)
```

$$\Delta_c = \frac{\Delta_a \cdot \text{step}}{\text{windowSize}}$$

AGSE (Algorytm Generowania Serii Epizodów) generuje okna przesuwne. Interwał wynikowy zależy od kroku i rozmiaru okna względem źródła.

Operatory de-hash (STREAM_DEHASH_DIV, STREAM_DEHASH_MOD)

Operacje odwrotne do `#` — wyznaczają, jaki interwał miał jeden ze strumieni wejściowych, znając interwał wyniku i drugiego argumentu:

$$\Delta_a = \frac{\Delta_c \cdot \Delta_b}{|\Delta_c - \Delta_b|}$$

41.3 Dlaczego iteracja?

W zapytaniu z wieloma strumieniami wynikowymi jeden strumień może zależeć od drugiego:

```
DECLARE a INTEGER STREAM core0, 0.1 FILE 'data.dat'
SELECT str1[0] STREAM str1 FROM core0
SELECT str2[0] STREAM str2 FROM str1
```

W pierwszej rundzie iteracji kompilator wyznacza Δ_str1 = 1/10 (bo Δ_core0 jest znana). W drugiej rundzie — Δ_str2 = 1/10 (bo Δ_str1 jest już znana). Gdyby nie iteracja, str2 musiałoby być zadeklarowane przed str1, co ograniczałoby ekspresywność języka.

Rozdział 42

Wykrywanie pętli w kompilacji

Graf zależności zapytań musi być acyklicznym grafem skierowanym (DAG). Jeśli zapytanie odwołuje się — bezpośrednio lub pośrednio — do własnych wyników, powstaje cykl. Kompilator wykrywa taką sytuację i kończy kompilację z błędem.

Uwaga

Opisana funkcjonalność ma pokrycie w teście: `issue95_loopInCompile` opisanym w załączniku pt. Testy Integracyjne.

42.1 Przykład pętli

```
DECLARE a BYTE, b INTEGER
STREAM core0, 0.1
FILE 'sensor_a.txt'

DECLARE c INTEGER, d FLOAT
STREAM core1, 0.2
FILE 'sensor_b.txt'

SELECT merged[0]*10, merged[2]+10 STREAM merged FROM core0 + core1
SELECT agg[0] STREAM agg FROM merged.max
SELECT * STREAM broken FROM merged + broken
```

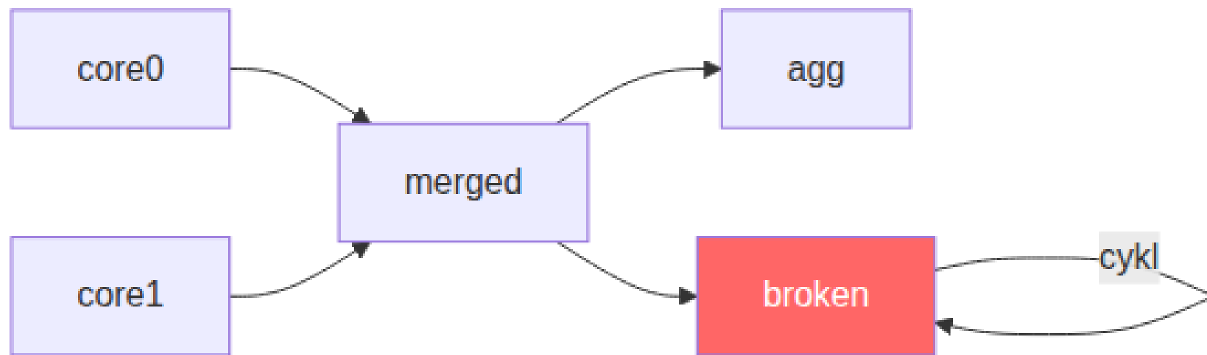
Ostatnie zapytanie definiuje `broken` jako wynik operacji `merged + broken` — strumień zależy od samego siebie. Graf zależności zawiera cykl (Rys. 29):

Rys. 29. Cykl w grafie zależności zapytań

42.2 Efekt kompilacji

Próba kompilacji takiego pliku kończy się błędem:

```
$ xretractor brokenQuery.rql -c 2>out.txt
```



Diagram

```
$ echo $?
1
$ cat out.txt
[error] Circular dependency: stream interval resolution stalled with 1 unresolved streams
```

Komunikat `"Circular dependency in stream definitions"` pojawia się, gdy etap `resolveStreamIntervals` wykryje, że liczba nierozwiązanych strumieni przestała maleć. Jak uruchomić kompilację i czytać komunikaty błędów — patrz Debugowanie kompilacji.

42.3 Mechanizm wykrywania

Etap `resolveStreamIntervals` w każdej rundzie iteracji liczy strumienie, dla których nie udało się jeszcze wyznaczyć interwału (`unresolvedCount`). W poprawnym grafie acyklicznym liczba ta maleje co rundę — zawsze co najmniej jeden strumień uzyskuje wyznaczoną deltę. W grafie z cyklem strumienie wzajemnie od siebie zależą i żaden nie może uzyskać wartości — `unresolvedCount` zatrzymuje się.

```
if (unresolvedCount >= prevUnresolved) {
    SPDLOG_ERROR("Circular dependency: stream interval resolution stalled with {} unresolved st
                 unresolvedCount);
    return std::string("Circular dependency in stream definitions");
}
prevUnresolved = unresolvedCount;
```

Warunek >= (a nie >) chroni przed fałszywymi pozytywami: jeśli liczba nie maleje nawet o jeden, postęp jest niemożliwy.

42.4 Jak naprawić

Usunąć odwołanie strumienia do samego siebie lub do strumienia, który od niego zależy. W powyższym przykładzie zapytanie:

```
SELECT * STREAM broken FROM merged + broken
```

należy zastąpić odwołaniem do strumienia, który istnieje niezależnie od `broken`:

```
SELECT * STREAM broken FROM merged + core0
```

Rozdział 43

Aliasowanie

W przypadku, w którym złączymy dwa strumienie danych operatorem sumy. Pojawi się nowy schemat danych. Do kolejnych wartości tego schematu możemy odwoływać się poprzez nazwę strumienia danych indeksowanych kolejno względem początku schematu.

Możemy jednak użyć też nazw z jakich strumień powstał. Na wartość wskazywać będzie nazwa strumienia wynikowego indeksowana względem początku schematu, jak również nazwa strumienia źródłowego przesunięta względem pozycji złączenia.

Przykład używa kanonicznych deklaracji z całego rozdziału:

```
DECLARE a BYTE, b INTEGER
STREAM core0, 0.1
FILE 'sensor_a.txt'

DECLARE c INTEGER, d FLOAT
STREAM core1, 0.2
FILE 'sensor_b.txt'

SELECT merged[0], merged[2], core0[0], core1[0]
STREAM merged
FROM core0 + core1
```

Po kompilacji otrzymamy:

```
$ xretractor -c query.rql
merged(1/10)
        :- PUSH_STREAM(core0)
        :- PUSH_STREAM(core1)
        :- STREAM_ADD
        merged_0: BYTE
                PUSH_ID(merged[0])
        merged_1: INTEGER
                PUSH_ID(merged[2])
        merged_2: BYTE
                PUSH_ID(merged[0])
```

```
        merged_3: INTEGER
                PUSH_ID(merged[2])
core0(1/10)     sensor_a.txt
        a: BYTE
        b: INTEGER
core1(1/5)      sensor_b.txt
        c: INTEGER
        d: FLOAT
```

`merged[0]` i `core0[0]` oba trafiają na `PUSH_ID(merged[0])` — to to samo pole. Natomiast `core1[0]` — pierwsze pole schematu `core1` — trafia na `PUSH_ID(merged[2])`, nie `merged[0]`. Kompilator przetłumaczył lokalny indeks `core1[0]` na absolutną pozycję w schemacie złączonym: `core0` zajmuje pozycje 0 i 1, więc `core1` zaczyna się na pozycji 2. A co, jeśli operację `+` zastąpimy `#`? Zachęcam do eksperymentów.

Uwaga

Opisana funkcjonalność ma pokrycie w teście: `Pattern7` opisanym w załączniku pt. Testy Integracyjne.

Rozdział 44

Przetwarzanie symbolu _

W niektórych zapytaniach można użyć symbolu podkreślenia. Ta technika to cukier syntaktyczny. Podobnie jak rozwijanie symbolu * w wyniku pojawienia się jednego odwołania w wyniku kompilacji pojawi się wiele pól. Ile tych pól powstanie ma wpływ co z czym i w jakiej kolejności zostało złączone w klauzuli FROM.

Przykład używa kanonicznych deklaracji z całego rozdziału — `core0` ma dwa pola (BYTE, INTEGER), `core1` ma dwa pola (INTEGER, FLOAT), schematy są równoliczne:

```
DECLARE a BYTE, b INTEGER   STREAM core0, 0.1 FILE 'sensor_a.txt'
DECLARE c INTEGER, d FLOAT  STREAM core1, 0.2 FILE 'sensor_b.txt'

SELECT core0[_] * core1[_]
STREAM scaled
FROM core0 + core1
```

Po przeprowadzeniu kompilacji:

```
$ xretractor -c query.rql
scaled(1/10)
        :- PUSH_STREAM(core0)
        :- PUSH_STREAM(core1)
        :- STREAM_ADD
        scaled_0: INTEGER
                PUSH_ID(scaled[0])
                PUSH_ID(scaled[2])
                MULTIPLY
        scaled_1: FLOAT
                PUSH_ID(scaled[1])
                PUSH_ID(scaled[3])
                MULTIPLY
core0(1/10)     sensor_a.txt
        a: BYTE
        b: INTEGER
core1(1/5)      sensor_b.txt
        c: INTEGER
```

```
d: FLOAT
```

Symbol _ rozwinął się w dwa pola: `scaled[0] * scaled[2]` (czyli `a * c`) i `scaled[1] * scaled[3]` (czyli `b * d`). Odwołania do `core0` i `core1` zostały przetłumaczone przez aliasowanie na absolutne pozycje w schemacie złączonym. Typy wynikowe to INTEGER (`BYTE * INTEGER`) i FLOAT (`INTEGER * FLOAT`) — wynik równania typów w górę, opisanego w osobnym podrozdziale.

Po pojawieniu się w formule operatora _ w indeksie tablicy, kompilator powieli formułę dla wszystkich pól argumentów. Schematy obu argumentów muszą być równoliczne. Czyli core0 i core1 muszą mieć schematy tej samej liczności – typy zostaną wyrównane do najwyższego. O równaniu typów wspomnę za chwilę.

Ta funkcjonalność ma główne zastosowanie w przypadku budowy zapytań w których budujemy algorytmy filtrów sygnałowych. Tam dochodzi do szeregu operacji matematycznych. Funkcjonalność związana z przetwarzaniem symbolu _ nie jest wymagana w celu osiągnięcia pełnej funkcjonalności systemu RetractorDB. Jednak znacząco upraszcza budowę specyficznych zapytań w których należy połączyć operacje na dwóch schematach. Przykład zastosowania zostanie przedstawiony w trakcie prezentacji algorytmów przetwarzania sygnałów.

Rozdział 45

Równanie typów w górę

Co się dzieje w przypadku, kiedy mnożymy dane typu BYTE z danymi typu INTEGER ? W systemie RetractorDB obowiązują ścisłe zasady równania typów w górę. Pomnożenie pola typu BYTE z wartością pola, które jest typu INTEGER spowoduje powstanie w schemacie typu pola INTEGER. To dzieje się na etapie kompilacji.

Na chwilę obecną system RetractorDB wspiera następujące typy danych:

Typ	Opis
BYTE	wartości 0–255
INTEGER	4 bajtowe wartości dla liczb ze znakiem
UINT	podobnie jak INTEGER dla liczb bez znaku
RATIONAL	liczby wymierne
FLOAT	liczby zmiennoprzecinkowe
DOUBLE	liczby zmiennoprzecinkowe podwójnej precyzji
STRING	ciągi znaków

Typy STRING i RATIONAL wymagają jeszcze przeglądu, poprawek i pokrycia testami. W trakcie rozwoju oprogramowania skupiłem wysiłek na przetwarzaniu liczb. Chcę w przyszłości jeszcze dołączyć do tego zbioru typy liczb zespolonych i wymiernych liczb zespolonych Eisensteina.

Przykład równania typów w praktyce — zapytanie `scaled` z rozdziału Przetwarzanie symbolu _:

```
SELECT core0[_] * core1[_]
STREAM scaled
FROM core0 + core1
```

`core0` ma pola BYTE i INTEGER, `core1` ma pola INTEGER i FLOAT. Po rozwinięciu _ kompilator wyznacza typy pól wynikowych:

Wyrażenie	Lewy typ	Prawy typ	Typ wynikowy
`scaled[0] * scaled[2]`	BYTE	INTEGER	INTEGER

Wyrażenie	Lewy typ	Prawy typ	Typ wynikowy
`scaled[1] * scaled[3]`	INTEGER	FLOAT	FLOAT

Rozdział 46

Debugowanie kompilacji

Kompilator transformuje plik `.rql` w plan wykonania przez kilka etapów. Efekt każdego etapu jest widoczny przez flagi diagnostyczne `xretractor`. Opisane tutaj narzędzia pozwalają odpowiedzieć na pytania: dlaczego schemat wygląda inaczej niż napisałem? skąd ta delta? dlaczego pojawił się substrat?

46.1 Podstawowe narzędzie: flaga `-c`

Flaga `-c` (`--onlycompile`) zatrzymuje `xretractor` po kompilacji i drukuje skompilowany plan na standardowe wyjście — bez uruchamiania przetwarzania:

```
xretractor -c query.rql
```

Kod wyjścia `0` oznacza sukces. Kod `1` — błąd kompilacji. Komunikaty błędów trafiają na stderr:

```
xretractor -c query.rql 2>errors.txt
echo $?
```

Kompilację można wywołać nawet gdy inny proces `xretractor` już działa — flaga `-c` nie próbuje przejąć blokady wykonania.

46.2 Jak czytać plan kompilacji

Dla kanonicznego `query.rql` z tego rozdziału plan wygląda następująco:

```
merged(1/10)
        :- PUSH_STREAM(core0)
        :- PUSH_STREAM(core1)
        :- STREAM_ADD
        core0_0: BYTE
                PUSH_ID(merged[0])
        core0_1: INTEGER
                PUSH_ID(merged[1])
        core1_2: INTEGER
```

```
                PUSH_ID(merged[2])
        core1_3: FLOAT
                PUSH_ID(merged[3])
result(1/10)
        :- PUSH_STREAM(merged)
        result_0: BYTE
                PUSH_ID(merged[0])
        result_1: INTEGER
                PUSH_ID(merged[2])
        result_2: BYTE
                PUSH_ID(merged[0])
        result_3: INTEGER
                PUSH_ID(merged[2])
core0(1/10)     sensor_a.txt
        a: BYTE
        b: INTEGER
core1(1/5)      sensor_b.txt
        c: INTEGER
        d: FLOAT
core2(3/10)     sensor_c.txt
        e: INTEGER
```

Każdy blok ma ustalony format:

```
nazwaStrumienia(delta)
        :- operacjaStrumieniowa(arg)
        nazwaPolaWyjściowego: TYP
                instrukcja
                ...
```

Element	Znaczenie
`nazwaStrumienia(delta)`	Nazwa strumienia i jego interwał jako ułamek: `1/10` = 0.1 s = 10 Hz
`:- PUSH_STREAM(x)`	Pcha strumień `x` na stos strumieniowy; pojawia się raz na każdy argument FROM
`:- STREAM_ADD`	Operator sumy strumieni (+ w FROM)
`:- STREAM_HASH`	Operator synchronizacji strumieni (`#` w FROM)
`:- STREAM_TIMEMOVE(n)`	Przesunięcie w czasie (`>n` w FROM)
`pole: TYP`	Pole schematu wynikowego po równaniu typów w górę
`PUSH_ID(s[n])`	Odkłada na stos wartość pola `n` ze strumienia `s` — tu widoczny efekt aliasowania
`PUSH_VAL(x)`	Odkłada stałą `x` na stos
`ADD`, `MULTIPLY`, ...	Operacja arytmetyczna: zdejmuje dwa argumenty ze stosu, odkłada wynik

Bloki efemerydów (`DECLARE`) pojawiają się na końcu planu — zawierają listę pól i ścieżkę

do pliku danych.

Aliasowanie w planie: jeśli dwa pola wyjściowe wskazują na ten sam `PUSH_ID`, są aliasami. W przykładzie `result_0` i `result_2` oba to `PUSH_ID(merged[0])` — potwierdzenie, że `merged[0]` i `core0[0]` to ta sama pozycja. Patrz Aliasowanie.

Substraty w planie: automatycznie wygenerowany substrat pojawia się jako blok z nazwą w stylu `STREAM_HASH_core0_core1` — bez odpowiadającego `SELECT` w pliku źródłowym. Patrz Substraty.

46.3 Wizualizacja grafu zależności

Zamiast tekstu można wygenerować graf w formacie DOT i przetworzyć przez `graphviz`:

```
xretractor -c -d -f -s query.rql > out.dot && dot -Tsvg out.dot -o out.svg
```

Dostępne flagi modyfikujące wyjście DOT:

Flaga	Pełna nazwa	Znaczenie
`-d`	`--dot`	generuj wyjście DOT zamiast tekstowego planu
`-f`	`--fields`	pokaż pola strumieni w węzłach grafu
`-s`	`--streamprogs`	pokaż sekwencje instrukcji stosu w węzłach
`-u`	`--rules`	pokaż reguły RULE
`-p`	`--transparent`	przezroczyste tło — do osadzania w dokumentach

Graf pokazuje zależności między strumieniami jako krawędzie skierowane od źródeł do wyników. Substraty mają inny kolor niż strumienie jawnie zdefiniowane przez użytkownika. Patrz Budowa drzewa zależności.

Uwaga

Opisana funkcjonalność ma pokrycie w teście: `issue31_doc` opisanym w załączniku pt. Testy Integracyjne.

46.4 Weryfikacja interwałów

Jeśli delta strumienia wynikowego jest niespodziewana:

1. Sprawdź delty strumieni źródłowych — widoczne w blokach DECLARE na końcu planu.

2. Sprawdź operator w klauzuli FROM — każdy operator ma inne równanie na deltę.

Przykład: `core0(1/10) # core1(1/5)` daje deltę `1/15` (średnia harmoniczna), nie `1/10`. Jeśli spodziewałeś się `1/10`, użyj `+` zamiast `#`. Pełne równania — patrz Rozwiązywanie interwałów.

46.5 Typowe błędy kompilacji

Cykl w grafie zależności

```
[error] Circular dependency: stream interval resolution stalled with N unresolved streams
```

Strumień odwołuje się pośrednio lub bezpośrednio do samego siebie. Wygeneruj graf przez `-d` — cykl będzie widoczny jako pętla. Patrz Wykrywanie pętli.

Nieznany strumień

Odwołanie do strumienia, który nie został jeszcze zadeklarowany. Pliki `.rql` przetwarzane są sekwencyjnie — `SELECT` nie może odwoływać się do strumienia zdefiniowanego niżej w pliku. Przesuń `DECLARE` lub `SELECT` wyżej.

Niezgodność krotności schematów przy _

Oba strumienie w wyrażeniu `core0[_] * core1[_]` muszą mieć schematy tej samej liczności. Sprawdź ile pól ma każdy z argumentów w blokach DECLARE planu. Patrz Przetwarzanie symbolu _.

Plik danych niedostępny

Błąd ten **nie pojawia się przy** `-c` — flaga weryfikuje poprawność zapytania, nie sprawdza czy pliki danych istnieją. Błąd dostępu do pliku pojawi się dopiero przy uruchamianiu przetwarzania bez `-c`.

Rozdział 47

Realizacja zapytań

Proces realizacji opiera się na ciągłym przeglądzie drzewa zapytań i sekwencyjnym i hierarchicznym wywoływaniu procedur budujących kolejne krotki strumieni i kolejne schematy danych.

Opis algorytmu należy zacząć od przedstawienia procedury sekwencjonowania. W systemie, w którym realizowane są zapytania z różnorodnymi wartościami definiującymi czasokres pomiędzy tworzonymi i napływającymi kolejnymi danymi potrzebny jest sposób wyznaczenia kolejnych interwałów.

Przeanalizujmy na początek następujący przykład. Załóżmy że w systemie występują dwa strumienie danych. Jeden napływa co sekundę drugi co dwie sekundy. Algorytm sekwencjonowania powinien zaproponować sekundowy interwał czasu pomiędzy wywoływaniem procedury przeznaczonej dla strumienia pierwszego i dwusekundowy dla strumienia drugiego. W praktyce przedstawiona zostanie sekundowa siatka czasowa – w której wszystkie sekundowe węzły zostaną wypełnione procedurą budowy krotek ze strumienia pierwszego i w tej samej siatce czasu – drugi strumień dołączy swoje procedury co drugą sekundę.

Wyznaczona w trakcie kompilacji siatka czasu jest bardzo istotna – to ona definiuje jak często i w jakich odstępach będą przetwarzane strumienie danych, które węzły zostaną pokryte przez wygenerowane procedury przetwarzania strumieni danych.

Zastanówmy się nad bardziej skomplikowanym przykładem. Załóżmy istnienie trzech strumieni danych. Pierwszy z częstotliwością napływu ⅓ drugi z szybkością ½ oraz trzeci napływający z szybkością ⅔. Wyznaczenie siatki i umieszczenie kolejnych procedur przetwarzania wymaga bardziej skomplikowanego rozwiązania. Wartość ⅔ może zostać uproszczona do ⅓. Bowiem istnieje naturalny podzielnik tych wartości. Nie istnieje natomiast naturalny podzielnik wartości ½ oraz ⅓. Wartość siatki jaka zostanie wyznaczona to ⅕. Jest to największa możliwa liczba wymierna, która jeśli zbuduje się na osi liczb wymiernych siatkę pomieści regularne serie czasowe o częstotliwości napływa ½ oraz ⅓.

Na wyznaczoną siatkę nakładane są wszystkie strumienie i wyznaczany jest zbiór minimalnych odstępów czasu dla zapytań w których istnieją mnożniki naturalne. W naszym przypadku minimalny zbiór odstępów czasu dla zapytań o mnożnikach (⅓, ½, ⅔) to

zbiór ($\frac{1}{3}$, $\frac{1}{2}$). Odstępy $\frac{1}{3}$ oraz $\frac{2}{3}$ będą współdzielić slot czasowy na siatce.

Analizując poniższy wywód może bardziej oczywiste staną się wspomniane w rozdziale komentarze generowane dla programu swirly przedstawiające generowane schematy kulkowe.

Rozdział 48

Algorytm przeglądu drzewa zapytań

48.1 Przegląd ogólny

Algorytm przeglądu drzewa zapytań realizowany jest przez dwa współpracujące komponenty: `dataModel` (logika przetwarzania) oraz `executorsm` (pętla czasowa i IPC). Przed wejściem w główną pętlę system wykonuje **krok zerowy**, po czym cyklicznie iteruje po minimalnym zbiorze interwałów czasowych (Rys. 30).

Rys. 30. Algorytm przeglądu drzewa zapytań – przegląd ogólny

48.2 Struktura danych: qTree

`qTree` (`src/retractor/lib/qTree.cpp`) rozszerza `std::vector<query>` i jest **wektorem topologicznie posortowanych zapytań**. Sortowanie odbywa się przez DFS po grafie zależności budowanym z `query.getDepStream()` (Rys. 31).

Rys. 31. Przykładowy graf zależności dla qTree

Po sortowaniu topologicznym kolejność w wektorze: `[A, B, C, D]`. Zapytanie C zależne od B zawsze trafi po B w iteracji — gwarantuje poprawność obliczeń.

Metoda `getAvailableTimeIntervals()` wyodrębnia ze wszystkich zapytań unikalne wartości `rInterval` (z pominięciem dyrektyw kompilatora i wartości zerowych) — wynik to wejście do konstruktora `TimeLine`.

48.3 Minimalna siatka czasowa: TimeLine / CRSMath

`TimeLine` (`src/retractor/lib/CRSMath.cpp`) zarządza racjonalnymi interwałami czasowymi. Konstruktor redukuje zbiór interwałów — usuwa wielokrotności, zachowując

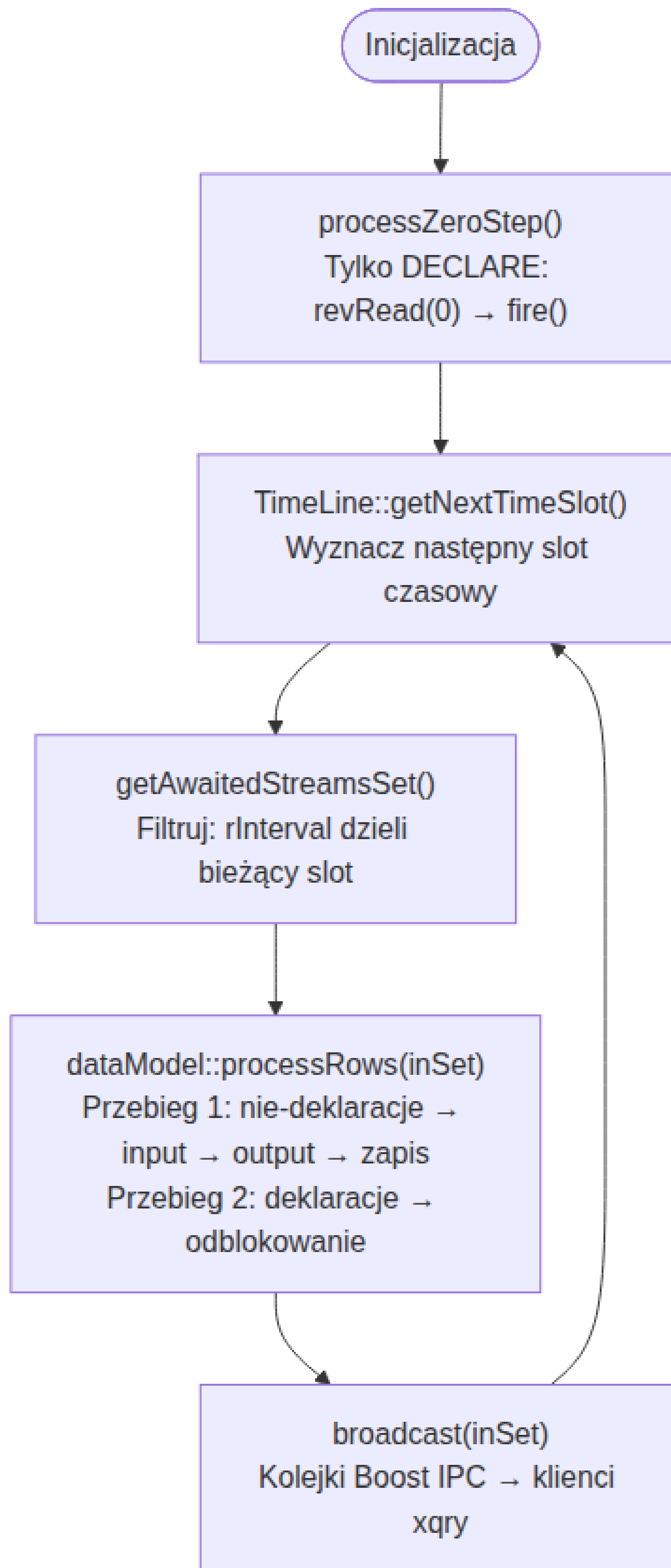
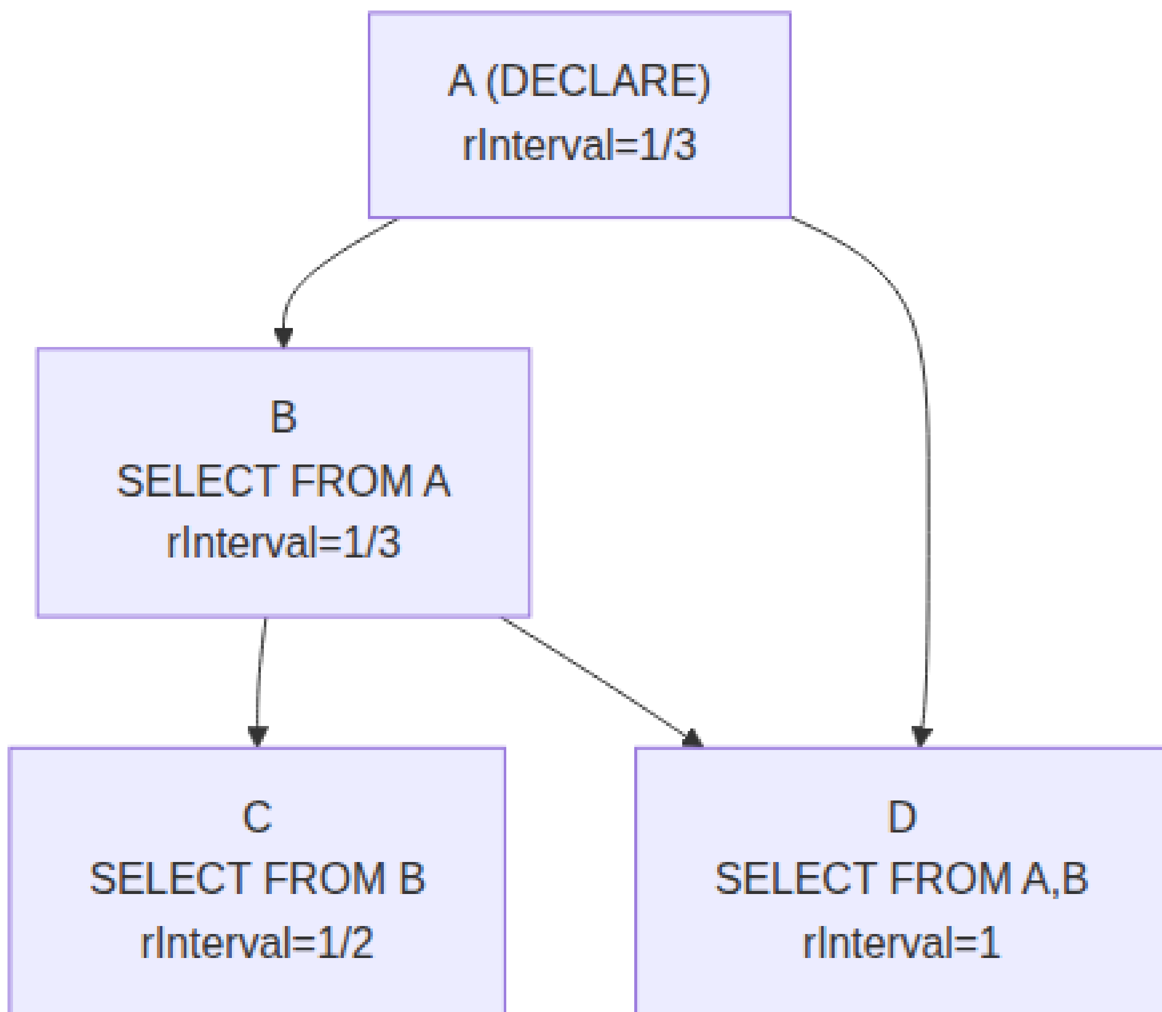


Diagram
174



Diagram

tylko koprimalne:

```
Wejście: {1/2, 1, 4}  →  Wyjście: {1/2}
(1 = 2 × 1/2, więc redundantne; 4 = 8 × 1/2, więc redundantne)

Wejście: {1/2, 1/3}  →  Wyjście: {1/2, 1/3}
(żadne nie jest wielokrotnością drugiego)
```

`getNextTimeSlot()` wyznacza kolejny slot jako `min(delta × counter[delta])` po wszystkich deltach. Poniższy diagram ilustruje sloty dla delt `{1/2, 1/3}` i aktywne zapytania w każdym z nich (Rys. 32):

Diagram

Rys. 32. Minimalna siatka czasowa dla delt {1/2, 1/3}

Sprawdzenie `isThisDeltaAwaitCurrentTimeSlot(inDelta)` zwraca `true`, gdy `ctSlot_ / inDelta` ma mianownik równy 1 (slot jest całkowitą wielokrotnością delty zapytania).

48.4 Krok zerowy: `processZeroStep()`

Przed wejściem w pętlę `executorsm::run()` wywołuje `processZeroStep()` (`dataModel.cpp`, linia ~85). Przetwarza **wyłącznie deklaracje** (strumienie wejściowe `DECLARE`):

```
for (auto &q : coreInstance_) {
    if (!q.isDeclaration()) continue;
    qSet[q.id]->bufferState = flux;   // odblokuj odczyt fizyczny
    qSet[q.id]->revRead(0);           // wczytaj z indeksu 0
    qSet[q.id]->fire();               // przepisz chamber_ → outputPayload
    assert(qSet[q.id]->bufferState == armed);
}
```

Po tym kroku każda deklaracja ma `bufferState = armed` — dane z fizycznego źródła są w `outputPayload`.

48.5 Główna pętla: filtrowanie i przetwarzanie

Filtrowanie zapytań: `getAwaitedStreamsSet()`

Dla bieżącego slotu `tl` (`executorsm.cpp`, linia ~88):

```
std::set<std::string> retVal;
for (auto &q : *coreInstancePtr)
    if (TimeLine::isThisDeltaAwaitCurrentTimeSlot(q.rInterval))
        retVal.insert(q.id);
return retVal;
```

Wynik `inSet` to identyfikatory zapytań aktywnych w tym slocie — podzbiór wszystkich zapytań.

Przetwarzanie: `processRows(inSet)`

Funkcja wykonuje **dwa przejścia** przez `inSet` (`dataModel.cpp`, linia ~98), co ilustruje Rys. 33:

Rys. 33. Algorytm processRows – dwa przejścia przetwarzania

Deklaracje są odblokowywane dopiero po tym, jak wszystkie zależne zapytania skonsumowały ich `outputPayload` w przejściu 1.

48.6 Rozgłaszanie wyników: `broadcast()`

Po każdym `processRows()` wywoływane jest `broadcast(inSet)` (`executorsm.cpp`, linia ~449) — algorytm przedstawia Rys. 34:

Rys. 34. Algorytm broadcast – rozsyłanie wyników przez Boost IPC

`printRowValue()` buduje strukturę z nazwą strumienia, liczbą pól, wartościami i bitmapą null, zapisuje jako Boost info format i wysyła przez `boost::interprocess::message_queue`.

48.7 Pełny przykład: zapytania A, B, C, D dla delt {1/2, 1/3}

Rys. 35 przedstawia kompletną sekwencję wywołań dla czterech zapytań A, B, C, D rozłożonych na siatce czasowej z deltami {1/2, 1/3}.

Rys. 35. Pełny przykład wykonania dla zapytań A, B, C, D przy deltach {1/2, 1/3}

Drzewo zależności determinuje kolejność przejścia 1. Interwały czasowe z algebry Beatty'ego wyznaczają, które węzły drzewa są aktywne w danym slocie.

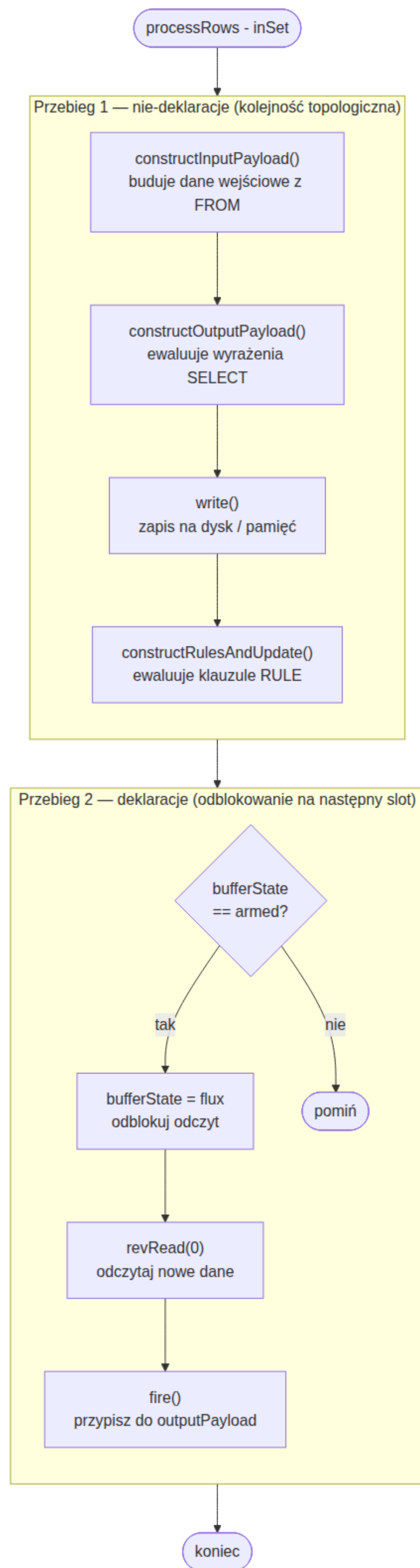


Diagram
178

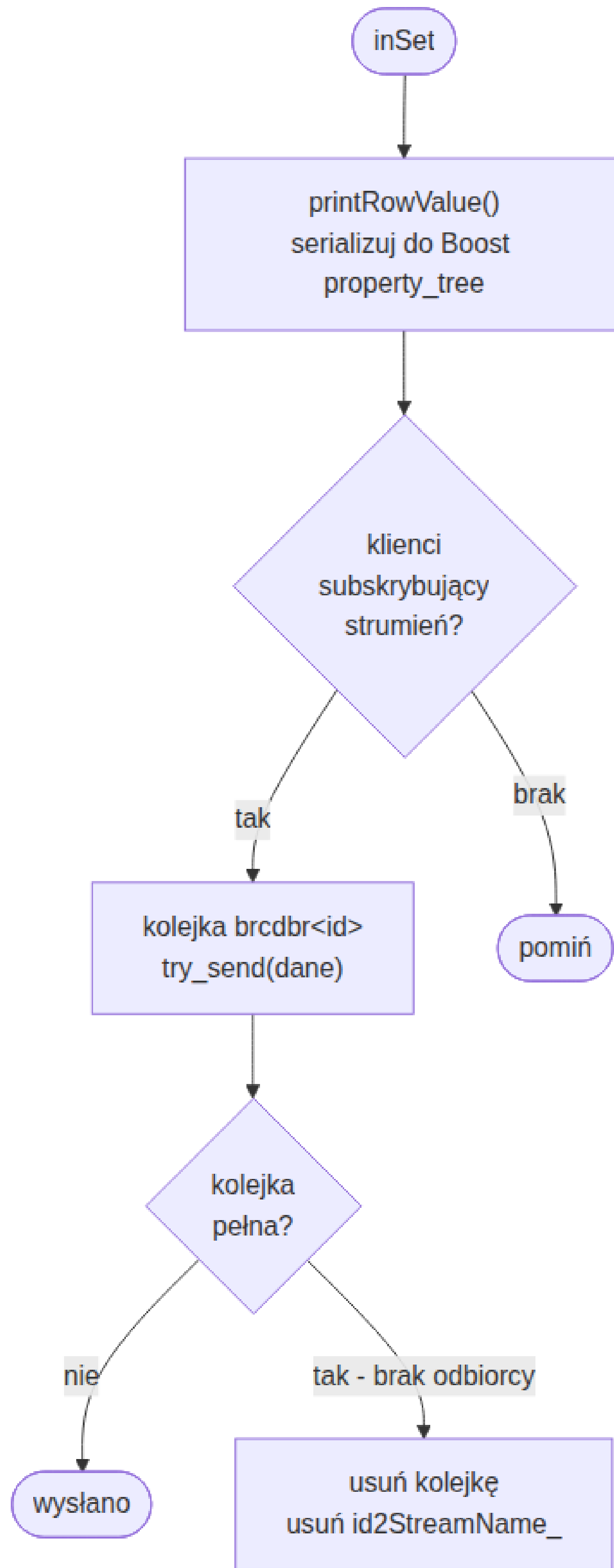


Diagram
179

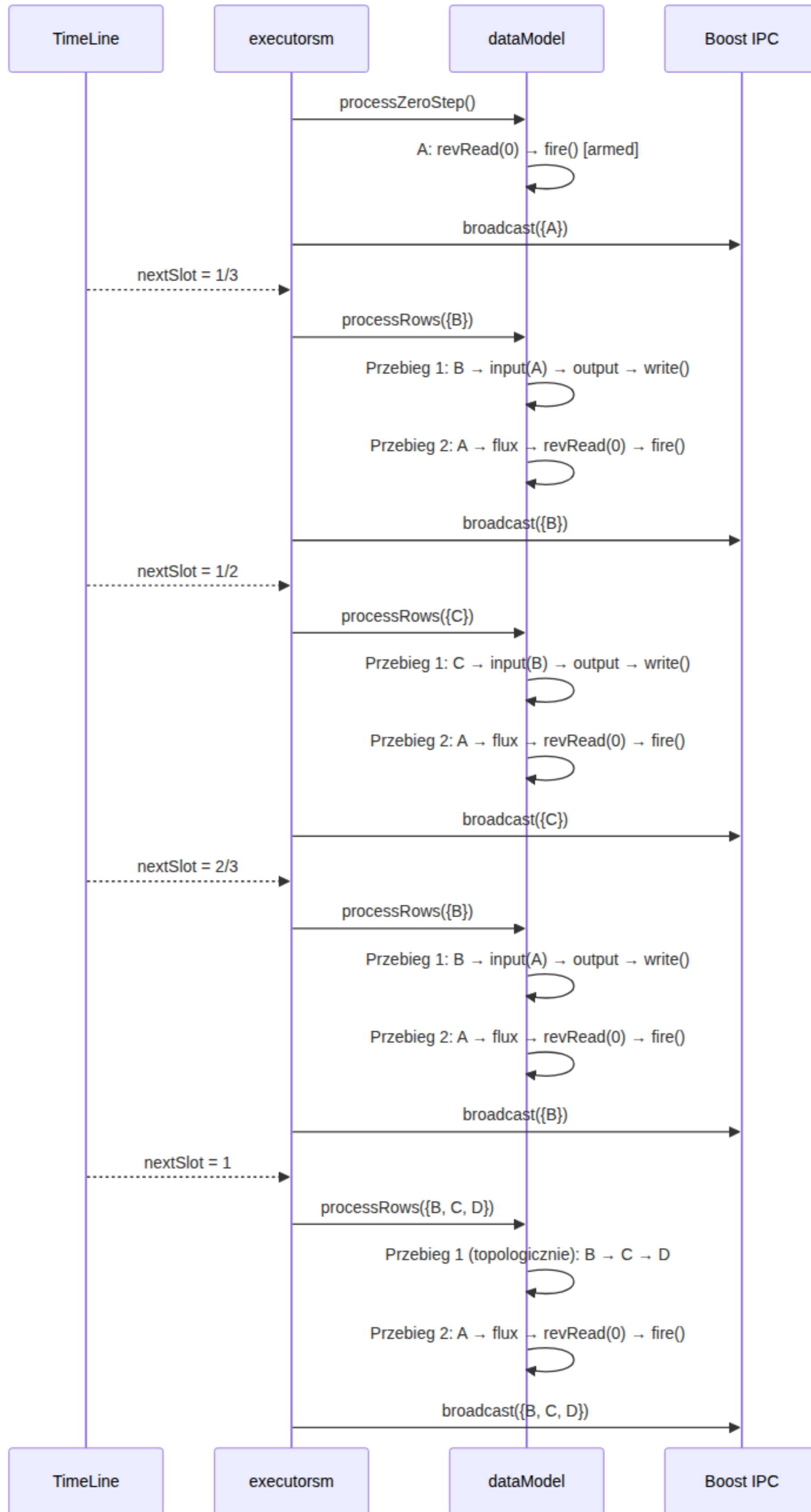


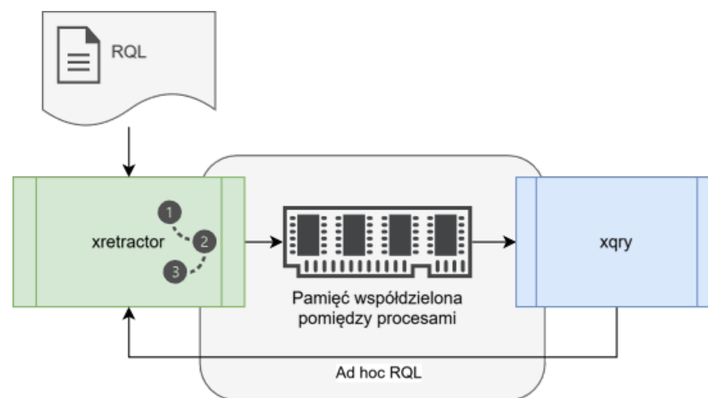
Diagram
180

Rozdział 49

Zapytania Ad hoc

Przez zapytania Ad hoc rozumiemy zapytania kierowane do działającego systemu. W typowym scenariuszu jaki zakładano w przypadku rozwoju systemu, założono we wstępnie rozpatrywanych scenariuszach, że użytkownik systemu będzie znał wszystkie zapytania i źródła danych wymagane do uzyskania przetworzonych serii czasowych.

W trakcie rozwoju systemu pojawiły się jednak dodatkowe scenariusze, zakładające, że praca systemu nie powinna być przerywana a dodatkowe zapytania powinny zostać dołączone do planu realizacji zapytań. Tego typu funkcjonalność będziemy nazywać zapytaniami Ad hoc, dołączanymi do systemu w trakcie jego działania bez przerywania jego pracy.



Rys. 36 Przepływ sterowania dla zapytań Ad Hoc

Na Rys. 36 przedstawiono opisany powyżej przepływ sterowania. Plik z zapytaniami i dyrektywami najpierw jest kierowany do procesu xretractor. Następnie poprzez pamięć współdzieloną proces xqry pobiera dane z xretractor. Tym samym procesem możemy wysłać do procesu xretractor polecenie. W tym poleceniu zawieramy tekst dodatkowego zapytania, które xretractor powinien dołączyć do przetwarzanego drzewa.

Przykład

Przykład rozpoczniemy od przygotowania prostego zapytania:

```
DECLARE a BYTE STREAM A, 1 FILE 'data1.txt'
DECLARE a BYTE STREAM B, 2 FILE 'data2.txt'
SELECT * STREAM str1 FROM A+B
```

Plik z zapytaniem zapiszemy pod nazwą qplan1.rql. Do poprawnej realizacji zapytania konieczne jest również przygotowanie plików data1.txt i data2.txt. Proponuję wypełnić data1.txt kolejnymi liczbami od 1 do 6 każda w nowej linii, a w pliku data2.txt liczby od 10 do 15. W tak przygotowanym katalogu uruchamiamy polecenie

```
$ xretractor qplan1.rql
```

Jeśli poprzednio w tym katalogu wykonywaliśmy jakieś operacje i stworzyliśmy strumień str1 o innym schemacie – otrzymamy błąd pt. „Error in data descriptor file". Pojawi się tam również informacja o różnicach pomiędzy dwoma deskryptorami. W takim przypadku plik str1 oraz str1.desc powinniśmy usunąć i ponownie wykonać polecenie.

Proces xretractor rozpocznie przetwarzanie danych. Należy w tym momencie uruchomić kolejny terminal i wydać w nim polecenie:

```
$ xqry -d
|str1|1|48|24|         |0|
|   A|1|-1| 3|data1.txt|1|
|   B|2|-1| 2|data2.txt|1|
ok.
```

W postaci tabelarycznej wyświetli się co w danym systemie się przetwarza. Ile bajtów już napłynęło, z jakich plików dane są czytane. Ile danych zostało już przetworzonych. Oczekując bardziej opisowej formy możemy wydać następujące polecenie:

```
$ xqry -y
---
apiVersion: xqry/v1
streams:
  - name: str1
    delta: 1
    size: 214
    count: 107
  - name: A
    delta: 1
    count: 86
    location: data1.txt
  - name: B
    delta: 2
    count: 43
    location: data2.txt
```

Udzielona odpowiedź jest w formie yaml.

Aby dołożyć do systemu kolejne zapytanie musimy wydać polecenie:

```
$ xqry -a "SELECT * STREAM str2 FROM A#B"
snd: adhoc SELECT * STREAM str2 FROM A#B
```

```
rcv: db OK
```

Polecenie w tej formie wysyła do procesu xretractor nowe zapytanie. System otrzymując je prowadzi kompilację i złączy drzewa planów zapytań.

Jeśli zajrzymy ponownie do stanu systemu, zobaczymy następujący obraz:

```
$ xqry -d
|str2|2/3| 10| 10|         |0|
|   A|  1| -1| 23|data1.txt|1|
|str1|  1|312|156|         |0|
|   B|  2| -1| 12|data2.txt|1|
ok.
```

Lub tak:

```
$ xqry -y
---
apiVersion: xqry/v1
streams:
  - name: str2
    delta: 2/3
    size: 7
    count: 7
  - name: A
    delta: 1
    count: 16
    location: data1.txt
  - name: str1
    delta: 1
    size: 298
    count: 149
  - name: B
    delta: 2
    count: 8
    location: data2.txt
```

Przyglądając się dokładniej zapytaniom za pomocą polecenia xqry zobaczymy następującą odpowiedź systemu dla zapytania str1:

```
$ xqry -t str1
---
apiVersion: xqry/v1
stream:
  name: str1
  delta: 1
query: SELECT * STREAM str1 FROM A+B
fields:
  str1.A_0:
    type: BYTE
  str1.B_1:
```

```
    type: BYTE
```

oraz dla zapytania str2:

```
$ xqry -t str2
---
apiVersion: xqry/v1
stream:
  name: str2
  delta: 2/3
query: SELECT * STREAM str2 FROM A#B
fields:
  str2.a:
    type: BYTE
```

Jak widać dodatkowe zapytanie str2 zostało poprawnie złączone z istniejącym planem realizacji zapytania. Widać też że zebranych danych jest o wiele mniej w porównaniu z str1.

Uwaga

Opisana funkcjonalność ma pokrycie w teście: `issue6_adhoc` opisanym w załączniku pt. Testy Integracyjne.

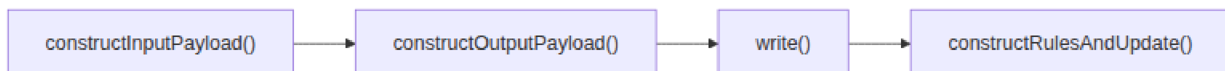
Rozdział 50

Realizacja alarmowania

Mechanizm alarmowania (dyrektywa `RULE`) jest nieodłączną częścią głównej pętli przetwarzania. Nie jest osobnym procesem działającym w tle — reguły są ewaluowane **synchronicznie**, w tej samej iteracji siatki czasowej co obliczenia `SELECT`. Daje to pewność, że alarm zawsze odnosi się do danych właśnie obliczonych, a nie z poprzedniego cyklu.

50.1 Miejsce RULE w cyklu przetwarzania

Przypomnijmy schemat funkcji `processRows()` opisanej w rozdziale Algorytm przeglądu drzewa zapytań. Dla każdego zapytania nie będącego deklaracją wykonywane są kolejno cztery kroki (Rys. 37):



Diagram

Rys. 37. Kolejność kroków przetwarzania jednego zapytania

Krok czwarty — `constructRulesAndUpdate()` — to właśnie wykonanie wszystkich reguł przypiętych do bieżącego zapytania. Wywoływany jest po zapisaniu wyników `SELECT` na dysk, co oznacza, że reguła zawsze ocenia **gotową, właśnie obliczoną próbkę** strumienia.

50.2 Ewaluacja warunku WHEN

Każda reguła zawiera listę tokenów opisujących wyrażenie logiczne (pole `condition` struktury `rule`). W momencie ewaluacji system:

1. Pobiera `outputPayload` bieżącego zapytania — to bieżąca próbka strumienia.

2. Przekazuje warunek do silnika `expressionEvaluator::eval()` — **tego samego silnika**, który oblicza wyrażenia `SELECT`.
3. Rzutuje wynik na wartość logiczną (`boolCast`): każda niezerowa wartość liczbowa to `true`, zero to `false`.

Jeśli warunek jest spełniony, wykonywana jest skojarzony z regułą akcja (`DO SYSTEM` lub `DO DUMP`). Jeśli niespełniony — reguła jest pomijana bez żadnych efektów ubocznych. Pełny przepływ przedstawia Rys. 38.

Rys. 38. Przepływ ewaluacji reguły

50.3 Akcja DO SYSTEM

Wywołanie `DO SYSTEM` jest najprostsze: system wywołuje `::system(polecenie)` bezpośrednio w wątku przetwarzania. Wywołanie jest **synchroniczne** — xretractor czeka na zakończenie procesu przed przejściem do następnej reguły.

Kod wyjścia polecenia jest sprawdzany: - `0` — sukces, brak wpisu w logu. - `0` — xretractor loguje błąd przez spdlog z kodem wyjścia. - Niepowodzenie `system()` (np. brak powłoki) — logowany jako błąd krytyczny.

Ostrzeżenie

Polecenie wykonywane jest synchronicznie. Długo trwające skrypty (np. wysyłanie dużych plików, wywołania sieciowe z timeoutem) opóźnią cały cykl przetwarzania. W takich przypadkach zaleca się uruchamianie procesu w tle: `DO SYSTEM 'mój_skrypt &'`.

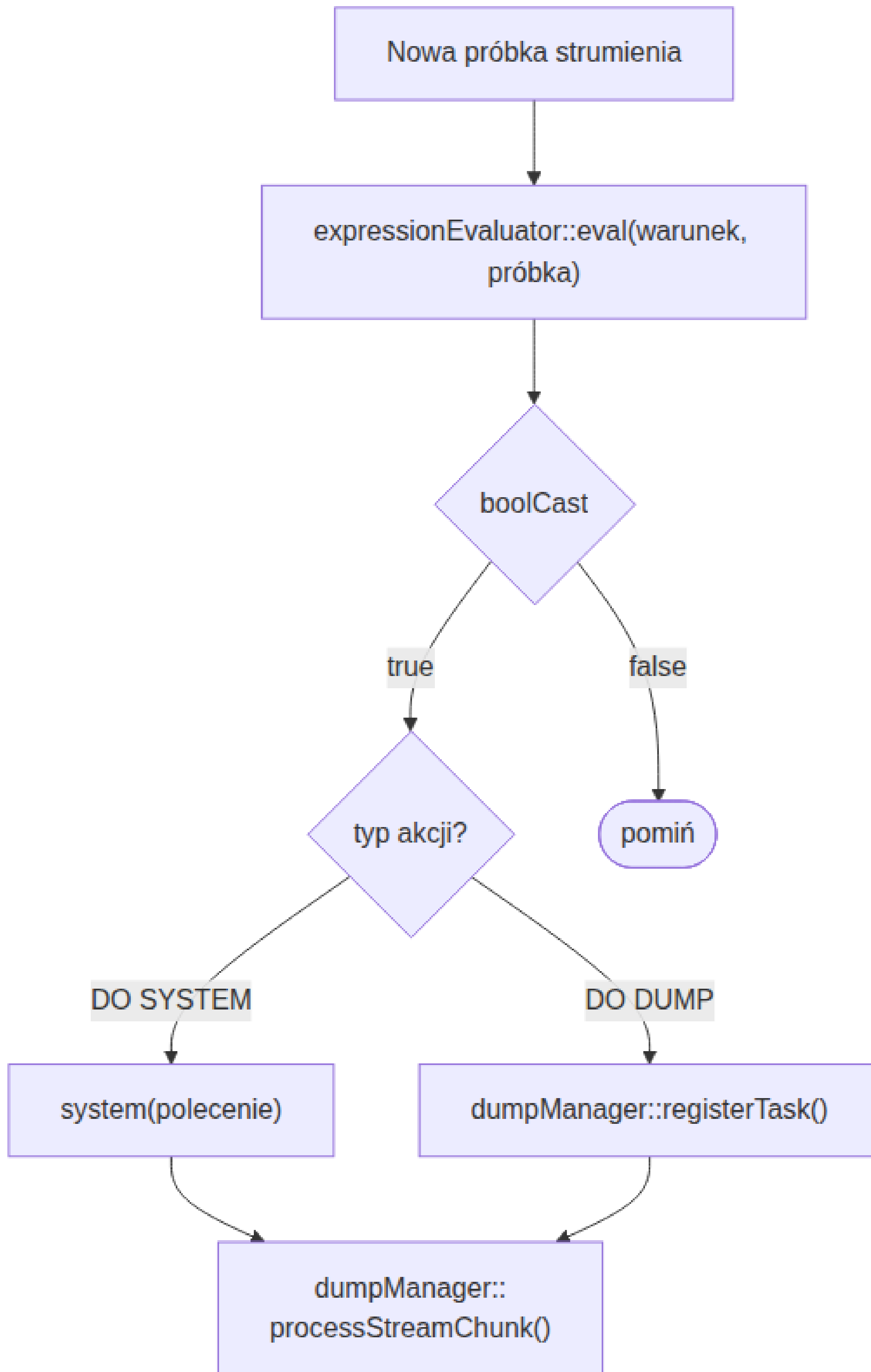
50.4 Akcja DO DUMP — szczegółowy algorytm

`DO DUMP` jest bardziej złożona, ponieważ wymaga zebrania danych **z przeszłości** (chwile przed zdarzeniem) i **z przyszłości** (chwile po zdarzeniu). Obsługuje to klasa `dumpManager`.

Faza 1: dane historyczne (przy rejestracji zadania)

W chwili wyzwolenia reguły — zaraz po stwierdzeniu, że warunek jest prawdziwy — `dumpManager::registerTask()`:

1. Tworzy plik docelowy na dysku (POSIX `open()` z flagą `O_CREAT | O_TRUNC`).
2. Jeśli `step_back < 0`, odczytuje `|step_back|` próbek z historycznego bufora strumienia.
   Dane historyczne istnieją, bo każdy strumień przechowuje okno poprzednich próbek niezbędne do obliczeń w oknach AGSE.



Diagram

3. Zapisuje próbki historyczne do pliku **od najstarszej do najnowszej** (tzn. od `step_back` do `-1`).
4. Oblicza, ile próbek z przyszłości jeszcze pozostało do zebrania (`dumpedRecordsToGo = |step_forward - step_back| - |step_back|`).
5. Jeśli `step_back  0` (opóźnienie startu), ustawia `delayDumpRecordsToGo = step_back`.

```
Przykład: DUMP -3 TO 2
  Przy rejestracji: zapisz próbki t-3, t-2, t-1  (history)
  Do zebrania z przyszłości: 2 próbki (t, t+1)
  dumpedRecordsToGo = 2
```

Faza 2: dane przyszłe (kolejne iteracje pętli)

Po rejestracji zadanie trafia do kolejki `bookOfTasks[streamName]`. W każdej kolejnej iteracji siatki czasowej (gdy strumień produkuje nową próbkę) wywoływane jest `dumpManager::processStreamChunk()`:

1. Dla każdego aktywnego zadania w kolejce (`dumpedRecordsToGo > 0`):
   - Jeśli `delayDumpRecordsToGo > 0` — dekrementuj i pomiń (opóźnienie startu).
   - Wpp. — zapisz bieżącą próbkę do pliku i dekrementuj `dumpedRecordsToGo`.
2. Gdy `dumpedRecordsToGo` osiągnie 0 — zamknij deskryptor pliku i usuń zadanie z kolejki.

Pełna sekwencja dla `DUMP -3 TO 2` przedstawiona jest na Rys. 39.

Rys. 39. Sekwencja zbierania danych przez DO DUMP -3 TO 2

Przypadek opóźnionego startu (step_back ≥ 0)

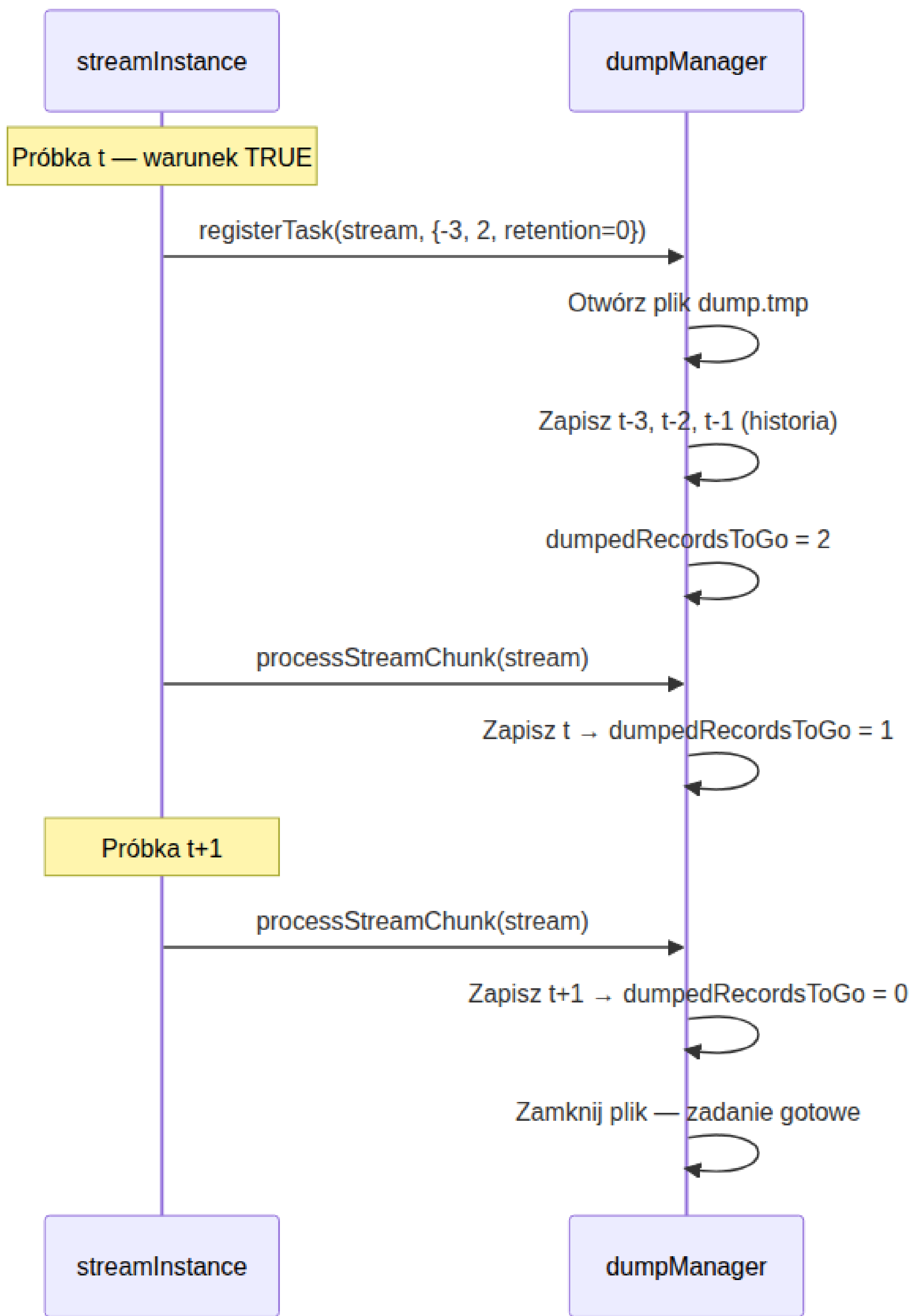
Gdy `step_back` jest nieujemny, zrzut nie zaczyna się od chwili zdarzenia, lecz od `step_back` próbek **po** zdarzeniu:

```
Przykład: DUMP 2 TO 5
  Przy rejestracji: delayDumpRecordsToGo = 2
  Próbka t   → pomiń (delay=2→1)
  Próbka t+1 → pomiń (delay=1→0)
  Próbka t+2 → zapisz (dumpedRecordsToGo = 3→2)
  Próbka t+3 → zapisz (dumpedRecordsToGo = 2→1)
  Próbka t+4 → zapisz (dumpedRecordsToGo = 1→0) - koniec
```

50.5 Retencja (RETENTION N)

Bez klauzuli `RETENTION` każde wyzwolenie reguły nadpisuje jeden plik `<strumień>_<reguła>_dump.tmp`. Pojemność kolejki `bookOfTasks` wynosi wtedy 1 — nowe zadanie wypycha stare (i zamyka jego deskryptor).

Z klauzulą `RETENTION N`: - Pojemność kolejki `bookOfTasks` ustawiana jest na `N`. - Numer pliku rotuje modulo `N`: `_dump_0.tmp`, `_dump_1.tmp`, …, `_dump_(N-1).tmp`. - Gdy `N`-te zada-



Diagram

nie trafia do kolejki, najstarsze (jeszcze niezakończone) jest **usuwane** — destruktor `dumpTask` zamyka otwarty deskryptor.

Oznacza to, że przy częstych zdarzeniach i małym `N` nieukończony zrzut może zostać przerwany. Wartość `N` powinna być dobrana tak, aby czas zbierania jednego zrzutu (`|step_back| + step_forward` cykli) był mniejszy niż interwał między zdarzeniami pomnożony przez `N`.

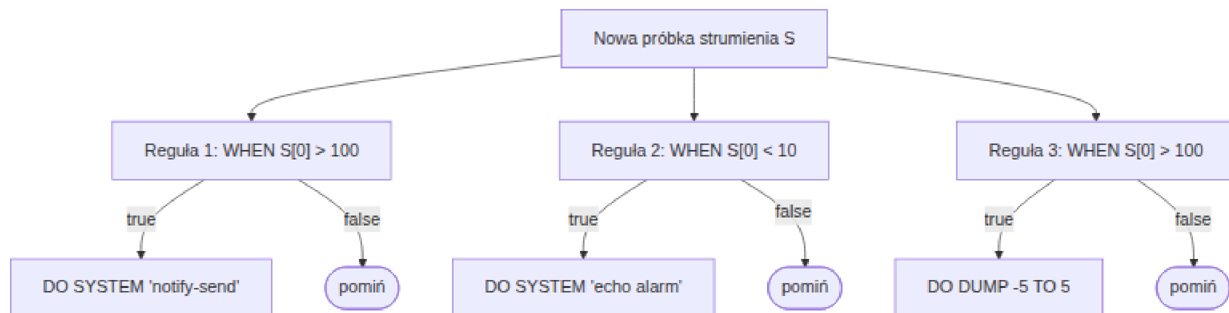
50.6 Format pliku zrzutu

Plik zawiera surowe rekordy binarne bez żadnego nagłówka — każdy rekord ma rozmiar określony przez deskryptor (`descriptor.getSizeInBytes()`). Format jest identyczny z formatem używanym przez artefakty strumienia, co pozwala odczytać go narzędziem `xtrdb` po ręcznym podaniu schematu:

```
$ xtrdb
> storage <ścieżka>
> open <strumień>_<reguła>_dump { <typ> <pole> }
> list
> quit
```

50.7 Wiele reguł — kolejność ewaluacji

Do jednego strumienia można przypiąć wiele reguł. Wszystkie ewaluowane są w jednej iteracji `constructRulesAndUpdate()`, w kolejności ich deklaracji w pliku `.rql`. Każda reguła jest niezależna — spełnienie jednej nie wpływa na ewaluację pozostałych (Rys. 40).



Diagram

Rys. 40. Niezależna ewaluacja wielu reguł na tym samym strumieniu

---

50.8 Ograniczenia i uwagi praktyczne

Sytuacja	Zachowanie
Warunek spełniony dwa razy z rzędu (np. pomiar stale powyżej progu)	Każda próbka rejestruje nowe zadanie DUMP — pliki nakładają się przy braku RETENTION
Strumień wejściowy `DECLARE` jako cel `ON`	Błąd kompilacji — reguły można podpiąć wyłącznie pod `SELECT`
Niedostateczna historia (bufor krótszy niż `|step_back|`)	Zapis zawiera tyle próbek, ile jest dostępnych; brak błędu
Plik docelowy niedostępny (brak katalogu STORAGE)	Błąd krytyczny `FatalError` — xretractor kończy działanie
DO SYSTEM zwraca niezerowy kod	Błąd w logu spdlog; przetwarzanie kontynuuje

Rozdział 51

Ruchome okno danych AGSE

Ruchome okno danych to pojęcie powszechnie stosowane w systemach przetwarzających strumienie lub serie czasowe. Idea polega na grupowaniu danych w oknach czasowych, dając możliwość użytkownikowi możliwość przetwarzania w zamrożonych migawkach.

RetractorDB wspiera ten model przetwarzania danych poprzez operator **AgSe** (Agregacja i Serializacja). Operator ten jest dwuargumentowy i działa na strumieniu. Oznaczany znakiem `@`, ma postać:

```
strumień@(k, w)
```

gdzie:

- **k** — skok okna (liczba naturalna): o ile rekordów źródłowych przesuwa się okno przy każdym kroku,
- **w** — rozmiar okna (liczba całkowita różna od zera): ile pól źródłowych zawiera jeden rekord wyjściowy.

Wartość ujemna `w` oznacza **agregację lustrzaną** — pola w rekordzie wyjściowym ułożone są w odwrotnej kolejności względem napływu.

51.1 Jak zmienia się interwał strumienia wyjściowego

Jeśli strumień źródłowy ma `W` pól w rekordzie i interwał `Δ`, to strumień wyjściowy operatora `@(k, w)` ma:

- `|w|` pól w rekordzie wyjściowym,
- interwał wyjściowy `Δ_out = (Δ / W) × k`.

Parametry	Efekt
`k = |w|`	okno tumbling — kolejne okna nie zachodzą na siebie
`k < |w|`	okno przesuwne (sliding) — kolejne okna zachodzą na siebie
`k > |w|`	próbkowanie z przerwami — część danych jest pomijana
`k = 1, |w| = 1`	serializacja — wielopolowy rekord rozbijany na jednoelementowe

Parametry	Efekt
`w < 0`	agregacja lustrzana — kolejność pól w oknie odwrócona

51.2 Typowe wzorce użycia

```
-- serializacja: 2 pola → 1 pole (interwał ÷ 2)
SELECT * STREAM s1 FROM A@(1,1)

-- tumbling window: okna po 4 rekordy, bez nakładania
SELECT * STREAM s2 FROM A@(4,4)

-- sliding window: okno 5-elementowe przesuwane o 1
SELECT * STREAM s3 FROM A@(1,5)

-- próbkowanie: co piąty rekord (skip=5, okno=1)
SELECT * STREAM s4 FROM A@(5,1)

-- deserializacja lustrzana: przywrócenie kolejności pól
SELECT * STREAM s5 FROM s1@(2,-2)
```

51.3 Wizualizacja operatora @

Poniżej schematyczne przedstawienie działania `source@(k, w)` dla strumienia jednoelementowego:

```
Dane wejściowe:   0  1  2  3  4  5  6  7  8  9  ...
                  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓

@(1, 3) - sliding window, skok=1, okno=3:
  [0,1,2]  [1,2,3]  [2,3,4]  [3,4,5]  ...

@(3, 3) - tumbling window, skok=3, okno=3:
  [0,1,2]           [3,4,5]           ...

@(5, 1) - próbkowanie co 5 elementów:
  [0]               [5]               ...

@(2,-2) - lustrzana, skok=2, okno=2:
  [1,0]    [3,2]    [5,4]    [7,6]    ...
```

51.4 Przykłady

Poniższe podrozdziały prezentują konkretne zastosowania operatora AgSe:

- **Przykład serializacji** — zamiana wielopolowego rekordu na sekwencję jednoelementowych rekordów i powrót przez agregację lustrzaną.
- **Przykład średniej ruchomej** — sliding window jako podstawa filtru uśredniającego sygnał.
- **Różne typy okien** — tumbling, sliding i próbkowanie na jednym strumieniu danych.

Na początku rozważymy proces serializacji w operatorze Agregacji i Serializacji – AgSe.

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `agse1`, `agse2`, `agse3`, `Pattern6` opisanych w załączniku pt. Testy Integracyjne.

Rozdział 52

Przykład serializacji

Na początku stwórzmy plik qplan3.rql o następującej zawartości:

```
DECLARE a BYTE, b BYTE STREAM A, 1 FILE 'data3.txt'
SELECT * STREAM str3 FROM A@(1,1)
```

Oraz przygotujmy plik data3.txt o następującej zawartości:

```
$ seq 0 9 | paste - -
```

```
0       1
2       3
4       5
6       7
8       9

```

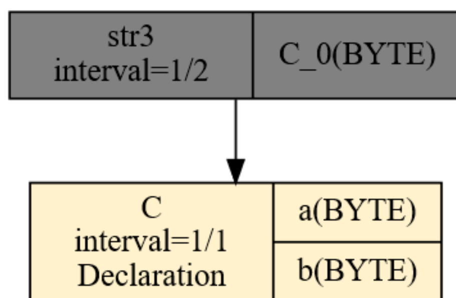
Ostatnia, pusta linia jest istotna i znacząca. Po uruchomieniu xretractor qplan3.rql a w drugim oknie xqry -s str3 ujrzymy coś zbliżonego:

```
$ xqry -s str3
7
8
9
0
1
2
3
4
5
6
```

To co widzimy to przykład serializacji. Ciekawy aspekt operatora Agse w tym przypadku widać również w planie realizacji zapytania. Możemy zajrzeć do niego za pomocą polecenia:

```
$ xretractor -c qplan3.rql -f -p -d > out.dot && dot -Tsvg out.dot -o out.svg
```

W pliku out.svg zobaczymy następujący plan realizacji zapytania (Rys. 41):



Rys. 41 Plan realizacji zapytania po kompilacji AGSE

Ze strumienia źródłowego, w którym co sekundę przychodzą dane zawierające dwa bajty – tworzony jest strumień danych w którym co pół sekundy pojawia się jeden bajt.

Skoro mamy już w systemie strumień danych str3 zwracający sekwencyjne liczby – możemy go wykorzystać do dalszych transformacji. Dodajmy do pliku qplan3.rql następujące zapytanie:

```
SELECT * STREAM str4 FROM str3@(2,2)
```

Po uruchomieniu nie ujrzymy jednak oczekiwanej źródłowej postaci pliku dane3.txt. Pojawi się natomiast coś podobnego:

```
$ xqry -s str4
2 1
4 3
6 5
8 7
0 9
2 1
4 3
6 5
```

Na drodze trudnej sztuki przetwarzania strumieni danych znajdują się pułapki. To jedna z nich. Dopiero jak czytelnik dobrze się przyjrzy, to zauważy że dane są odbite w lustrze. Proszę zamień to zapytanie na taką formę:

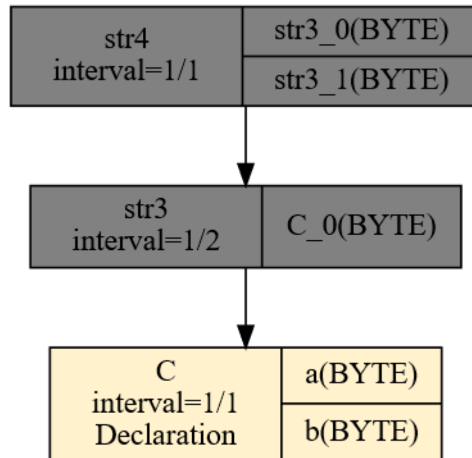
```
SELECT * STREAM str4 FROM str3@(2,-2)
```

Dopiero tak zbudowany strumień przedstawi postać źródłową widoczną w pliku dane3.txt:

```
$ xqry -s str4
3 4
5 6
7 8
9 0
1 2
3 4
```

Ten minus przy wskazaniu szerokości okna, to odbicie lustrzane. Rozmiar okna wynosi dwa, natomiast sekwencja pól jest zbudowana w odwrotnej kolejności.

Generując obraz planu zapytania realizujący najpierw serializację a potem deserializację ujrzymy następującą zależność (Rys. 42):



Rys. 42 SErializacja i DEserializacja

Zaprezentowano tutaj najbardziej podstawowy przykład zastosowania operatora ruchomego okna danych. Jeśli zaczniemy eksperymentować ze skokiem i rozmiarem okna, zauważymy, że jesteśmy w stanie stworzyć dowolne, przesuwające się okno nad strumieniem danych lub pominąć niektóre elementy budując skok większy od szerokości okna.

Zapis realizacji eksperymentu przedstawia się następująco:

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `agse1`, `agse2`, `agse3`, `Pattern6` opisanych w załączniku pt. Testy Integracyjne.

Rozdział 53

Przykład średniej ruchomej

Średnia ruchoma (ang. *moving average*) to jeden z najprostszych i najczęściej stosowanych filtrów sygnałowych. Każdy punkt wyjściowy jest średnią arytmetyczną `N` ostatnich próbek. Operator `@(1, N)` w RetractorDB tworzy dokładnie takie okno: dla każdego nowego pomiaru dostępne jest `N` ostatnich wartości.

53.1 Dane źródłowe

Przyjmijmy strumień temperatury mierzonej co sekundę. Plik `temp.txt` zawiera kolejne odczyty:

```
$ seq 10 5 60 > temp.txt
```

```
10
15
20
25
30
35
40
45
50
55
60
```

53.2 Zapytanie RQL

Plik `avg.rql`:

```
DECLARE temp INTEGER
STREAM sensor, 1
FILE 'temp.txt'

SELECT *
```

```
STREAM window5
FROM sensor@(1,5)

SELECT window5[0]+window5[1]+window5[2]+window5[3]+window5[4]
STREAM sumRow
FROM window5

SELECT sumRow[0]/5
STREAM avg5
FROM sumRow
```

Co robi każde zapytanie

1. `sensor@(1,5)` — tworzy przesuwne okno 5-elementowe. Każdy rekord `window5` zawiera 5 ostatnich odczytów temperatury. Interwał wyjściowy: `1s / 1 × 1 = 1s` (skok=1, W=1 pole).
2. Suma pięciu pól — klasyczne `SELECT` po polach `window5[0]..window5[4]`.
3. Podzielenie sumy przez 5 — wynik to średnia ruchoma.

53.3 Uruchomienie

```
$ xretractor avg.rql &
$ xqry -s avg5
```

Przykładowy wynik (okno wypełnia się po pierwszych 5 próbkach):

```
30
35
40
45
50
```

Wartość `30` odpowiada średniej z pierwszego pełnego okna: `(10+15+20+25+30)/5 = 20...` uwaga — system RetractorDB nie wyświetla niepełnych okien, więc pierwsze pojawienie się wyniku odpowiada chwili gdy okno jest w pełni nasycone danymi.

53.4 Weryfikacja planu zapytania

```
$ xretractor -c avg.rql -f -p -d > out.dot && dot -Tsvg out.dot -o out.svg
```

W wygenerowanym planie widać łańcuch: `sensor → window5 → sumRow → avg5`. Kluczowy jest węzeł `sensor@(1,5)` — z jednoelementowego strumienia wchodzącego co sekundę powstaje strumień pięcioelementowy, ciągle przesuwany.

53.5 Zależność między parametrami okna a opóźnieniem

Średnia ruchoma wprowadza opóźnienie o połowę długości okna. Dla okna N=5 opóźnienie wynosi 2 próbki (2 sekundy). Zwiększenie okna:

- zmniejsza szum (większe wygładzenie),
- zwiększa opóźnienie,
- nie zmienia interwału wyjściowego (przy stałym skoku k=1).

Zmiana skoku przy stałym oknie:

```
sensor@(5,5)   -- tumbling: wynik co 5 sekund, bez nakładania
sensor@(1,5)   -- sliding:  wynik co sekundę, pełne nakładanie
sensor@(3,5)   -- częściowe nakładanie: wynik co 3 sekundy
```

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `agse1`, `agse2`, `agse3`, `Pattern6` opisanych w załączniku pt. Testy Integracyjne.

Rozdział 54

Różne typy okien

Operator `@(k, w)` przez dobór dwóch parametrów pozwala zbudować każdy z klasycznych typów okien stosowanych w przetwarzaniu strumieni. Poniżej zestawienie wzorców na jednym wspólnym strumieniu źródłowym.

54.1 Strumień źródłowy

Plik `data.txt` — 12 kolejnych liczb całkowitych:

```
$ seq 1 12 > data.txt
```

Deklaracja źródła — jeden rekord co sekundę, jedno pole:

```
DECLARE val INTEGER
STREAM src, 1
FILE 'data.txt'
```

54.2 Tumbling window — okna bez nakładania

Skok równy rozmiarowi okna: `k = w`. Każdy element wejściowy należy dokładnie do jednego okna wyjściowego.

```
SELECT *
STREAM tumbling
FROM src@(4,4)
```

Interwał wyjściowy: `1s × 4 / 1 = 4s`. Rekordy wyjściowe:

```
$ xqry -s tumbling
1  2  3  4
5  6  7  8
9 10 11 12
```

Zastosowania: agregacja próbek w stałych przedziałach czasu (np. minutowe, godzinowe).

54.3 Sliding window — okna z nakładaniem

Skok mniejszy od rozmiaru okna: `k < w`. Każdy element wejściowy pojawia się w kilku kolejnych oknach.

```
SELECT *
STREAM sliding
FROM src@(1,4)
```

Interwał wyjściowy: `1s × 1 / 1 = 1s`. Rekordy wyjściowe:

```
$ xqry -s sliding
1  2  3  4
2  3  4  5
3  4  5  6
4  5  6  7
...
```

Zastosowania: średnia ruchoma, detekcja trendów, filtry FIR (jak w implementacji filtru sygnałowego).

54.4 Próbkowanie — okna z przerwami

Skok większy od rozmiaru okna: `k > w`. Część elementów wejściowych jest pomijana.

```
SELECT *
STREAM sampled
FROM src@(3,1)
```

Interwał wyjściowy: `1s × 3 / 1 = 3s`. Rekordy wyjściowe:

```
$ xqry -s sampled
1
4
7
10
```

Zastosowania: decimacja sygnału, redukcja częstotliwości próbkowania, diagnostyka co N-ty pomiar.

54.5 Okno lustrzane — odwrócona kolejność pól

Ujemna wartość `w` odwraca kolejność pól w rekordzie wyjściowym przy zachowaniu tego samego rozmiaru okna.

```
SELECT *
STREAM mirrored
FROM src@(2,-2)
```

Interwał wyjściowy: `1s × 2 / 1 = 2s`. Rekordy wyjściowe (kolejność pól odwrócona):

```
$ xqry -s mirrored
2  1
4  3
6  5
8  7
...
```

Porównaj z `src@(2,2)`, które dałoby `1 2`, `3 4`, `5 6`... — kolejność zgodna z napływem. Agregacja lustrzana jest niezbędna przy odwracaniu serializacji (deserializacja), jak opisano w przykładzie serializacji.

54.6 Zestawienie wzorców

Zapytanie	Typ okna	Interwał	Rozmiar rekordu	Nakładanie
`src@(4,4)`	tumbling	4 s	4 pola	brak
`src@(1,4)`	sliding	1 s	4 pola	pełne
`src@(2,4)`	hop window	2 s	4 pola	częściowe
`src@(3,1)`	próbkowanie	3 s	1 pole	brak
`src@(2,-2)`	lustrzane	2 s	2 pola	brak

54.7 Plan realizacji zapytań

Wszystkie cztery warianty można uruchomić jednocześnie umieszczając je w jednym pliku `.rql`:

```
DECLARE val INTEGER STREAM src, 1 FILE 'data.txt'

SELECT * STREAM tumbling FROM src@(4,4)
SELECT * STREAM sliding  FROM src@(1,4)
SELECT * STREAM sampled  FROM src@(3,1)
SELECT * STREAM mirrored FROM src@(2,-2)
```

```
$ xretractor -c windows.rql -f -p -d > out.dot && dot -Tsvg out.dot -o out.svg
```

Plan zapytania pokaże cztery niezależne gałęzie wywodzące się ze wspólnego węzła `src`. Każda gałąź realizuje inny typ okna bez wzajemnych zależności.

Uwaga

Opisana funkcjonalność ma pokrycie w testach: `agse1`, `agse2`, `agse3`, `Pattern6` opisanych w załączniku pt. Testy Integracyjne.

Rozdział 55

Odtwarzanie strumienia

Serie czasowe bardzo często rozumiemy jako dane oznaczone znacznikami czasowymi. Dane zachowane np. w pliku możemy dowolnie przetwarzać – zachowując ich kolejność w oparciu o zarejestrowane zależności czasowe. System RetractorDB został wyposażony w możliwość ponownego wyemitowania takiego strumienia zachowując zarejestrowane zależności czasowe, tak jakby faktycznie ponownie te dane napływały.

W celu przedstawienia przykładu przygotujmy plik tekstowy wypełniony danymi np. od 30 do 45.

```
$ seq 30 45 > dane.txt
```

Tak przygotowany plik będziemy odtwarzać w systemie RetractorDB.

W kolejnym kroku stwórzmy następujący plik wypełniony zapytaniami dla systemu – query.rql zawierający tylko jedną deklarację zakończoną HOLD.

```
DECLARE a INTEGER STREAM core, 1 FILE 'dane.txt' HOLD
```

Uruchamiamy w jednym oknie polecenie:

```
$ xretractor query.rql
```

W kolejnym wydajemy polecenie:

```
$ xqry -s core
0
0
0
…
```

Zobaczymy ciąg zer …

W kolejnym oknie wydajemy następujące polecenie:

```
$ xqry -a "SELECT * STREAM ping FROM core VOLATILE"
snd: adhoc SELECT * STREAM ping FROM core VOLATILE
rcv: db OK
```

W tym momencie w oknie prezentującym wartości ze strumienia core pojawią się wartości core

```
$ xqry -s core
0
0
0
…
0
0
0
30
31
32
33
…
```

Nagrany przykład poniżej (Rys. 43):

Rys. 43 Nagrany przykład odtwarzania strumienia

Rozdział 56

Przykłady zastosowań

W niniejszym rozdziale zostaną przedstawione krótkie przykłady zastosowania systemu RetractorDB w rozwiązaniu konkretnych zagadnień spotykanych w konstrukcjach systemów monitorowania.

Rozdział 57

Implementacja filtru sygnałowego

Zagadnienia związane z przetwarzaniem sygnałów cyfrowych zawierają w sobie problemy związane z filtracją. Celem filtracji jest rozdzielenie informacji zawartych wewnątrz sygnału. Zazwyczaj celem jest oddzielenie sygnału od jego zakłóceń.

Filtry mogą być analogowe oraz cyfrowe. W ramach proponowanego rozwiązania skupimy się na filtrach cyfrowych. Filtr cyfrowy implementowany jest jako ciąg operacji na kolejnych danych przetwarzanego sygnału w danym oknie czasowym. Z reguły dobierając filtr cyfrowy musimy zdecydować na jakie kompromisy musimy się zgodzić. Dodatkowo, możemy trafić na zabezpieczenia prawne związane z niektórymi algorytmami lub metodami [9].

Projektowanie filtru w Octave

Projektując filtr cyfrowy musimy określić jaki zakres częstotliwości chcemy wytłumić a jaki wzmocnić lub pozostawić nienaruszony. Parametry te określamy jako pasmo zaporowe i przepustowe. Jednym ze znanych mi narzędzi używanych do konstrukcji filtrów cyfrowych jest program GNU Octave (https://octave.org). Za pomocą tego narzędzia możemy wygenerować wymagane współczynniki do obliczeń prostego cyfrowego filtru sygnałowego.

Dla przykładu przyjmiemy następujące wartości wymagane do konstrukcji filtru sygnałowego:

- Częstotliwość próbkowania sygnału wejściowego 50Hz
- Pasmo przepustowe 0-2Hz
- Pasmo zaporowe 5-25Hz

Częstotliwość próbkowania sygnału wejściowego 50Hz oznacza że 50 próbek pojawi się w ciągu sekundy. W systemie RetractorDB oznacza to że sygnał źródłowy powinien napływać z szybkością Delta = 0,02. I taką częstotliwość powinno wspierać zdefiniowane źródło danych.

Dla takich założeń filtra sygnałowego program w języku Octave tworzący filtr sygnałowy przedstawia się następująco:

```
pkg signal load
```

```
filtord = 25 % Długość filtru
Fs = 50;     % Częstotliwość próbkowania 50Hz
FNq = Fs/2;  % Częstotliwość Nyquista
F1c = 2;     % Pasmo przepustowe 0 - 2Hz
F2c = 5;     % Pasmo zaporowe 5 Hz ->
F3c = 25;    % Pasmo zaporowe <- 25 Hz
f=[0,F1c/FNq,F2c/FNq,F3c/FNq]
m = [ 1 , 1 , 0, 0 ]
freqz ( remez(filtord,f,m) );
```

Tak przygotowany plik powinniśmy zapisać na dysku lub wkopiować bezpośrednio do okna terminala programu Octave.

Parametry zmiennoprzecinkowe filtru możemy wyświetlić wydając polecenie remez(filtord,f,m). Graficzną charakterystykę filtru uzyskamy wydając następujące polecenie:

```
octave:1> [h, w] = freqz ( remez(filtord,f,m) );
subplot(2,1,1);
plot (f, m, '', w/pi, abs (h), '');
xlabel('Znormalizowana częstotliwość')
ylabel('wzmocnienie')
grid on
subplot(2,1,2);
plot(f,20*log10(m+1e-5),'', w/pi,20*log10(abs(h)),'');
xlabel('Znormalizowana częstotliwość')
ylabel('wzmocnienie (dB)')
grid on
```

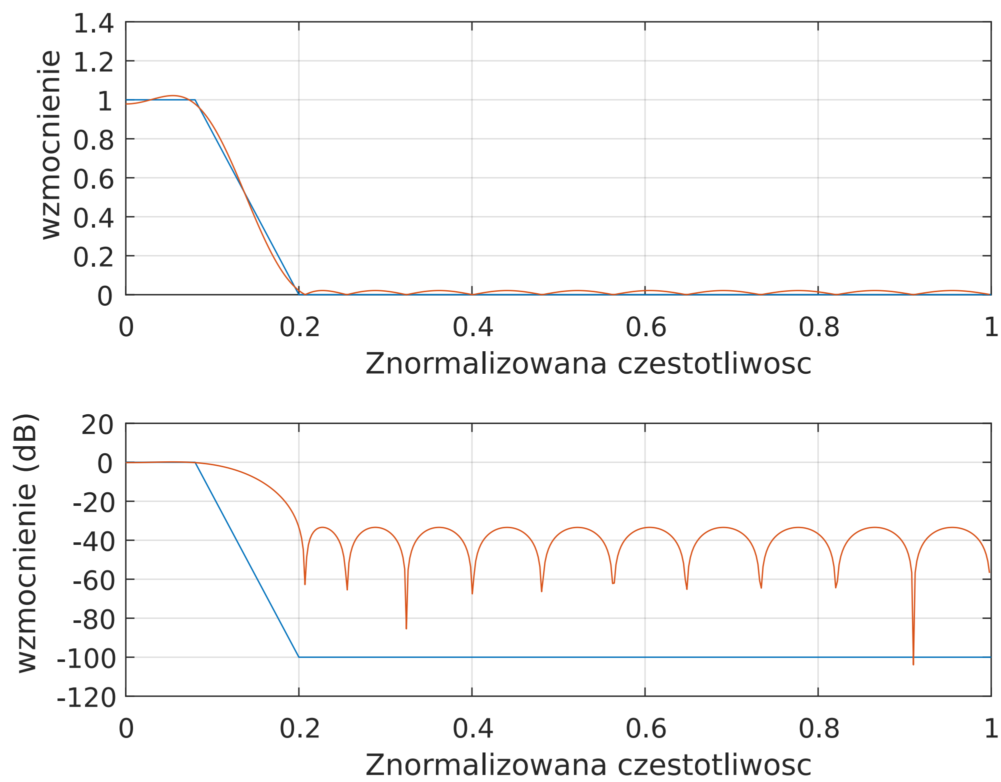
Uruchomienie powyższego kodu w programie Octave zaprezentuje następującą odpowiedź w postaci graficznej (Rys. 44):

Na osi rzędnych Octave przedstawia znormalizowaną częstotliwość. Zakres prezentowanej na rysunku częstotliwości na osi rzędnych od 0 do 1 odpowiada częstotliwości od 0Hz do 25Hz. Na osi odciętych pierwszy rysunek prezentuje liniowe wzmocnienie, drugi tą samą wielkość ale w skali logarytmicznej.

Parametry filtru można wyświetlić za pomocą polecenia:

```
octave:11> remez(filtord,f,m)
ans =
  -4.2689e-03
  -2.0148e-02
  -1.4865e-02
  -1.8188e-02
  -1.4031e-02
  -4.5861e-03
…
```

Chcąc otrzymać stałoprzecinkowe parametry 16 bitowego filtru należy wydać polecenie:



Rys. 44 Reprezentacja graficzna w dziedzinie częstotliwości wyznaczonego filtru cyfrowego

```
octave:12> floor(remez(filtord,f,m) * 32767)
ans =
   -140
   -661
   -488
   -596
   -460
```

Implementacja w systemie RetractorDB

Uzyskane wartości powinniśmy przenieść do pliku tekstowego o nazwie filterremez.txt

Dla celów testowych sygnał źródłowy pobierzemy z generatora liczb pseudolosowych. Dane efemeryczne pobierzemy bezpośrednio ze źródła z częstotliwością 50Hz.

Początkowa część pliku query.rql zapytania zawierająca deklaracje źródeł dla systemu RetractorDB przedstawia się następująco:

```
DECLARE coef INTEGER[25]
STREAM filter, 1
FILE 'filterremez.txt'

DECLARE data BYTE
STREAM source, 0.02
FILE '/dev/urandom'
```

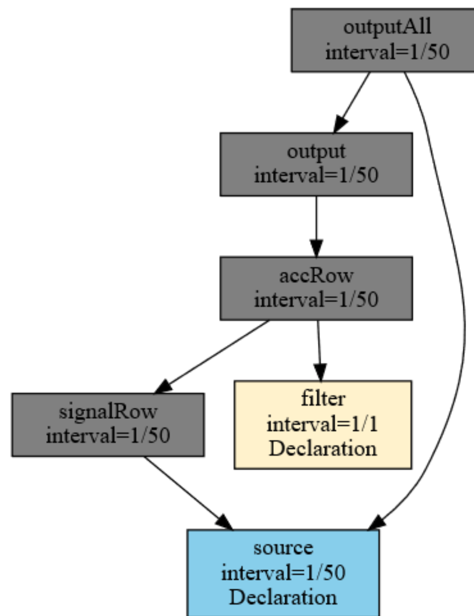
W kolejnej części znajdziemy polecenia tworzące proces przetwarzania sygnałów.

```
SELECT *
STREAM signalRow
FROM source@(1,25)

SELECT signalRow[_] * filter[_]
STREAM accRow
FROM signalRow+filter

SELECT accRow[0]
STREAM output
FROM accRow.sumc

SELECT (output[0]/25)/1000,source[0]
STREAM outputAll
FROM output+source
```

Widzimy tutaj 4 zapytania. Przeglądając rozdział dotyczący rozwijania symbolu _ nie powinno na zdziwić że próba podejrzenia wyniku kompilacji tego pliku przewinie nam kilka ekranów. Możliwy do szybkiej analizy podgląd zachodzącego procesu możemy uzyskać wydając polecenie:

```
$ xretractor -c query.rql -p -d > out.dot && dot -Tsvg out.dot -o out.svg
```

Ujrzymy następujący obraz (Rys. 45):



Rys. 45 Zależność przetwarzanych strumieni danych w trakcie realizacji filtru sygnałowego

Uruchomienie

Próba podejrzenia zawartych pól oraz typów danych spowoduje rozszerzenie wygenerowanego rysunku na tyle, że niemożliwe jest załączenie tutaj wygenerowanego wyniku bez utraty czytelności.

Pragnąc podejrzeć proces przetwarzania sygnałów w czasie rzeczywistym powinniśmy wydać następujący ciąg poleceń:

- w pierwszym oknie uruchomić proces serwera przetwarzający zgromadzone dane. Powinny się w tym katalogu znajdować pliki query.rql oraz filterremez.txt za pomocą polecenia

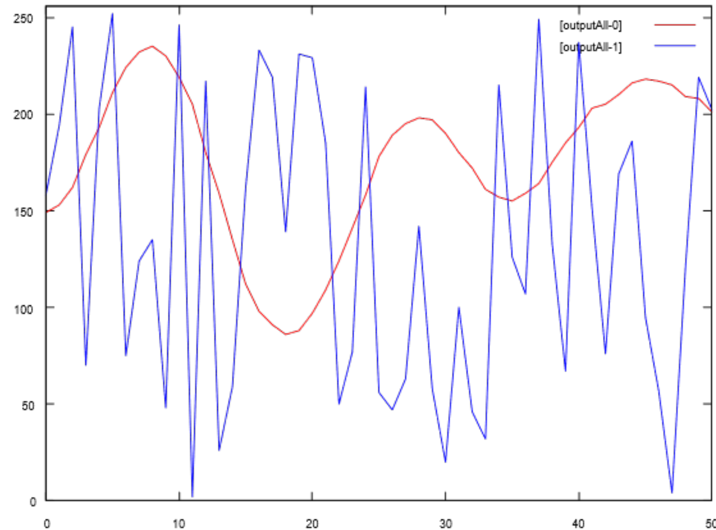
```
$ xretractor query.rql
```

- w drugim oknie wydać należy następujące polecenie:

```
$ xqry -s outputAll -p 50:256 | gnuplot
```

Na ekranie powinniśmy ujrzeć następujący wykres biegnący z lewa na prawo wypełniany na bieżąco danymi (Rys. 46):

Na Rys. 46 widzimy dwa wykresy nałożone na siebie. Ten bardziej zróżnicowany – na ekranie komputera widoczny jako niebieska linia zawierająca dużą zmienność to wizualizacja sygnału wejściowego. Dane pobrane z generatora liczb pseudolosowych z częstotliwością 50 próbek na sekundę. Oraz drugi wykres opływający dane wejściowe – na ekranie komputera prezentowany w kolorze czerwonym, bardziej łagodny, opływający – to właśnie dane przefiltrowane opracowanym filtrem sygnałowym. Sygnał, którego pasmo przepustowe zostało ograniczone do 0-2Hz (niskich częstotliwości) i ograniczone zaporowo w obszarze (5-25Hz) w obszarze wysokich częstotliwości. Ob-



Rys. 46 Filtracja sygnału zrealizowana wewnątrz RetractorDB

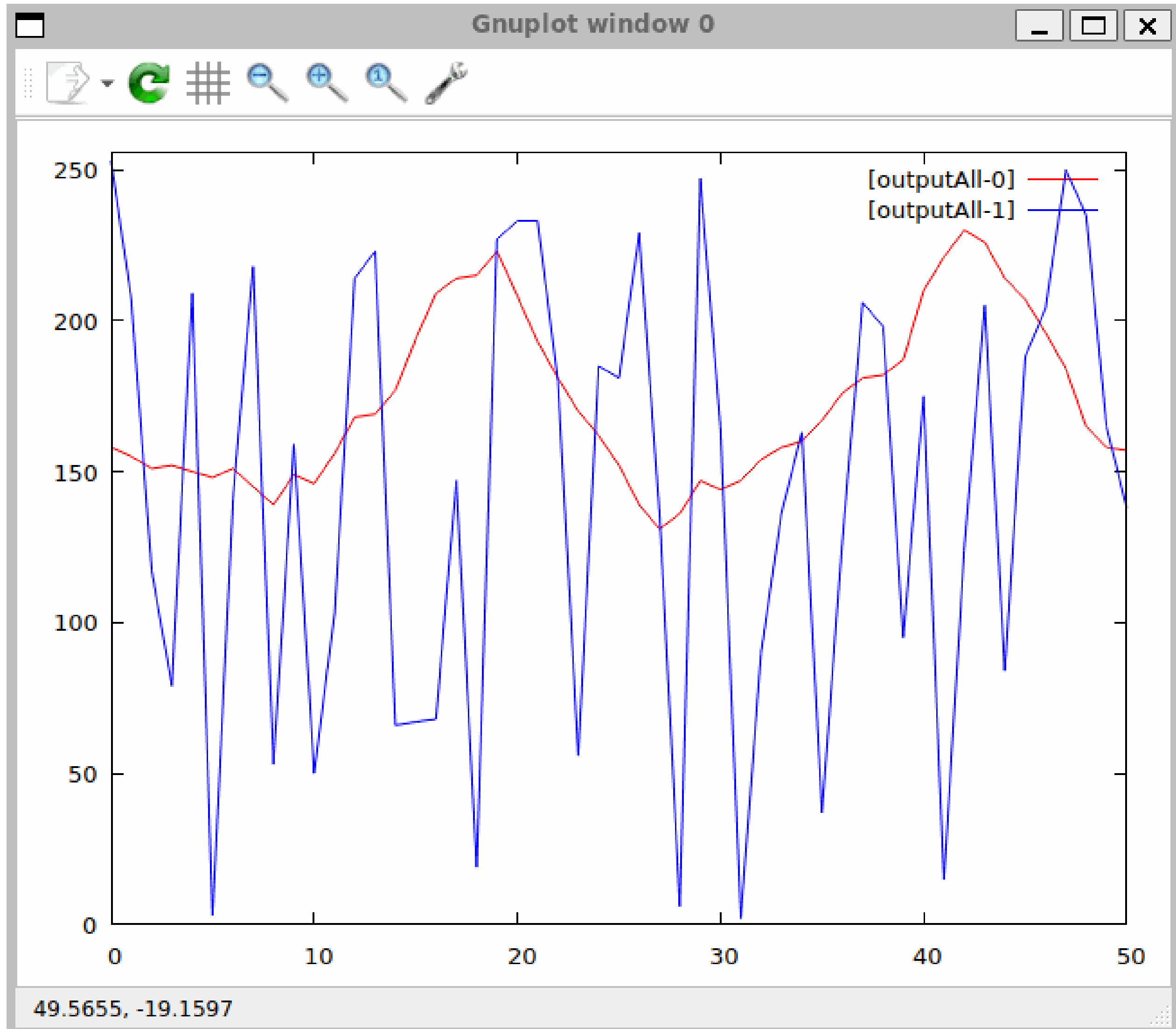
razowo można powiedzieć, że wyizolowaliśmy linię melodyczną dla basów.

Należy pamiętać, że na ekranie komputera ten wykres przesuwa się w prawo bardzo szybko, prezentując obraz możliwości bieżącego przetwarzania danych realizowanych w systemie RetractorDB.

Zapis ekranu w trakcie realizacji procesu przetwarzania ekranu:

Uwaga

Opisana funkcjonalność ma pokrycie w teście: `dsp` opisanym w załączniku pt. Testy Integracyjne.



Rys. 47 Animacja procesu filtracji sygnału w czasie rzeczywistym

Rozdział 58

Wizualizacja EKG i Detekcja Arytmii — baza MIT-BIH

58.1 Źródło danych — PhysioNet MIT-BIH Arrhythmia Database

Baza MIT-BIH Arrhythmia Database jest publicznie dostępnym zbiorem nagrań elektrokardiograficznych opublikowanym przez PhysioNet pod adresem:

`https://physionet.org/content/mitdb/1.0.0/`

Zawiera 48 półgodzinnych nagrań dwukanałowych zebranych od 47 pacjentów w Beth Israel Hospital w Bostonie w latach 1975–1979. Nagrania zostały manualnie zaadnotowane przez co najmniej dwóch niezależnych kardiologów i są szeroko stosowane w badaniach nad automatyczną detekcją arytmii.

Rekord 205

Przykład korzysta z rekordu **205** — nagrania 59-letniego mężczyzny leczonego Digoksyną i Quinaglutem. Rekord zawiera przypadki częstoskurczu komorowego (VT) i jest często cytowany w literaturze jako trudny diagnostycznie ze względu na dwie morfologicznie odmienne formy dodatkowych pobudzeń komorowych (PVC).

Parametry nagrania:

Parametr	Wartość
Czas trwania	≈ 30 min
Częstotliwość próbkowania	360 Hz
Liczba próbek	650 000
Kanał 1 (MLII)	Zmodyfikowane odprowadzenie kończynowe II
Kanał 2 (V1)	Odprowadzenie przedsercowe V1
Rozdzielczość	12 bitów, wzmocnienie 200 LSB/mV, punkt zerowy 1024

Wartości surowe przechowywane są jako liczby całkowite bez jednostek (tzw. wartości ADC). Przeliczenie na miliwolty:

$$\mathrm{mV} = \frac{\mathrm{ADC} - 1024}{200}$$

Zakres rzeczywistych wartości w pliku rec205 mieści się w przedziale 589–1315 (MLII) i 718–1106 (V1), co odpowiada amplitudzie sygnału w granicach ok. ±1,5 mV.

58.2 Przygotowanie danych

Oryginalne pliki nagrania (`205.hea`, `205.dat`, `205.atr`) są dostarczane w formacie MIT-BIH i wymagają konwersji do formatu binarnego rozpoznawanego przez RetractorDB.

Format MIT-BIH 212

Sygnał w pliku `205.dat` jest spakowany dwanaście-bitowo w formacie 212: każde trzy bajty przechowują dwie kolejne próbki obu kanałów według schematu:

```
[B0][B1][B2] → MLII = B0  | ((B1 & 0x0F) << 8)
               V1   = B2  | ((B1 >>  4)  << 8)
```

Wartości są 12-bitowe ze znakiem (zakres –2048..2047).

Konwersja do formatu RetractorDB

Skrypt `examples/ecg/mitbih2rdb.py` czyta nagłówek `205.hea`, dekoduje pary próbek i zapisuje je jako rekordy `int32` little-endian do pliku `rec205`:

```
650 000 rekordów × 2 pola × 4 bajty = 5 200 000 bajtów
```

Jednocześnie skrypt generuje skrypt RQL odtwarzający sygnał (`rec205-replay.rql`). Plik deskryptora `rec205.desc` jest tworzony przez `build.sh`.

Całość przygotowania uruchamia się jednym poleceniem z katalogu głównego projektu:

```bash
bash examples/ecg/build.sh
```

Wynikiem są trzy pliki w katalogu `examples/ecg/rec205/`:

Plik	Generuje	Opis
`rec205`	`mitbih2rdb.py`	Dane binarne (int32 LE)
`rec205.desc`	`build.sh`	Deskryptor strumienia
`rec205-replay.rql`	`mitbih2rdb.py`	Skrypt RQL odtwarzania

58.3 Zapytanie RQL

Plik `rec205-replay.rql` definiuje dwa strumienie:

```
DECLARE MLII INTEGER, V1 INTEGER STREAM ecg, 1/360 FILE 'rec205'

SELECT ecg.MLII, ecg.V1 STREAM s205out FROM ecg VOLATILE
```

Klauzula `STREAM ecg, 1/360` określa interwał czasowy jednej próbki jako 1/360 s, co odpowiada rzeczywistej częstotliwości próbkowania 360 Hz. Klauzula `TYPE DEVICE` w deskryptorze powoduje, że plik `rec205` jest czytany sekwencyjnie w pętli (po ostatniej próbce odczyt wraca do początku), co umożliwia ciągłe odtwarzanie nagrania.

Strumień wyjściowy `s205out` jest zadeklarowany jako `VOLATILE`, dlatego nie jest zapisywany na dysk — dane trafiają wyłącznie do procesu konsumenta (`xqry`).

58.4 Wizualizacja na ekranie

Do wyświetlenia wykresu w czasie rzeczywistym służy cel `ecg` w systemie budowania. Uruchamia on skrypt `scripts/xplot.sh`, który startuje `xretractor` w tle, a następnie przepuszcza strumień danych przez `xqry` do `gnuplot`.

```
# z katalogu build/Debug
ninja ecg
```

Wywołanie rozwijane przez CMake:

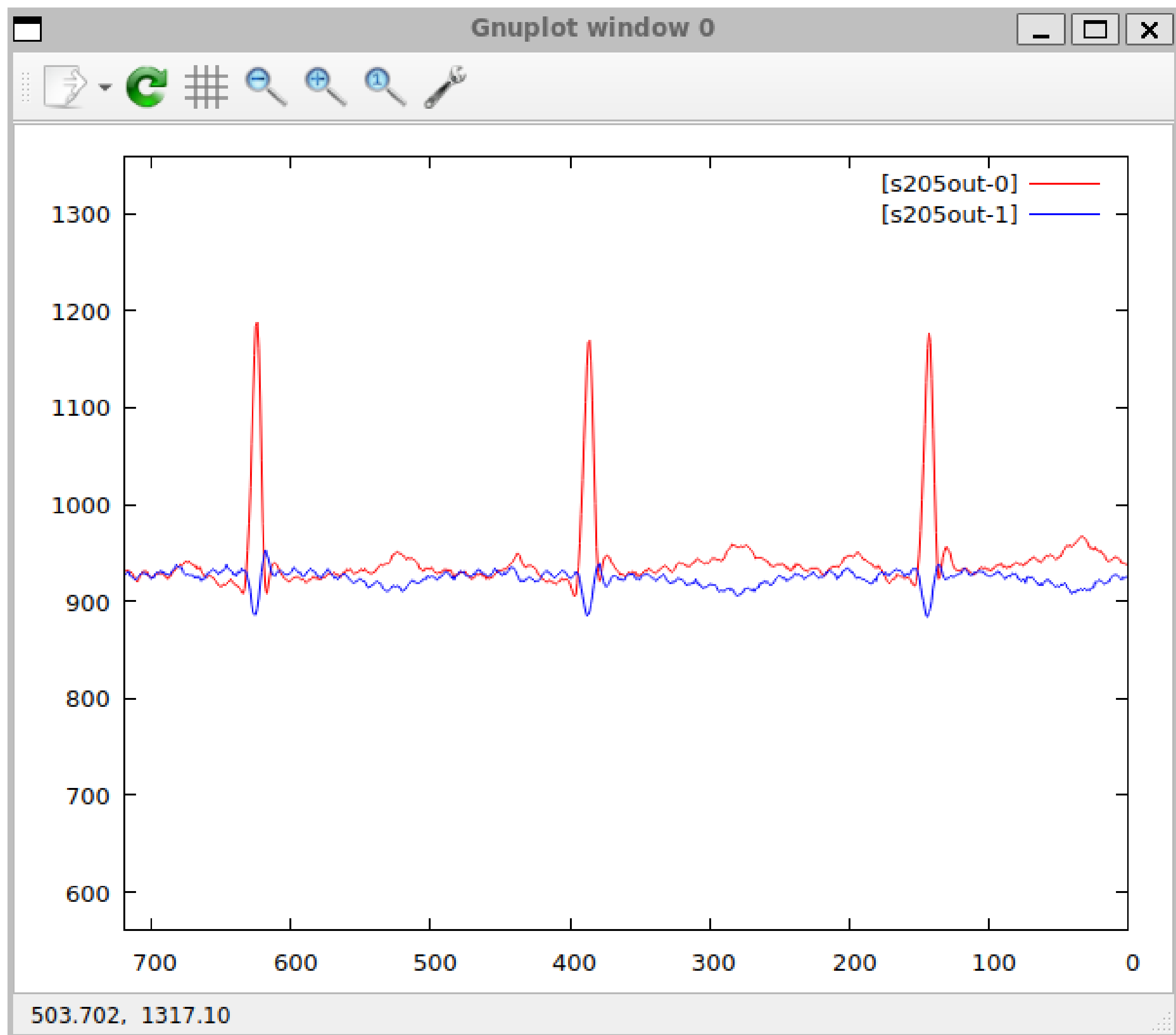
```
scripts/xplot.sh s205out rec205-replay.rql 720,560,1360 --gnuplot-rtl
```

Znaczenie parametrów:

Parametr	Znaczenie
`s205out`	Nazwa strumienia wynikowego
`rec205-replay.rql`	Plik zapytań
`720`	Szerokość okna danych (próbki widoczne jednocześnie)
`560,1360`	Zakres osi Y (wartości ADC pasujące do rzeczywistego sygnału)
`--gnuplot-rtl`	Najnowsze próbki po prawej stronie, wykres przesuwa się od prawej do lewej

Opcja `--gnuplot-rtl` jest parametrem `xqry` powodującym odwrócenie osi X gnuplota (`set xrange [720:0]`). Efekt jest taki, że najświeższe próbki pojawiają się po prawej stronie okna, a starsze przesuwają się w lewo — analogicznie do klasycznego wydruku EKG na taśmie papierowej.

Okno prezentuje 720 próbek, czyli dokładnie 2 sekundy sygnału przy 360 Hz, co odpowiada typowej szerokości jednego paska EKG używanej w diagnostyce.



Rys. 48 Widok okna gnuplot z odtwarzanym sygnałem EKG (rekord 205)

58.5 Detekcja QRS i identyfikacja arytmii

Kontekst — algorytm Pan-Tompkins

Detekcja zespołów QRS jest fundamentem automatycznej analizy EKG. Zespół QRS reprezentuje depolaryzację komór serca i odpowiada każdemu uderzeniu serca widocznemu jako ostry pik w sygnale. Znając położenia QRS w czasie, można wyliczyć interwały RR, a na ich podstawie rozpoznać podstawowe zaburzenia rytmu:

Miara pochodna od QRS	Zastosowanie
Interwały RR	Częstotliwość akcji serca (HR), VT, bradykardia
Zmienność RR (HRV)	Autonomiczny układ nerwowy, przewidywanie zdarzeń
Morfologia QRS	Rozróżnienie PVC od normalnego rytmu, APC
Czas trwania QRS	Blok odnogi pęczka Hisa (BBB)

Algorytm Pan-Tompkins (1985) jest klasycznym, pięcioetapowym potokowym algorytmem cyfrowego przetwarzania sygnałów realizowanym za pomocą filtrów FIR. RetractorDB implementuje go bezpośrednio jako strumień zapytań RQL, bez specjalistycznych bibliotek DSP.

Generowanie filtrów sygnałowych (coef)

Algorytm wymaga dwóch zestawów współczynników FIR, przechowywanych jako pliki tekstowe (`bp_coef.txt`, `d_coef.txt`). Generowane są jednorazowo skryptami Pythona przed uruchomieniem detekcji.

Filtr pasmowoprzepustowy — `gen_bp_coef.py`

Krok 1 algorytmu wymaga filtru wycinającego szumy i artefakty poza pasmem QRS. Pasm przepustowe 5–15 Hz przy fs = 360 Hz daje odpowiedź zawierającą morfologię QRS przy jednoczesnym tłumieniu linii bazowej (< 5 Hz) i szumów mięśniowych (> 15 Hz).

Metoda projektowania to **okienkowy sinc** (windowed sinc):

```
h_bp[n] = (h_lp2[n] - h_lp1[n]) · w[n]
```

gdzie:

- `h_lp[n] = 2·fc·sinc(2·fc·(n-M))` — idealny filtr dolnoprzepustowy
- `w[n] = 0,54 - 0,46·cos(2 n/(N-1))` — okno Hamminga tłumiące efekty Gibbsa
- `M = (N-1)/2 = 12` — punkt centralny filtru (opóźnienie grupowe = 12 próbek)

Parametry:

Parametr	Wartość
Długość filtru N	25 współczynników
Dolna f. graniczna fc_1	5 Hz (znorm. 5/360)
Górna f. graniczna fc_2	15 Hz (znorm. 15/360)

Parametr	Wartość
Skala całkowitoliczbowa	×1000 (dzielona /1000 w RQL)

Uruchomienie skryptu:

```
cd examples/ecg/rec205
python3 gen_bp_coef.py
# Zapisano 25 współczynników do bp_coef.txt
# Współczynniki: [-2, -2, -1, 0, 3, 8, 14, 23, 32, 41, 49, 54, 56, ...]
# Suma (wzmocnienie DC): 5 / 1000 = 0.0050
```

Współczynniki są symetryczne względem centrum (n=12), co potwierdza fazę liniową filtru — niezbędną właściwość przy analizie EKG, gdyż gwarantuje brak zniekształceń fazowych morfologii QRS.

Filtr różniczkujący — `gen_d_coef.py`

Krok 2 algorytmu stosuje filtr podkreślający strome zbocza QRS. Pan i Tompkins zaproponowali 5-punktowy estymator pochodnej:

```
y[n] = (1/8T) · (-x[n-4] - 2·x[n-3] + 2·x[n-1] + x[n])
```

Współczynniki (od najstarszej do najnowszej próbki):

```
h = [-1, -2, 0, 2, 1]
```

Właściwości filtru:

Właściwość	Wartość
Suma współczynników	0 (zerowe wzmocnienie DC — eliminuje offsety)
Maksymalna odpowiedź	f ≈ 10–25 Hz (zakres zbocza QRS)
Czynnik skali (1/8T)	360/8 = 45 Hz (pomijany — nie wpływa na detekcję)

```
cd examples/ecg/rec205
python3 gen_d_coef.py
# Zapisano 5 współczynników do d_coef.txt
# Współczynniki: [-1, -2, 0, 2, 1]
# Suma (wzmocnienie DC): 0  (powinno być 0)
```

Implementacja potoku w RQL — `rec205-detect.rql`

Plik `rec205-detect.rql` implementuje kompletny pięcioetapowy potok dla dwóch kanałów EKG (MLII i V1):

```
DECLARE MLII INTEGER, V1 INTEGER STREAM ecg, 1/360 FILE 'rec205'
DECLARE bp_coef INTEGER[25] STREAM bpf, 1 FILE 'bp_coef.txt'
DECLARE d_coef INTEGER[5]   STREAM df,  1 FILE 'd_coef.txt'

# Wyodrębnienie kanałów
```

```
SELECT ecg.MLII STREAM mlii FROM ecg VOLATILE
SELECT ecg.V1   STREAM v1   FROM ecg VOLATILE

# 1. Filtr pasmowoprzepustowy (5-15 Hz) - splot FIR 25-tap
SELECT *                        STREAM mlii_win FROM mlii@(1,25)  VOLATILE
SELECT mlii_win[_]*bpf[_]       STREAM bp_acc   FROM mlii_win+bpf VOLATILE
SELECT bp_acc[0]/1000           STREAM bp_out   FROM bp_acc.sumc  VOLATILE

# 2. Różniczkowanie - splot FIR 5-tap
SELECT *                        STREAM bp_win   FROM bp_out@(1,5) VOLATILE
SELECT bp_win[_]*df[_]          STREAM d_acc    FROM bp_win+df    VOLATILE
SELECT d_acc[0]                 STREAM d_out    FROM d_acc.sumc   VOLATILE

# 3. Kwadrat (/1000 zapobiega przepełnieniu int32)
SELECT d_out[0]*d_out[0]/1000   STREAM sq_out   FROM d_out        VOLATILE

# 4. Całkowanie ruchome 30 próbek (~83 ms)
SELECT *                        STREAM mwi_win  FROM sq_out@(1,30) VOLATILE
SELECT mwi_win[0]               STREAM mwi      FROM mwi_win.avg   VOLATILE

# 5. Próg adaptacyjny - 2× średnia ruchoma 180 próbek (0,5 s)
SELECT *                        STREAM mwi_long FROM mwi@(1,180)  VOLATILE
SELECT mwi_long[0]              STREAM mwi_thr  FROM mwi_long.avg VOLATILE

# Wyjście: MLII wycentrowane, V1 wycentrowane, sygnał detekcji ×5
SELECT mlii[0]-900, v1[0]-900, (mwi[0]-mwi_thr[0]*2)*5
STREAM detect_out FROM mlii+v1+mwi+mwi_thr VOLATILE
```

Uzasadnienie parametrów

Operator `@(1,25)` tworzy ruchome okno 25 próbek, natomiast `[_]` i `sumc` realizują splot dyskretny — patrz rozdział Przetwarzanie symbolu _.

Dzielenie `/1000` w kroku 3 kompensuje skalę całkowitoliczbową współczynników — bez tego iloczyn `d_out × d_out` osiągnąłby wartości przekraczające zakres `int32` (2 147 483 647) dla typowych amplitud EKG.

Wyrażenie wyjściowe `(mwi[0]-mwi_thr[0]*2)*5` implementuje próg adaptacyjny: wartość jest dodatnia tylko wówczas, gdy obwiednia MWI przekracza dwukrotność bieżącej średniej ruchomej — co wskazuje na wykryty QRS. Mnożnik `×5` skaluje sygnał detekcji do zakresu wizualnie porównywalnego z surowym EKG na wykresie.

Uruchomienie — ninja ecg-detect-qrs

Proces uruchamia się jedną komendą z katalogu `build/Debug`:

```
cd build/Debug
ninja ecg-detect-qrs
```

CMake rozwinął ten cel do polecenia:

```
scripts/xplot.sh detect_out rec205-detect.rql 720,-400,400 --gnuplot-rtl
```

Znaczenie parametrów:

Parametr	Znaczenie
`detect_out`	Nazwa strumienia wynikowego (3 pola)
`rec205-detect.rql`	Plik zapytań z powyższym potokiem
`720`	Szerokość okna: 720 próbek = 2 sekundy przy 360 Hz
`-400,400`	Zakres osi Y w jednostkach ADC (≈ ±2 mV)
`--gnuplot-rtl`	Najnowsze próbki po prawej stronie (prawo-lewo)

Skrypt `xplot.sh` uruchamia `xretractor` w tle (kompiluje i wykonuje zapytania), a następnie przez `xqry` przekazuje strumień `detect_out` do `gnuplot` w trybie ciągłym. Okno `gnuplot` odświeża się przy każdej nowej paczce próbek.

Opis rysunku — okno gnuplot

Na rysunku widoczne są trzy sygnały odpowiadające trzem polom strumienia `detect_out`:

[detect-out-0] linia czerwona — MLII wycentrowane (mlii − 900)

Surowy sygnał EKG z odprowadzenia MLII przesunięty o punkt bazowy 900 ADC tak, że oś zerowa odpowiada izolinii. Dwa ostre piki (amplituda ≈ 280 ADC ≈ 1,4 mV) w okolicach próbek 520 i 350 od prawej krawędzi reprezentują dwa kolejne zespoły QRS. Wyraźna morfologia QRS z dominującym pikiem R potwierdza prawidłowe działanie filtru pasmowoprzepustowego — szumy zostały stłumione, a pik zachował amplitudę.

[detect-out-1] linia niebieska — V1 wycentrowane (v1 − 900)

Sygnał z odprowadzenia V1 tego samego nagrania. Morfologia QRS w V1 jest z reguły mniej wyrażona niż w MLII, co widać na rysunku — sygnał niebieski wykazuje mniejszą amplitudę piku R przy podobnych pozycjach czasowych QRS. Jednoczesna obecność obu kanałów pozwala różnicować pobudzenia nadkomorowe (APC) od komorowych (PVC), ponieważ QRS komorowe wykazują odmienną morfologię w V1.

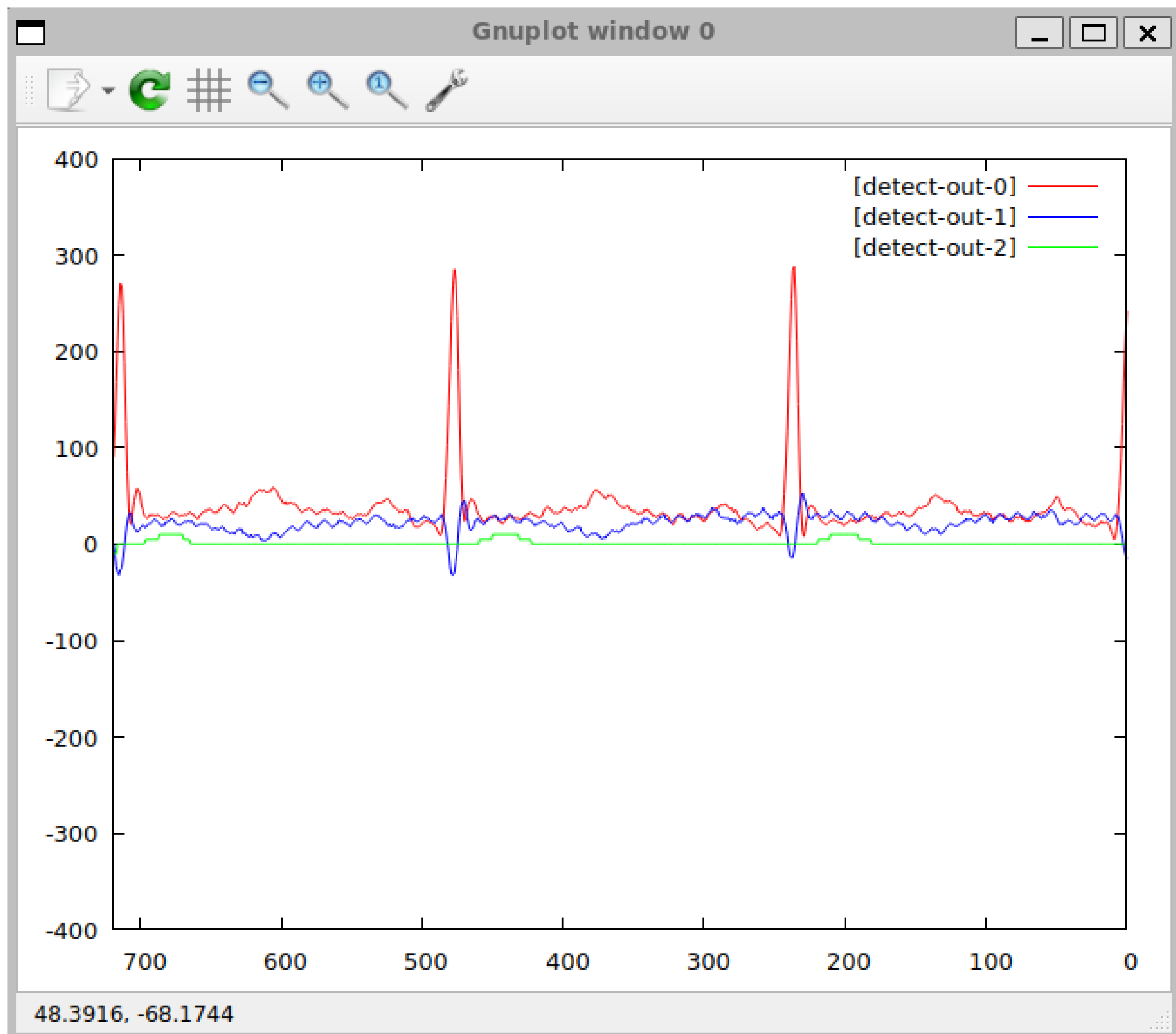
[detect-out-2] linia zielona — sygnał detekcji QRS ((mwi − 2·mwi_thr) × 5)

Sygnał wyniku algorytmu. Wartość **dodatnia** oznacza wykryty zespół QRS — obwiednia całkowania ruchomego przekroczyła dwukrotność progu adaptacyjnego. Na rysunku widoczne są dwa wyraźne dodatnie impulsy pokrywające się w czasie z pikami QRS na kanale MLII. Między uderzeniami linia pozostaje blisko zera lub nieznacznie poniżej — potwierdzając specyficzność detekcji.

Odstęp między dwoma widocznymi QRS wynosi w przybliżeniu 170 próbek, co przy 360 Hz daje:

```
RR  170 / 360  0,47 s  →  HR  127 bpm
```

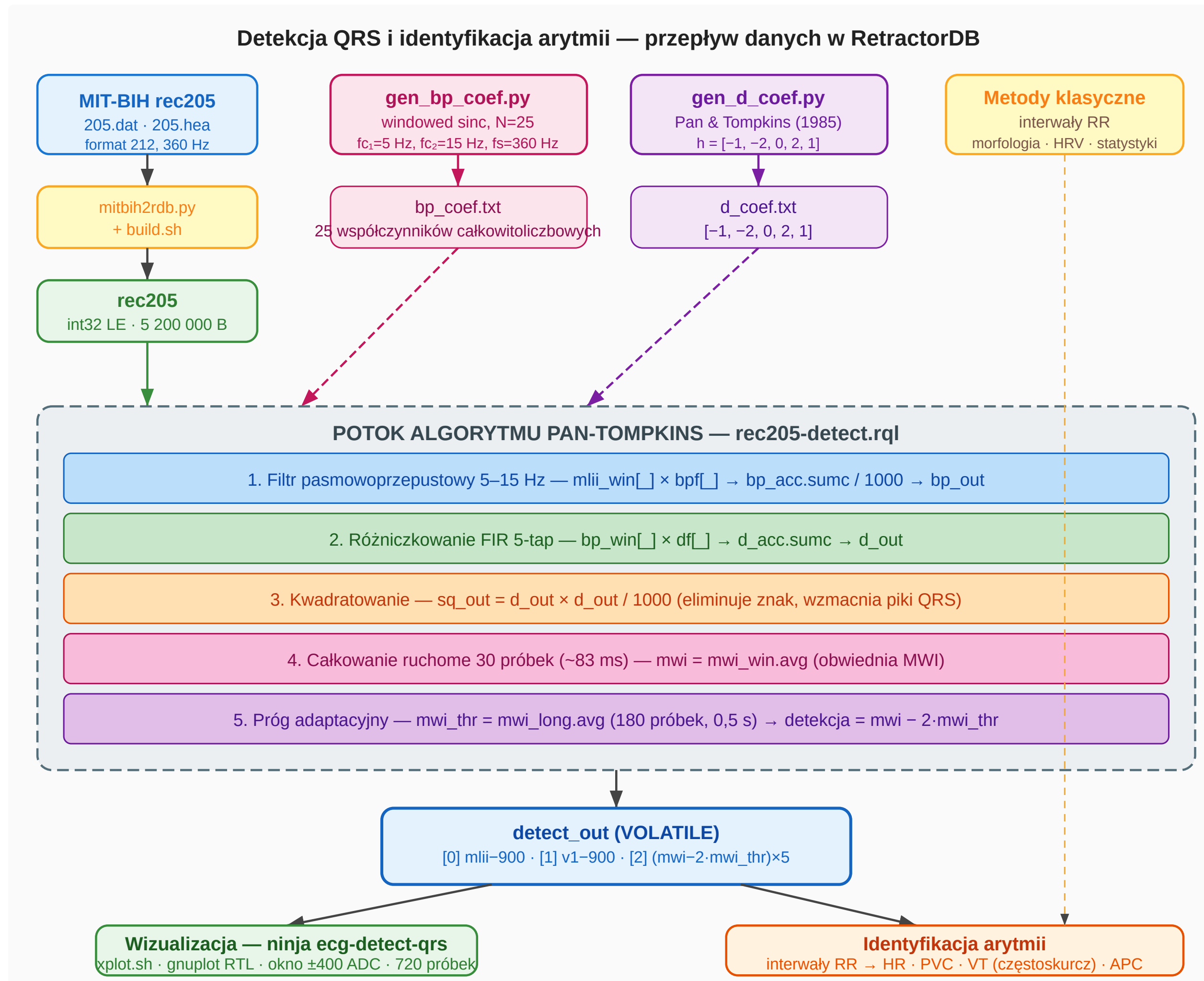
Wartość ta mieści się w zakresie odnotowanego w rekordzie 205 częstoskurczu komorowego (VT, 79–216 bpm), co sugeruje że wizualizowany fragment nagrania pochodzi z jednego z 6 epizodów VT odnotowanych przez kardiologów MIT-BIH.



Rys. 49 Okno gnuplot uruchomionego celem ninja ecg-detect-qrs — rekord 205 MIT-BIH, 720 próbek (2 s), RTL

Schemat przepływu procesu

Poniższy diagram pokazuje kompletny przepływ danych od surowego nagrania MIT-BIH do identyfikacji arytmii, ze wskazaniem miejsca, w którym RetractorDB realizuje algorytm Pan-Tompkins, oraz powiązania z klasycznymi metodami rozpoznawania arytmii:



Rys. 50 Przepływ danych — od nagrania MIT-BIH przez potok Pan-Tompkins w RQL do wizualizacji i identyfikacji arytmii

Prawa gałąź diagramu — **Identyfikacja arytmii** — reprezentuje klasyczne metody analizy po detekcji QRS, które można zbudować jako kolejne zapytania RQL nadbudowane na strumieniu `detect_out`:

Metoda	Opis	Powiązanie z QRS
Interwały RR	Czas między kolejnymi QRS → HR	bezpośrednio z pozycji detekcji
HRV (zmienność)	Odchylenie standardowe RR	statystyki strumienia RR

Metoda	Opis	Powiązanie z QRS
Klasyfikacja PVC	Szerokość QRS > 120 ms, morfologia V1	szerokość okna `mwi`
Detekcja VT	Sekwencja ≥ 3 PVC z HR > 100 bpm	RULE na strumieniu HR+PVC
Detekcja APC	Wczesny, wąski QRS poprzedzający pauzę	morfologia MLII vs V1

RetractorDB udostępnia operatory `RULE` oraz agregaty okienkowe (`.avg`, `.sumc`), które umożliwiają implementację powyższych metod w tym samym języku zapytań RQL, bez wychodzenia poza środowisko systemu. Detekcja QRS jest pierwszym i niezbędnym etapem tej hierarchii.

Rozdział 59

Załączniki

W obszarze załączników znalazły się dokumenty, które nie są związane bezpośrednio z konstrukcją systemu ale stanowią pewnego rodzaju opis motywację decyzji projektowych lub opis metod wywołania.

Rozdział 60

Opcje wywołania

RetractorDB składa się z trzech narzędzi wiersza poleceń, z których każde pełni odrębną rolę w architekturze systemu:

Narzędzie	Rola
`xretractor`	Główny proces przetwarzania: kompiluje zapytania RQL i realizuje plan
`xqry`	Klient: odpytuje działający `xretractor` przez wspólną pamięć
`xtrdb`	Narzędzie inspekcji: analizuje artefakty binarne i metadane

Każde z narzędzi opisano w osobnym podrozdziale.

Rozdział 61

xretractor

Program `xretractor` jest podstawowym procesem systemu RetractorDB. Kompiluje pliki z zapytaniami RQL i realizuje plan przetwarzania danych. Przygotowany jest do uruchomienia autonomicznego jako proces demona systemd.

61.1 Tryby pracy

`xretractor` uruchamia się w jednym z dwóch trybów:

Tryb	Opis
Przetwarzania	Domyślny — kompiluje zapytania i uruchamia pętlę realizacji zapytań
Tylko kompilacja `-c`	Kompiluje zapytania bez uruchamiania pętli; umożliwia wizualizację planu

Wywołanie `-h` pokazuje inną listę opcji w zależności od trybu — skróty opcji się nakładają, dlatego należy zwrócić uwagę, w którym trybie dana opcja funkcjonuje.

61.2 Tryb przetwarzania (domyślny)

```
$ xretractor -h
xretractor - compiler & data processing tool.

Usage: xretractor queryfile [option]

Available options:
  -h [ --help ]              Show program options
  -c [ --onlycompile ]       compile only mode
  -q [ --queryfile ] arg     query set file
  -r [ --quiet ]             no output on screen, skip presenter
```

```
  -s [ --status ]            check service status
  -v [ --verbose ]           verbose mode (show stream params)
  -x [ --xqrywait ]          wait with processing for first query
  -k [ --noanykey ]          do not wait for any key to terminate
  -t [ --realtime ]          enable real-time scheduling (SCHED_FIFO, mlockall, absolute wakeu
  -m [ --tlimitqry ] arg (=0) query limit, 0 - no limit
```

Opcje trybu przetwarzania

Opcja

Znaczenie

help

Wyświetlenie tekstu podpowiedzi. Lista różni się w zależności od trybu (z -c lub bez).

onlycompile

Przełączenie narzędzia w tryb „tylko kompilacja”. Pętla realizacji zapytań nie jest uruchamiana.

queryfile

Nazwa pliku z zapytaniami do kompilacji i uruchomienia.

quiet

Pominięcie wyświetlania wyników na ekranie. Przetwarzanie działa normalnie, ale prezenter wyników nie jest uruchamiany.

status

Sprawdzenie, czy inny proces xretractor jest uruchomiony lub pozostawił pliki blokujące wielokrotne uruchomienie.

verbose

Tryb zwiększonej komunikatywności — wyświetla parametry strumieni. Pozostałość po fazie rozwojowej; prawdopodobnie zostanie zachowana.

xqrywait

Kompiluje zapytania i wstrzymuje pętlę przetwarzania do chwili nadejścia pierwszego zapytania z procesu xqry. Wymagane przy jednoczesnym użyciu -m N w skryptach i testach: bez tej flagi serwer może przetworzyć wszystkie N cykli zanim klient zdąży się podłączyć, co skutkuje brakiem danych i oczekiwaniem po stronie xqry aż do przekroczenia limitu czasowego. Pierwsze polecenie odebrane od xqry (np. -d lub -s) odblokowuje pętlę przetwarzania.

noanykey

Dowolny klawisz nie przerywa pętli przetwarzania. Bez tej opcji naciśnięcie dowolnego klawisza zatrzymuje system.

realtime

Włącza szeregowanie czasu rzeczywistego: SCHED_FIFO, mlockall i absolutne uśpienie wątku przetwarzającego. Wymaga uprawnień CAP_SYS_NICE i CAP_IPC_LOCK (lub root). Zalecane w środowisku produkcyjnym przy wymogu deterministycznego czasu reakcji.

tlimitqry

Ogranicza liczbę iteracji w pętli realizacji zapytań. Wartość 0 oznacza brak limitu.

61.3 Tryb tylko kompilacja (`-c`)

```
$ xretractor -h -c
xretractor - compiler & data processing tool.

Usage: xretractor -c queryfile [option]

Available options:
  -h [ --help ]          show help options
  -c [ --onlycompile ]   compile only mode
  -q [ --queryfile ] arg query set file
  -r [ --quiet ]         no output on screen, skip presenter
  -d [ --dot ]           create dot output
  -m [ --csv ]           create csv output
  -f [ --fields ]        show fields in dot file
  -t [ --tags ]          show tags in dot file
  -s [ --streamprogs ]   show stream programs in dot file
  -u [ --rules ]         show rules in dot file
  -i [ --hideruleprog ]  hide rule program in rules (-u) output
  -p [ --transparent ]   make dot background transparent
  -w [ --diagram ] arg   create diagram output
```

W tym trybie dostępne są opcje tworzenia diagramów i zrzutów diagnostycznych opisywanych szerzej w opracowaniu.

Opcje wizualizacji i diagnostyki

Opcja

Znaczenie

help

Wyświetlenie tekstu podpowiedzi (identycznie jak w trybie przetwarzania, lista różni się w zależności od trybu).

onlycompile

Włączony — w tej tabeli opisano opcje obowiązujące przy aktywnej fladze -c.

queryfile

Nazwa pliku z zapytaniami do kompilacji.

quiet

Testowanie samego procesu kompilacji bez prezentowania wyników. Pozostałe opcje prezentacji nie są uruchamiane. Opcja dołączona na potrzeby rozwojowe.

dot

Tworzy plik tekstowy w formacie DOT opisujący hierarchiczne struktury wytworzone przez kompilator. Plik można przekazać do narzędzia Graphviz w celu wygenerowania graficznego opisu zależności.

csv

Eksportuje hierarchiczne struktury danych do pliku CSV (wartości oddzielone przecinkami).

fields

Dołącza do wykresu DOT pola i ich typy dla każdego strumienia danych.

tags

Dołącza do wykresu DOT programy wewnętrznego języka systemu, które tworzą pola poszczególnych zapytań. Musi być wywołana razem z fields — wizualnie łączy pola z ich programami.

streamprogs

Dołącza do wykresu DOT programy algebry strumieniowej tworzące poszczególne strumienie zapytań.

rules

Dołącza reguły alarmowania do wykresu.

hideruleprog

Ukrywa programy opisujące warunki alarmowania (używane razem z rules).

transparent

Generuje wykres z przezroczystym tłem.

diagram

Generuje diagramy kulkowe. Argument w postaci typ:ilość_cykli: typ (0 lub 1) określa, czy diagramy prezentują znaczniki czasu; ilość_cykli określa liczbę cykli na diagramie.

61.4 Informacje o wersji

Na końcu każdego komunikatu pomocy wyświetlana jest linia z informacjami o buildzie:

```
Branch: issue_31-doc:2707ce0,
Code compiler: GNU Ver. 13.3.0,
Build time: 2512211449,
Type: Debug
```

Pole	Znaczenie
`Branch`	Nazwa odnogi repozytorium i skrót commita (hash), z którego zbudowano program
`Code compiler`	Wersja kompilatora GCC użytego do budowy
`Build time`	Data i godzina kompilacji w formacie `YYMMDDHH` (tu: 21 grudnia 2025, godz. 14:49)
`Type`	Typ buildu: `Debug` lub `Release`

Kolejna linia wskazuje lokalizację pliku dziennika:

```
Log: /tmp/xretractor.log
```

Plik `/tmp/xretractor.log` rejestruje historię wywołań i zdarzeń wewnętrznych systemu. W środowisku produkcyjnym należy zadbać o regularne czyszczenie lub rotację tego pliku.

Ostatnia linia zawiera informację o licencji MIT, która umożliwia bezpieczne użycie kodu w zastosowaniach korporacyjnych.

Rozdział 62

xqry

Program `xqry` jest integralną częścią systemu RetractorDB. Dzieli z `xretractor` wspólny obszar w pamięci (Boost IPC) używany do komunikacji. Służy do odpytywania działającego procesu przetwarzania, odbioru wyników z pętli zapytań oraz sterowania pracą serwera.

W odróżnieniu od `xretractor`, `xqry` może być uruchomiony w wielu instancjach jednocześnie.

62.1 Uruchomienie

```
$ xqry -h
xqry - data query tool.

Usage: xqry [option]

Allowed options:
  -s [ --select ] arg         show this stream
  -t [ --detail ] arg         show details of this stream
  -a [ --adhoc ] arg          adhoc query mode
  -m [ --tlimitqry ] arg (=0) limit of elements, 0 - no limit
  -n [ --null ]               if null row appear - skip it in output
  -l [ --hello ]              diagnostic - hello db world
  -k [ --kill ]               kill xretractor server
  -d [ --dir ]                list of queries
  -y [ --diryaml ]            list of queries in yaml format
  -r [ --raw ]                raw output mode (default)
  -g [ --graphite ]           graphite output mode
  -f [ --influxdb ]           influxDB output mode
  -p [ --gnuplot ] arg        x,y or x,ymin,ymax - gnuplot output mode
  -h [ --help ]               produce help message
  -c [ --needctrlc ]          force ctl+c for stop this tool
  -w [ --wait-server ]        poll until xretractor server is available
```

62.2 Odbiór danych ze strumieni

Opcja	Znaczenie
`-s` / `select arg`	Odbiera dane z podanego strumienia udostępnianego przez `xretractor`.
`-t` / `detail arg`	Wyświetla szczegółowe informacje o strumieniu: nazwę, delta, treść zapytania i listę pól z typami (YAML).
`-a` / `adhoc arg`	Dołącza zapytanie do systemu w trakcie jego działania (tryb ad hoc).
`-m` / `tlimitqry arg`	Ogranicza liczbę odebranych wyników. Wartość `0` oznacza brak limitu. Szczególnie przydatne z opcją `-k`.
`-n` / `null`	Pomija wiersze, w których wszystkie pola mają wartość null. Przydatne przy strumieniach z lukami pomiarowymi — eliminuje szum w wyjściu bez filtrowania po stronie klienta.

Przykładowa odpowiedź opcji `detail`:

```yaml
---
apiVersion: xqry/v1
stream:
  name: str4
  delta: 1
  query: SELECT (str4[0]+1)*2 STREAM str4 FROM core0>1
  fields:
    str4.str4_0:
      type: INTEGER
```

62.3 Diagnostyka i sterowanie serwerem

Opcja	Znaczenie
`-l` / `hello`	Weryfikacja działania kanału komunikacyjnego z `xretractor` (ping diagnostyczny).
`-k` / `kill`	Żądanie zatrzymania procesu `xretractor`.
`-d` / `dir`	Wylistowanie wszystkich zapytań realizowanych przez `xretractor` w formacie tekstowym.
`-y` / `diryaml`	Wylistowanie wszystkich zapytań w formacie YAML.

Opcja	Znaczenie
-w / wait-server	Odpytuje co 100 ms czy xretractor jest dostępny (maks. 30 s), a po potwierdzeniu wykonuje żądane polecenie. Umożliwia niezawodne uruchamianie xqry w skryptach startowych i kontenerach, gdy kolejność startu procesów nie jest gwarantowana. Sprawdza wyłącznie dostępność IPC — nie wysyła żadnej komendy do serwera i nie wyzwala przetwarzania danych.

62.4 Formaty wyjścia

xqry obsługuje cztery formaty prezentacji danych. Format wybiera się flagą — można go łączyć z opcją select.

Opcja	Format	Zastosowanie
-r / raw	Tekstowy	Domyślny. Dane bez dekoracji — przydatny do skryptów i potokowania.
-g / graphite	Graphite	Format metryka wartość znacznik_czasu — gotowy do wysłania do Graphite.
-f / influxdb	InfluxDB	Line protocol InfluxDB — gotowy do importu do bazy szeregów czasowych.
-p / gnuplot x,y lub x,ymin,ymax	Gnuplot	Agregaty dla bezpośredniego zasilania gnuplot. Argument x,y podaje oś czasu i wartość; x,ymin,ymax dodatkowo ogranicza zakres osi Y. Separatorem może być , lub :.

62.5 Sterowanie trybem odbioru

Opcja	Znaczenie
-h / help	Wyświetlenie tekstu pomocy.
-c / needctrlc	W normalnym trybie dowolny klawisz zatrzymuje odbiór danych. Ta opcja wymaga użycia Ctrl+C.

62.6 Wzorzec uruchamiania w skryptach

Przy użyciu xretractor -m N (ograniczona liczba cykli) istnieje ryzyko wyścigu: serwer może przetworzyć wszystkie dane zanim klient zdąży się podłączyć. Gwarantowany

wzorzec:

```
# Strona serwera: -x powoduje wstrzymanie przetwarzania do czasu
# nadejścia pierwszej komendy od xqry
xretractor query.rql -m 100 -k -x &

# Strona klienta: -w sprawdza gotowość IPC bez wysyłania komend,
# więc nie wyzwala przypadkowo przetwarzania
xqry -w -s strumien -m 10
```

Flagi `-w` i `-x` są komplementarne:

Flaga	Narzędzie	Rola
`-w` / `wait-server`	`xqry`	Czeka na gotowość IPC serwera przed wysłaniem komendy
`-x` / `xqrywait`	`xretractor`	Wstrzymuje przetwarzanie do nadejścia pierwszej komendy od klienta

Bez `xretractor -x` przy strumieniach plikowych (szybkich) dane mogą zostać przetworzone w całości przed połączeniem klienta — `xqry` będzie czekał na dane, które nigdy nie nadejdą.

62.7 Informacje o wersji

Informacje na dole listy pomocy są identyczne jak w przypadku `xretractor` — zawierają nazwę odnogi repozytorium, wersję kompilatora, czas budowy oraz ścieżkę do pliku dziennika (`/tmp/xqry.log`). Opis formatu znajdziesz w rozdziale xretractor — Informacje o wersji.

Rozdział 63

xtrdb

Program `xtrdb` to interaktywne narzędzie do analizy artefaktów i substratów zapisanych przez system RetractorDB. Pracuje głównie w trybie interaktywnym (REPL), ale udostępnia także kilka opcji uruchomienia (np. `--help`, `--noprompt`, `--storagemap`).

Ostrzeżenie

Wywołanie `xtrdb` blokuje uruchomiony równolegle `xretractor` — przed użyciem `xtrdb` zatrzymaj serwer lub poczekaj na zakończenie pracy systemu. Narzędzie samo wykrywa blokadę i zgłosi błąd, jeśli `xretractor` działa.

63.1 Uruchomienie

```
$ xtrdb                      # tryb interaktywny (z promptem)
$ xtrdb -n                   # tryb wsadowy (bez promptu i bez "ok")
$ xtrdb --noprompt           # to samo co -n
$ xtrdb noprompt             # zgodność wsteczna (legacy, argument pozycyjny)
$ xtrdb -s plik_danych       # pokaż strukturę storage dla wskazanego pliku i zakończ
$ xtrdb --storagemap plik    # to samo co -s
$ xtrdb -h                   # help i informacje o buildzie, potem zakończ
```

Tryb `-n`/`--noprompt` usuwa kolorowanie, prompt `.` i komunikat `ok` — przydatny, gdy wejście pochodzi z pliku lub potoku. Wciąż działa też historyczny wariant pozycyjny `noprompt`.

```
$ xtrdb -n < script.xtrdb
```

Opcja `-s`/`--storagemap` uruchamia tylko raport struktury pliku danych i kończy działanie programu (bez wejścia do REPL).

Po uruchomieniu narzędzie wypisuje prompt `.` i czeka na polecenie. Każde polecenie kończy się naciśnięciem Enter.

63.2 Przegląd poleceń

Polecenie `help` lub `h` wyświetla listę dostępnych poleceń:

```
$ xtrdb
.help
exit|quit|q                     exit
quitdrop|qd                     exit & drop artifacts (data, .desc, .meta)
open file [schema]              open or create database with schema
                                example: .open test_db { INTEGER dane STRING name[3] }
storage [path]                  set storage path for database
policy [name]                   set storage policy
dropfile [file1] [file2] ... }  remove listed file(s), end with }
desc|descc                      show schema
read|rread [n]                  read record from database into payload
write [n]                       from payload send record to database
purge                           remove all records from database
append                          append payload to database
set [field][value]              set payload field value
setpos [position][number value] set payload field number value
getpos [position]               show payload field value
status                          show current payload status
rox                             remove on exit flip (data, .desc, .meta)
print|printt                    show payload
list|rlist [count]              print first records
input [[field][value]]          fill payload
hex|dec                         type of input/output of byte/number fields
size                            show database size in records
cap [value]                     set device stream backread capacity
dump                            show payload memory
meta                            show meta index (null patterns) for open db
metaraw                         show internal meta file structure
echo                            print message on terminal
system                          execute system command
#|rem [text]                    comment line
help|h                          show this help
```

63.3 Zarządzanie sesją

Polecenie	Opis
`exit`, `quit`, `q`	Zakończ narzędzie. Dane niezapisane w bazie pozostają na dysku.
`quitdrop`, `qd`	Zakończ i usuń otwarte pliki artefaktu (dane, `.desc`, `.meta`).

63.4 Konfiguracja środowiska

Polecenie	Opis
`storage [ścieżka]`	Ustaw katalog roboczy. Kolejne polecenie `open` szuka pliku w tej ścieżce.
`policy [nazwa]`	Ustaw politykę przechowywania (`DEFAULT`, `DIRECT`, `POSIX`, `MEMORY`, ...). Musi poprzedzać `open`.

63.5 Otwieranie artefaktu

```
open nazwa_pliku
open nazwa_pliku { TYP pole TYP pole ... }
```

Jeśli plik `.desc` istnieje — schemat jest z niego odczytany. Jeśli nie istnieje — schemat należy podać w nawiasach `{}`.

Tablicowe typy pól: `STRING name[8]` oznacza pole tekstowe o długości 8 bajtów (array multiplicity = 8).

Przykłady:

```
.open str1                          # schemat z pliku str1.desc
.open dump.tmp { INTEGER wartosc }  # schemat podany ręcznie
.open wyniki { INTEGER a FLOAT b STRING name[8] }
```

63.6 Odczyt i zapis rekordów

Polecenie	Opis
`read N`	Odczytaj rekord N (0-based) z pliku do bufora payload.
`rread N`	Jak `read`, ale odczytuje od końca pliku (reverse read).
`write N`	Zapisz bieżący payload do rekordu N w pliku.
`append`	Dołącz bieżący payload jako nowy rekord na końcu pliku.
`purge`	Usuń wszystkie rekordy z pliku (skróć plik do 0 rekordów).

63.7 Przeglądanie zawartości

Polecenie	Opis
`list N`	Wypisz N pierwszych rekordów (od początku), jeden wiersz = jeden rekord.
`rlist N`	Jak `list`, ale odczytuje od końca pliku.
`print`	Wypisz bieżący payload w formacie wieloliniowym.
`printt`	Wypisz bieżący payload w jednym wierszu.
`size`	Wypisz liczbę rekordów i rozmiar jednego rekordu w bajtach.
`dump`	Wypisz surowe bajty bieżącego payload w formacie hex.
`desc`	Wypisz schemat pól otwartego artefaktu (wieloliniowy).
`descc`	Wypisz schemat w jednym wierszu (compact).

63.8 Edycja payload

Polecenie	Opis
`set pole wartość`	Ustaw pole o podanej nazwie w buforze payload.
`setpos N wartość`	Ustaw pole o indeksie N (0-based) w buforze payload.
`getpos N`	Wypisz wartość pola o indeksie N z bieżącego payload.
`input`	Interaktywne wypełnienie payload — wpisz wartości po kolei dla każdego pola.
`status`	Wypisz stan payload: `clean`, `fetched`, `changed`, `stored`.
`hex` / `dec`	Przełącz format wejścia/wyjścia pól liczbowych między szesnastkowym a dziesiętnym.

63.9 Metadane null (`.meta`)

Polecenie	Opis
`meta`	Wypisz indeks null i przerw w transmisji z pliku `.meta` — opisowo (segmenty z liczbą rekordów i wzorcem null).
`metaraw`	Wypisz surową strukturę binarną pliku `.meta` — każdy wpis RLE z polami `count`, `gap`, `bitsetHex`.

`meta` wyświetli segmenty z informacją o brakach (`null`) i przerwach w transmisji (`gap`). `metaraw` pokaże surową strukturę binarną pliku `.meta`.

63.10 Pozostałe polecenia

Polecenie	Opis
`rox`	Przełącz flagę „remove on exit” — po zakończeniu narzędzia usuwa dane, `.desc`, `.meta`.
`cap N`	Ustaw pojemność bufora cofania (backread) dla urządzeń strumiennych.
`dropfile f1 f2 … }`	Usuń wymienione pliki. Lista kończy się tokenem `}`.
`echo tekst`	Wypisz tekst na terminal (przydatne w skryptach).
`system polecenie`	Wywołaj polecenie powłoki.
`#` lub `rem`	Linia komentarza (ignorowana). `#` nie wypisuje nawet promptu.

63.11 Przykłady użycia

Podgląd artefaktu

```
$ xtrdb
.storage temp
.open str1
.size
.list 10
.quit
```

Odczyt pliku DUMP bez deskryptora

Pliki zrzutu tworzone przez `DO DUMP` nie mają pliku `.desc` — schemat należy podać ręcznie:

```
$ xtrdb
.open wyniki_alarm_dump.tmp { INTEGER wartosc }
.size
.list 6
.quit
```

Skrypt wsadowy

```
xtrdb noprompt << 'EOF'
storage /var/retractor
open sensor_dump.tmp { INTEGER a FLOAT b }
list 20
quit
EOF
```

Inspekcja metadanych null

```
.open str1
.meta
.metaraw
```

Rozdział 64

Geneza systemu

Ponad dwadzieścia lat temu pracowałem w pewnym instytucie naukowym w Zabrzu. Zajmowałem się m.in. budową systemu nadzoru neonatologicznego. Stosunkowo niedawno ukończyłem studia, moja głowa nadal była wypełniona teorią dotyczącą budowy systemów opartych na centralnej bazie danych. Budując system monitorowania stwierdziłem – zrobię go tak jak sztuka każe – oparty na relacyjnej bazie danych. To nie był dobry pomysł. Trafiłem na problem ogólnej wydajności takiego rozwiązania. Rejestrowane sygnały cechowały się wysoką granulacją. Dodatkowo, dostępne systemy baz danych nie były przygotowane na ciągły i nieskończony napływ danych.

Rok 2003 był czasem, w którym bardzo obiecująco prezentowały się w literaturze naukowej tzw. bazy strumieniowe. Po analizie stwierdziłem, że to chyba najbliższa dziedzina w tym czasie, która odpowiada temu, czego potrzebuję. Przyjąłem założenie, że tworzę strumieniową bazę danych do przetwarzania sygnałów. Decyzja z czasem okazała się nie do końca zgodna z prawdą. Systemy strumieniowe przyszły i poszły – ale potrzeba systemów przetwarzających szeregi czasowe pozostała. Systemy strumieniowe przeobraziły się w systemy przetwarzające serie czasowe – Time Series Databases. Do dnia dzisiejszego systemy baz danych przetwarzające serie czasowe znajdują zastosowanie w systemach monitorowania.

Opracowany system nadzoru neonatologicznego obsługiwał kilkanaście pulsoksymetrów. Na sali nadzoru leżało kilkanaście noworodków wymagających ciągłego nadzoru. Każdy noworodek podłączony był m.in. do pulsoksymetru. Każdy pulsoksymetr monitorował rytm serca oraz zawartość tlenu we krwi noworodka. Noworodki się wierciły, sondy odpadały, pulsoksymetry podnosiły alarm co chwilę raportując różnego typu problemy. W takim szumie informacyjnym jeden z noworodków mógł się dusić. Nie działo się to nagle – ale powoli, można było to rozpoznać w szerszym horyzoncie czasowym. Ten jeden przypadek wymagał jednak natychmiastowej reakcji. Równocześnie i do tego bardzo głośno sygnalizowało dźwiękiem kilka urządzeń - a ten jeden z noworodków, ten, który potrzebował pomocy, łapał powietrze cichutko w rogu sali. Tak mniej więcej można opisać skalę problemu. Budowany system umożliwiał jednym rzutem oka stwierdzić, czy wycie urządzenia na sali nadzoru to efekt zsunięcia się czujnika, chwilowy problem czy może coś poważniejszego. Zmieniając skalę czasową można było od razu zidentyfikować problem. Szybka ocena zagrożenia w oparciu o wskazania systemu monitorującego w takim przypadku ratuje zdrowie i

życie.

System monitorowania powstał i został wdrożony u klienta w jednym z Warszawskich szpitali. Byłem na miejscu i widziałem, jak działa. Niestety wewnątrz nie było systemu zarządzania danymi, który opisywałem w publikacjach naukowych. Rozwiązanie opracowałem ręcznie bez implementacji języka zapytań, algorytmów i mechanizmów zarządzania. Termin i ograniczone zasoby wymagały dowiezienia tematu na czas. Publikacje, które wtedy powstały opisywały szlachetne potrzeby i założenia – jednak praktyka była inna. Trzeba było dostarczyć produkt a czasu nie było.

Tak przedstawia się w ogólnym zarysie generyczna przyczyna, z której wynikła potrzeba stworzenia systemu zarządzania danymi dla potrzeb przetwarzania sygnałów. Z czasem doszły kolejne obszary zastosowań wynikające z rozszerzających się obszarów rozwojowych związanych z telemetrią, monitorowaniem oraz rozbudową systemów IoT.

Rozdział 65

Dlaczego wybrano taką nazwę dla systemu?

Retraktory w medycynie to cała grupa narzędzi chirurgicznych. Retraktory, znane są również jako haki chirurgiczne lub rozwieraki. Są to narzędzia umożliwiające odsuwanie lub łącznie ze sobą struktur anatomicznych (np. ran, mięśni, kości, itd...). Znajdują zastosowanie z reguły podczas operacji lub innego zabiegu. Niektóre, te bardziej pomysłowe noszą nazwy swoich twórców.

Na zasadzie analogii postanowiłem że nazwę swoje narzędzie retraktorem. **RetractorDB** ma za zadanie rozdzielać, łączyć oraz umożliwiać realizację obliczeń na seriach czasowych w czasie rzeczywistym, w biegu operując na danych efemerycznych, artefaktach lub substratach (patrz podrozdział pt. Artefakty, Substraty, Efemerydy).

Info

Definicja (Retrakcja i Retraktor danych): Zastosowanie aparatu numerycznego do wydobycia, przetworzenia a następnie zwrócenia danych zawartych w seriach czasowych lub sygnałach cyfrowych nazywamy retrakcją danych. Narzędzie służące do realizacji tego procesu nazywamy Retraktorem danych.

https://youtu.be/bOabgAn15lg

Rozdział 66

Dalsze kierunki rozwoju

System RetractorDB potencjalnie może rozwinąć się w bardziej zaawansowaną formę. Poniżej wskazuję na potencjalne dalsze kierunki rozwoju.

Rozdział 67

Jeszcze inna matematyka

Kilka lat temu poszukiwałem rozszerzenia algebry przedstawionej w rozdziale o podstawach matematycznych o liczby zespolone. Bezpośrednie zastosowanie gaussowskich liczb zespolonych - zakładając, że bazą obliczeń będą liczby wymierne nie dało spodziewanych efektów. Modele obliczeniowe wskazywały na to, że rozkład zbioru liczb naturalnych w oparciu o te liczby nie działa.

Moje wewnętrze przeczucie wskazywało, że problem leży w samej naturze przetwarzanych liczb zespolonych. Moduł liczby zespolonej, której oba elementy są liczbami wymiernymi jest liczbą Rzeczywistą. Inaczej, długość przeciwprostokątnej w trójkącie, którego przyprostokątne są wyrażone liczbami wymiernymi – wyląduje w zbiorze liczb rzeczywistych.

Sytuacja nie była komfortowa. Zacząłem przeglądać literaturę. Trafiłem na coś ciekawego – liczby Eisensteina. (Proszę zwróć uwagę – nie Einsteina tylko Eisensteina). Na Wikipedii znajdziesz artykuł pt. „Liczby całkowite Eisensteina”. Eisenstein – a tak naprawdę Ferdinand Gotthold Max Eisenstein – to niemiecki matematyk, który żył jedynie 29 lat – zostawił po sobie wkład w matematykę, który możemy wykorzystać.

Liczby całkowite Eisensteina definiujemy w postaci:

$$z_C = a + b\omega \qquad a, b \in \mathbb{Z}$$

$$\omega = \frac{-1 + i\sqrt{3}}{2} = e^{\frac{2}{3}\pi i}$$

jednostka *i* jest jednostką urojoną.

Tak przedstawione liczby postanowiłem zmodyfikować w następujący sposób:

$$z_W = \frac{a}{b} + \frac{c}{d}\omega \qquad a, b, c, d \in \mathbb{Z}$$

I takie właśnie wymierne liczby zespolone użyłem do budowy algebry rozkładającej zbiór liczb naturalnych - działają.

Opracowany model numeryczny znajdziesz tutaj: https://github.com/michalwidera/equations

Rozdział 68

Kolorowanie składni RQL

Pliki zapytań RetractorDB mają rozszerzenie `.rql`. Repozytorium dostarcza gotowe definicje kolorowania składni dla trzech środowisk: Visual Studio Code, Vim oraz narzędzia `bat`/`batcat`. Wszystkie potrzebne pliki znajdują się w katalogu `scripts/` projektu.

68.1 Visual Studio Code

Rozszerzenie `rql-vscode` dodaje do VS Code pełną obsługę języka RQL: kolorowanie składni, rozpoznawanie rozszerzenia `.rql` oraz ikonę pliku.

Instalacja z repozytorium GitHub:

```
git clone https://github.com/michalwidera/rql-vscode.git
cd rql-vscode
npm install
npm run compile
code --install-extension *.vsix
```

Jeżeli repozytorium zawiera gotowy plik `.vsix`, można pominąć kompilację i zainstalować go bezpośrednio:

```
code --install-extension rql-vscode-*.vsix
```

Po instalacji VS Code automatycznie rozpozna pliki `.rql` i zastosuje kolorowanie składni. Brak konieczności modyfikacji ustawień użytkownika.

Przykład podświetlonego zapytania w VS Code:

```
STORAGE 'temp'

DECLARE a INTEGER STREAM core0, 0.1 FILE '/dev/urandom'

# Wybierz kolumnę i jej połowę
SELECT str[0], str[0] / 2 STREAM str1 FROM core0
```

```
1  STORAGE 'temp'
2
3  DECLARE a INTEGER STREAM core0, 0.1 FILE '/dev/urandom'
4
5  # Wybierz kolumnę i jej połowę
6  SELECT str[0], str[0] / 2 STREAM str1 FROM core0
```

Podswietlenie - zrzut okna

Słowa kluczowe (`STORAGE`, `DECLARE`, `SELECT`, `FROM`) są podświetlane jako komendy, typy danych (`INTEGER`) jako typy, a komentarze zaczynające się od `#` lub `//` jako komentarze.

68.2 Vim

Repozytorium zawiera dwa pliki Vima w katalogu `scripts/.vim/`:

Plik	Opis
`scripts/.vim/syntax/rql.vim`	Definicja grup składniowych i ich przypisań kolorystycznych
`scripts/.vim/ftdetect/rql.vim`	Automatyczne wykrywanie typu pliku po rozszerzeniu `.rql`

Instalacja przez buildrdb.sh

Najwygodniejsza metoda — skrypt kopiuje oba pliki do odpowiednich podkatalogów `~/.vim/`:

```
scripts/buildrdb.sh vimsyntax
```

Skrypt tworzy brakujące katalogi i informuje o lokalizacji docelowej:

```
-- RetractorQL vim syntax installed to /home/user/.vim
```

Instalacja przez CMake

Cel `vimconf` z `scripts/CMakeLists.txt` kopiuje cały katalog `.vim` do katalogu domowego:

```
cmake --build build --target vimconf
```

Instalacja ręczna

```
mkdir -p ~/.vim/syntax ~/.vim/ftdetect
cp scripts/.vim/syntax/rql.vim   ~/.vim/syntax/
cp scripts/.vim/ftdetect/rql.vim ~/.vim/ftdetect/
```

Po instalacji Vim automatycznie aktywuje kolorowanie dla każdego pliku z rozszerzeniem `.rql`. Plik `ftdetect/rql.vim` zawiera jedną linię:

```
au BufRead,BufNewFile *.rql set filetype=rql
```

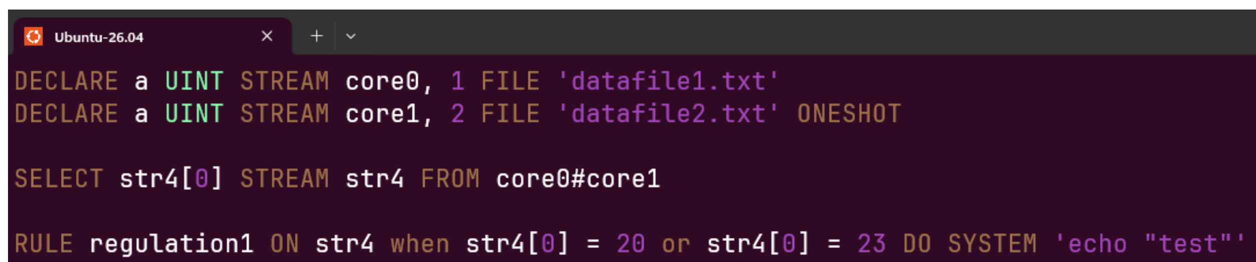
Podświetlane elementy

Grupa Vima	Przykłady
`Keyword`	`SELECT, DECLARE, STREAM, FROM, FILE, RULE, ON, WHEN, DO`
`PreProc`	`STORAGE, ROTATION, SUBSTRAT`
`Operator`	`AND, OR, NOT`
`Constant`	`MEMORY, POSIX, DIRECT, GENERIC, TEXTSOURCE`
`Type`	`INTEGER, FLOAT, BYTE, CHAR, UINT, STRING, DOUBLE`
`Function`	`MIN, MAX, AVG, Count, Sqrt, Abs, ToNumber`
`Comment`	`# komentarz, // komentarz, /* blok */`
`String`	`'ścieżka/do/pliku.dat'`
`Number`	`42, 3.14, 1/2, 1e5`

Przykład pliku zapytania z zaznaczonymi fragmentami:

```
DECLARE a UINT STREAM core0, 1 FILE 'datafile1.txt'
DECLARE a UINT STREAM core1, 2 FILE 'datafile2.txt' ONESHOT

SELECT str4[0] STREAM str4 FROM core0#core1

RULE regulation1 ON str4 when str4[0] = 20 or str4[0] = 23 DO SYSTEM 'echo "test"'
```

Widok tekstu w edytorze vim.



widok w edytorze vim

68.3 bat / batcat

Narzędzie `bat` (na niektórych dystrybucjach dostępne jako `batcat`) to ulepszony zamiennik `cat` z wbudowaną obsługą podświetlania składni. Obsługuje definicje syntaktyczne w formacie Sublime Text 3, które repozytorium RetractorDB dostarcza pod ścieżką `scripts/sublime/retractorql.sublime-syntax`.

Wymaganie wstępne

Upewnij się, że `bat` jest zainstalowany:

```
# Debian/Ubuntu
sudo apt-get install bat

# Sprawdzenie polecenia (może być bat lub batcat zależnie od dystrybucji)
command -v batcat || command -v bat
```

Instalacja przez buildrdb.sh

```
scripts/buildrdb.sh batsyntax
```

Skrypt samodzielnie wykrywa polecenie (`bat` lub `batcat`), kopiuje plik składni do właściwego katalogu konfiguracyjnego i przebudowuje pamięć podręczną syntaktyk:

```
-- RetractorQL syntax installed to /home/user/.config/bat/syntaxes
```

Instalacja ręczna

```
# Wykryj nazwę polecenia
BAT=$(command -v batcat || command -v bat)

# Utwórz katalog na definicje syntaktyk
mkdir -p "$($BAT --config-dir)/syntaxes"

# Skopiuj definicję
cp scripts/sublime/retractorql.sublime-syntax "$($BAT --config-dir)/syntaxes/"

# Przebuduj pamięć podręczną
$BAT cache --build
```

Użycie

Po instalacji `bat` automatycznie koloruje pliki `.rql`:

```
bat query.rql
```

Rozpoznawane jest też rozszerzenie `.desc` (pliki deskryptorów strumieni). Można wymusić podświetlanie ręcznie, jeśli plik ma inne rozszerzenie:

```
bat --language rql dowolny-plik.txt
```

Weryfikacja instalacji — dostępne języki:

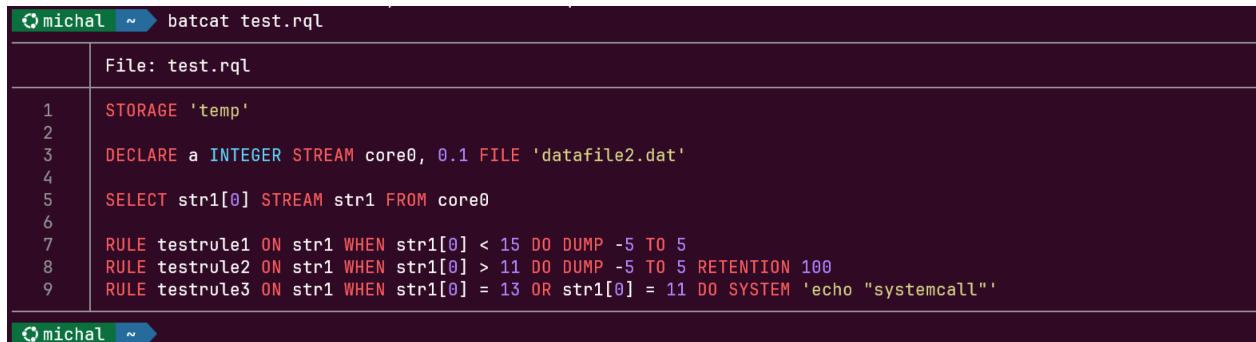
```
bat --list-languages | grep -i rql
# RetractorQL:rql,desc
```

Przykład wywołania

Dla pliku `query.rql` zawierającego:

```
STORAGE 'temp'

DECLARE a INTEGER STREAM core0, 0.1 FILE 'datafile2.dat'

SELECT str1[0] STREAM str1 FROM core0

RULE testrule1 ON str1 WHEN str1[0] < 15 DO DUMP -5 TO 5
RULE testrule2 ON str1 WHEN str1[0] > 11 DO DUMP -5 TO 5 RETENTION 100
RULE testrule3 ON str1 WHEN str1[0] = 13 OR str1[0] = 11 DO SYSTEM 'echo "systemcall"'
```

Wywołanie `bat query.rql` wyświetli zawartość pliku z numeracją linii i podświetleniem składni w terminalu, gdzie słowa kluczowe, typy, komentarze i literały łańcuchowe będą miały odrębne kolory zgodne z aktywnym motywem `bat`.



Rozdział 69

Testy integracyjne

Testy integracyjne weryfikują zachowanie systemu jako całości — uruchamiają rzeczywiste binaria (`xretractor`, `xqry`, `xtrdb`) i porównują ich wyjście z wzorcami lub sprawdzają konkretne właściwości plików wynikowych. Różnią się tym od testów jednostkowych, które za pomocą frameworka GTest testują izolowane klasy i funkcje bibliotek `rdb` i `retractor` (np. `payload`, `descriptor`, `crsMath`, `compiler`), nie wymagają uruchomionego serwera i nie produkują artefaktów na dysku. Testy integracyjne uruchamiają się poleceniem `ninja test` (lub `ctest`) w katalogu `build/Debug/`; pojedynczy test można uruchomić przez `ctest -R <nazwa> -V`.

Testy integracyjne są podzielone na dwa katalogi ze względu na wymagania dotyczące współbieżności. Testy w katalogu `IntegrationTest_serial` uruchamiają serwer `xretractor` w trybie IPC — korzystają ze wspólnego pliku blokady `/tmp/xretractor_service.lock` i segmentów pamięci dzielonej Boost. Aby uniknąć konfliktów między współbieżnymi instancjami, CMake wymusza na nich tryb `RUN_SERIAL TRUE` (jeden po drugim). Testy w katalogu `IntegrationTest_parallel` nie uruchamiają serwera IPC — kompilują zapytania (`xretractor -c`) lub wykonują operacje na plikach przez `xtrdb` — i mogą bezpiecznie działać równolegle.

69.1 Testy sekwencyjne — `IntegrationTest_serial`

Nazwa testu	Opis
`agse1`	Operator okna czasowego `@(start, length)` — warianty do przodu `@(1,4)`, wstecz `@(1,-4)`, różne długości. Patrz: *Ruchome okno danych AGSE*.
`agse2`	Kombinacje okna `@(n,m)` na strumieniu 3-polowym, wyrównanie i rate-conversion przy proporcjach 1:1, 1:2, 2:3, 2:4. Patrz: *Ruchome okno danych AGSE*.
`agse3`	Operator `@(n,m)` gdy output rate jest niższy niż input rate (source rate 0.1) — okna `@(3,2)`, `@(3,3)`, `@(3,-3)`. Patrz: *Ruchome okno danych AGSE*.

Nazwa testu	Opis
`consistency`	Spójność odczytu: dwa strumienie czytają to samo źródło; ich różnica musi być stale równa 100. Patrz: *Przepływ danych i sterowania*.
`issue113_meta_internal`	Struktura pliku sidecar `.meta`: rozmiar nagłówka (8 B), rozmiar wpisu (18 B), interwał próbkowania, bitsety null dla rekordów z null i bez. Patrz: *Format zapisu danych — Pliki*.
`issue113_meta_xtrdb`	Weryfikacja przez `xtrdb` że po uruchomieniu `xretractor`+`xqry` plik `.meta` powstaje i jest raportowany poprawnie (`meta: temp/str_null.meta`). Patrz: *Format zapisu danych — Analiza artefaktów*.
`issue113_null_skip`	Flaga `-n` w `xqry` — wiersze w całości null są pomijane; bez flagi wszystkie wiersze (łącznie z all-null) muszą być obecne. Patrz: *Opcje wywołania — xqry*.
`issue113_null_xqry`	Null przesyłane przez IPC: wartości null z pliku źródłowego wyświetlane jako `null` w wyjściu `xqry`. Patrz: *Opcje wywołania — xqry*.
`issue121_isnull`	Funkcja `isnull(field)` — zwraca 1 gdy pole jest null, 0 gdy nie jest. Patrz: *Operatory agregujące i to_string*.
`issue121_null_propagation`	Propagacja wartości null przez `SELECT` do strumienia wynikowego. Patrz: *Polecenie SELECT*.
`issue128_numeric_to_string`	Konwersja `INTEGER`/`FLOAT` do `STRING` funkcją `to_string()` z deklaracją szerokości pola; weryfikacja deskryptora wynikowego. Patrz: *Operatory agregujące i to_string*.
`issue128_string_to_numeric`	Konwersja `STRING` do typów numerycznych: `to_integer()`, `to_float()`, `to_double()`; propagacja null przez konwersję. Patrz: *Operatory agregujące i to_string*.
`issue167_dedup_cascaded`	Kaskadowe wchłanianie substratów przez `deduplicateSubstrats()` — wieloetapowe przepisywanie tokenów `PUSH_ID`. Patrz: *Substraty*.
`issue167_dedup_field_names`	Deduplikacja substratów bez porównania nazw pól schematu — scalanie gdy typy pól są równoważne, niezależnie od nazw. Patrz: *Substraty*.
`issue167_dedup_nonzero_offset`	Aktualizacja `PUSH_ID` w `lSchema` konsumenta przy niezerowym offsecie wchłanianego substratu; pokrycie ścieżki w `compiler.cpp`. Patrz: *Substraty*.
`issue167_dedup_positive`	Podstawowy przypadek deduplikacji: substrat scalany z nazwanym strumieniem o równoważnym programie i typach pól. Patrz: *Substraty*.
`issue167_triarg`	Wieloargumentowe wyrażenia strumieniowe: `s1+s2+s3`, `(s1#s2)#s3`, `s1+(s2#s3)`, `s1+s2+s3+s4`; substraty pamięciowe i dyskowe. Patrz: *Substraty, Sekwencjonowanie operacji*.
`issue42_rule`	Polecenie `RULE` — warunkowe akcje `DUMP` i `SYSTEM` wyzwalane na wartościach strumienia; uruchamia `xretractor` i odczytuje wynik przez `xtrdb`. Patrz: *Polecenie RULE*.

Nazwa testu	Opis
`issue56_timeshift`	Operator filtra `>` na połączonych strumieniach — do wynikowego strumienia trafiają tylko rekordy spełniające warunek. Patrz: *Polecenie SELECT — Sekwencjonowanie*.
`issue61_tmpmem`	Substrat pamięciowy `SUBSTRAT 'memory'` — dane pośrednie przechowywane w RAM zamiast na dysku. Patrz: *Typy STORAGE*.
`issue6_adhoc`	Tryb zapytania ad-hoc: `xqry -a 'SELECT ...'` — definicja i wykonanie zapytania w locie bez pliku `.rql`. Patrz: *Zapytania Ad hoc*.
`operations`	Operator `#` (HASH merge) dwóch strumieni o różnych rate — weryfikacja stosunku liczby rekordów w wyjściu. Patrz: *Sekwencjonowanie operacji przeplotu*.
`rotation_test`	Mechanizm rotacji plików binarnych strumieni (`ROTATION`) — liczba plików po dwóch cyklach `xretractor -m 2`. Patrz: *Mechanizm rotacji*.
`simple`	Dymny test arytmetyki na połączonych strumieniach (`core0` rate 0.1 + `core1` rate 0.2) z odczytem przez `xtrdb`. Patrz: *Polecenie SELECT*.
`simple_max`	Operator `.max` na strumieniu — wartość maksymalna i złączenie z oryginalnym strumieniem. Patrz: *Operatory agregujące i to_string*.
`xqry_elem_limit`	Parametr `-m N` w `xqry` — limit liczby odebranych rekordów do dokładnie N, niezależnie od długości źródła. Patrz: *Opcje wywołania — xqry*.

69.2 Testy równoległe — `IntegrationTest_parallel`

Nazwa testu	Opis
`dsp`	Regresja dla potoku filtra FIR: okno przesuwne `@(1,25)`, mnożenie tablicowe indeksem `_`, redukcja `.sumc`, złączenie sygnału i wyjścia. Patrz: *Implementacja filtru sygnałowego*.
`issue113_meta`	Operacje `xtrdb` po dwóch `append` — lista rekordów i hexdump pliku binarnego porównywane ze wzorcem. Patrz: *Format zapisu danych — Analiza artefaktów*.
`issue113_meta_autocreate`	Automatyczne tworzenie pliku sidecar `.meta` po pierwszym `append`; rozmiar >16 B; `xtrdb` raportuje poprawną ścieżkę. Patrz: *Format zapisu danych — Pliki*.
`issue113_null_txtsrc`	Komendy `rread`/`getpos` w `xtrdb` na strumieniu TEXTSOURCE zawierającym null. Patrz: *Format zapisu danych — Analiza artefaktów*.
`issue153_storagemap_meta_cases`	Mapa składowania `xtrdb -s` dla pliku zwykłego i retractordb-style: znaczniki slotów, lista segmentów, pliki rotowane, referencje `.meta`/`.shadow`. Patrz: *Narzędzie inspekcji xtrdb -s*.

Nazwa testu	Opis
`issue31_doc`	Generowanie grafów DOT/SVG przez `xretractor -c -d ...` dla przykładów dokumentacyjnych na trzech poziomach szczegółowości. Patrz: *Debugowanie kompilacji*.
`issue42_rule`	Kompilacja składni `RULE` — tylko etap `-c`, bez uruchamiania serwera. Patrz: *Polecenie RULE*.
`issue56_timeshift`	Kompilacja operatora filtra `>` — tylko etap `-c`. Patrz: *Polecenie SELECT — Sekwencjonowanie*.
`issue61_tmpmem`	Kompilacja zapytania z `SUBSTRAT 'memory'` — tylko etap `-c`. Patrz: *Typy STORAGE*.
`issue95_loopInCompile`	Wykrywanie cykli w grafie zapytań przez kompilator — oczekiwany błąd "Circular dependency" i niezerowy kod wyjścia. Patrz: *Wykrywanie pętli w kompilacji*.
`issue96_no_substrat_redundant`	Strumienie zdefiniowane przez użytkownika o identycznej strukturze NIE są scalane; scalaniu podlegają tylko automatyczne substraty. Patrz: *Substraty*.
`issue96_substrat_reference`	Wygenerowany substrat współdzielony przez dwa strumienie użytkownika — poprawne referencje w drzewie zależności. Patrz: *Substraty*.
`Pattern1`	Kompilacja operatora `#` (HASH-merge), selekcji pól z przesunięciem i łączenia strumieni `+`. Patrz: *Sekwencjonowanie operacji przeplotu i sumowania*.
`Pattern2`	Kompilacja zapytań na strumieniach `BYTE` z `/dev/urandom`: `SELECT`, arytmetyka, łączenie strumieni. Patrz: *Polecenie SELECT*.
`Pattern3`	Kompilacja `SELECT *` (unfold) z deklaracją pliku wyjściowego i retencją. Patrz: *Rozwijanie symbolu **.
`Pattern4`	Kompilacja funkcji `Crc(bits, seed)` w wariantach 16-bit i 8-bit na strumieniu 2-polowym. Patrz: *Operatory agregujące i to_string*.
`Pattern5`	Operacje `xtrdb` na rekordzie wielotypowym (STRING, INTEGER, BYTE, FLOAT): `append`, `read`, `list/rlist`, `input`, `write`. Patrz: *Analiza artefaktów*.
`Pattern6`	Kompilacja operatora okna `@(n,m)` do przodu `@(1,10)` i wstecz `@(1,-10)` + valgrind bez wycieków. Patrz: *Ruchome okno danych AGSE*.
`Pattern7`	Kompilacja z identycznymi nazwami pól w wielu strumieniach (issue #17) — poprawna identyfikacja pola przez indeks strumienia. Patrz: *Polecenie DECLARE, Aliasowanie*.
`retention`	Operacje `xtrdb` na pliku z parametrem `RETENTION`: `open`, `purge`, `append`, `list`, `write` dla konkretnego rekordu. Patrz: *Format zapisu danych — Mechanizm rotacji*.
`simple`	Kompilacja podstawowego zapytania arytmetycznego + graf DOT + valgrind; korzysta z danych `IntegrationTest_serial/simple`. Patrz: *Polecenie SELECT*.

Nazwa testu	Opis
`simple_max`	Kompilacja zapytania z `.max` + graf DOT + valgrind; korzysta z danych `IntegrationTest_serial/simple_max`. Patrz: *Operatory agregujące i to_string*.
`subquery`	Kompilacja zagnieżdżonych podzapytań: `(a#b)>1` (hash-merge wewnątrz filtru) i `(a>1)#b` (filtr wewnątrz hash-merge). Patrz: *Budowa drzewa zależności*.
`txtsrc`	Operacje `xtrdb descc/rread/printt` na strumieniu TEXTSOURCE (plik tekstowy jako źródło danych). Patrz: *Format zapisu danych*.

Rozdział 70

Literatura

1. S. Beatty, “Problem 3173” American Mathematical Monthly, vol. 33, p. 159, 1926.

2. A.S.Fraenkel, „The bracket function and complementary sets of integers” Canadian Journal of Mathematics, tom 21, pp. 6-27, 1969. (link)

3. M. Widera, „Deterministyczna metoda przetwarzania ciagow danych” w XXI Autumn Meeting of Polish Information Processing Society, 2006. (pdf) (link)

4. Z. W. Zen, „Classified publications on covering systems” updated 2006. [Online]. Available: http://maths.nju.edu.cn/~zwsun/. (pdf)

5. T. Parr, The Definitive ANTLR 4 Reference, The Pragmatic Bookshelf, 2013. (amazon)

6. T. D. Pauw, „Swirly - A marble diagram generator.” 2022. [Online]. Available: https://github.com/timdp/swirly. [Data uzyskania dostępu: 3 11 2025]. (link)

7. A. Staltz, „RxJS Marbles” https://github.com/staltz/rxmarbles, [Online]. Available: https://rxmarbles.com/. [Data uzyskania dostępu: 4 11 2025]. (link)

8. „Conan.io - the Open Source C and C++ Package Manager for Developers” JFrog, [Online]. Available: https://conan.io/. [Data uzyskania dostępu: 9 11 2025].

9. D. W. Gunness, „Creating digital signal processing (DSP) filters to improve loudspeaker transient response”. US Patent US8081766B2, 20 12 2011. (link)

10. M. Widera, „RetractorDB - separator serii czasowych” Programista, tom 92, nr 5/2020, pp. 14-20, 6/7 2020. (ebookpoint)

11. J. Shallit, „A Generating Function Technique for Beatty Sequences and Other Step Sequences” Journal of Number Theory, tom 64, nr 2, pp. 273-298, 1997.

12. L. Schaeffer, J. Shallit i S. Zorcic, „Beatty Sequences for a Quadratic Irrational: Decidability and Applications” arXiv:2402.08331, 2024. (pdf)

13. M. A. Berger, A. Felzenbaum i A. S. Fraenkel, „Disjoint covering systems of rational Beatty sequences” Journal of Combinatorial Theory, Series A, tom 42, nr 1, pp. 150-153, 1986.

14. D. Eppstein i in., „Aperiodic pinwheel scheduling using Beatty sequences” – omówienie problemu szeregowania okresowego w oparciu o komplementarne sekwencje Beatty'ego, 2023. (link)

15. „Pinwheel Scheduling with Real Periods” arXiv:2510.24068, 2026 – dowody oparte na podziale Rayleigha/Beatty'ego z tożsamościami na funkcjach podłogi i sufitu. (html)

16. S. Samadi, M. O. Ahmad i M. N. S. Swamy, „Characterization of nonuniform perfect-reconstruction filterbanks using unit-step signal” IEEE Transactions on Signal Processing, tom 52, nr 9, pp. 2490-2499, 2004. (link)

17. G. Margolis i Y. C. Eldar, „Nonuniform Sampling of Periodic Bandlimited Signals” IEEE Transactions on Signal Processing, tom 56, nr 7, pp. 2728-2745, 2008. (pdf)

18. J. Kovačević i M. Vetterli, „Perfect Reconstruction Filter Banks with Rational Sampling Factors” IEEE Transactions on Signal Processing, tom 41, nr 6, pp. 2047-2066, 1993.

19. S. Kalra i N. K. Shukla, „Ramanujan sums in signal recovery and uncertainty principle inequalities” arXiv:2512.16190, 2025. (pdf)

20. A. Arasu, S. Babu i J. Widom, „The CQL continuous query language: semantic foundations and query execution” The VLDB Journal, tom 15, nr 2, pp. 121-142, 2006. (pdf)

21. J. Krämer i B. Seeger, „Semantics and implementation of continuous sliding window queries over data streams” ACM Transactions on Database Systems, tom 34, nr 1, pp. 1-49, 2009 (system PIPES).

22. S. K. Jensen, T. B. Pedersen i C. Thomsen, „Time Series Management Systems: A Survey” IEEE Transactions on Knowledge and Data Engineering, tom 29, nr 11, pp. 2581-2600, 2017. (pdf)